Don't build it.

A GUIDE FOR PRACTITIONERS IN CIVIC TECH / TECH FOR DEVELOPMENT

✕ This guide aims to help you avoid bad projects, structure the team right, ship and learn quicker, and mature longer.

Luke Jordan / Grassroot and MIT Governance Lab / **2021**

GRASSROOT

MITGOV/LAB

Suggested Citation:

Jordan, Luke. 2021. “Don’t Build It: A Guide For Practitioners In Civic Tech / Tech For Development”, Grassroot (South Africa) and MIT Governance Lab (United States).

About

Luke Jordan is the Founder and CEO/CTO of Grassroot, a civic technology organization based in South Africa, and the 2021 practitioner-in-residence at MIT GOV/LAB. *Contact: luke.jordan@grassroot.org.za.*

Grassroot

Grassroot is a tech platform built for low-bandwidth, low-data settings that allows for smart-messaging through text message (www.grassroot.org.za).

MIT Gov/Lab

MIT Governance Lab is a group of political scientists focusing on innovation in citizen engagement and government responsiveness (www.mitgovlab.org).

Thanks

Thanks to the entire Grassroot team for suffering the lessons learned in the guide, including Katleo Mohlabane, Busani Ndlovu, Mbalenhle Nkosi, March Ratsela, Zinhle Miya, and Paballo Ditshego. And, thanks to Tiago Peixoto, Arjun Khoosal, Lily L. Tsai, Alisa Zomer, Selmah Goldberg, and Maggie Biroscak for providing input into versions of this guide. Illustrations and graphic design by Susy Tort and Gabriela Reyagadas.

PREFACE

This guide is for teams or managers involved in considering or building "civic technology", i.e., technology that helps people engage government more effectively. It is the distillation of my four years spent building Grassroot, a civic tech platform in South Africa.

The guide is focused on the practical. I have chosen the topics by reflecting on what people have asked for advice on over the years; on what I wish I knew when I started, or on what early advice to me was most valuable; and on some of the things that went wrong along the way.

Since software provides in itself no guardrails against building what should not be built, an organization or leadership team needs to develop its own precautions. But that is very hard when all around you people are pretending to build cool new apps and one article after another is talking breathlessly about supposed "technology for good". As proof of these forces, we can observe that for half a decade one research report after another has pointed to the limited effect (if any) of well-intentioned but insufficiently rigorous technology projects ("let's build an app"). And despite all of that research, the apps keep being built.[1]

That brings you to my motivation for writing this guide. I believe that technology can help ordinary people build power and make the state more accountable and responsive. I believe that, when targeted at the right problem at the right time, it can make an enormous difference. I've also seen close-up how the forces of contemporary thought, funding and status will push you towards building what should not be built, with teams who don't know how to build it. You'll notice the tone isn't typical of academic how-to guides—my approach is to describe the process honestly and realistically, with hopes that it will give people a better sense of what "building an app" entails, and how they can do it well, or (better yet) not do it in the first place.

[1] Among others: https://opendocs.ids.ac.uk/opendocs/bitstream/handle/20.500.12413/13452/RR_Synth_Online_final.pdf.

Don’t build it

The guide starts with project selection, including why the best project to select is no project at all. It moves on to team structure, and the extreme importance of a full-time senior tech lead (or chief technology officer (CTO), understood as an excellent engineering manager). It then covers timelines, emphasizing shipping early but having enormous patience getting to maturity, above all in finding product-use-fit, and avoiding vanity metrics. The guide then goes into some detail on hiring, covering the CTO role, senior contractors, designers and young engineers.

The longest section, by some distance, is that on hiring. Hiring is the one thing considered critical in every piece of the lore, by founders and investors and managers alike, across all sectors. It is also the field in which I think I got it mostly right, and for reasons I can explain in ways that I believe will be helpful.

EXECUTIVE SUMMARY

If you just remember these...

If you can avoid building it, don’t build it; if you have to build it, hire a CTO, ship early, and mature long; and no matter what, draw on a trusted crew, build lean and fast, and get close to and build with your users as soon as possible.

Don’t build it

IF YOU HAVE TO:

- *Outsourcing is great as a tactic, but a terrible strategy.*
- *Add full-time talent cautiously, at cost levels where you can keep them in the team and invest in their growing skills over time.*
- *Get close to your users and to do so fast with a dedicated community engagement team.*
- *Adaptability and speed of learning are core criteria in every role.*
- *Set a budget that gets you off to a quick start, but allows you to keep iterating over time.*

A rule for this sort of guide is never give specifics that people can hang around your neck. A rule for software development is “it takes as long as it takes”. I’m going to break both rules, and say that for a good team of 3-4, following good practices, you should be able to ship a moderately complex product in about 3 months, and get it to some form of stability and maturity within 12 months. After that, of course, you start again—if you have users. If you don’t, stop at 6 months, or better yet, 4 months, or best yet—**don’t build it.**

> Tech principles: stick to open source and pick a popular language.

> Nothing is more important than rapid early learning, and vanity metrics short-circuit such learning.

DOES THIS PROJECT NEED TO HAPPEN?

↳ *PROBABLY NOT*

The central problem of software is that anything can be built.

The central problem of software is that anything can be built. With a physical structure, nature and physics puts some constraints on the space of ideas. With software, a project can be wholly completed and deployed and only then reveal itself as fundamentally flawed, and then we are all so inured to bad technology that no one will really notice. **Construct a monstrous building** in the middle of nowhere and movies might be made about you; build a pointless app that no one uses and you will just need to cite a misleading metric in a donor report and no one will care. Conversely, construct a good building in a sensible place and no one will think it worthy of notice; build a not-terrible app that people use for longer than the launch press release circulates, and you will immediately be nominated to half a dozen “X under X” lists.

So, by far the best method is to adopt a simple principle:

Don’t build it.

When someone says,

“We should build some tech for that”

Just say no.

When an investor or donor says,

“Why don’t you build some technology”

Just say no.

When you read another article or see another TEDx talk about someone pretending their app achieved something, while citing numbers that are both unverified and meaningless, and a voice inside says,

“Why don’t we also build technology”

Just say no.

Does that mean that the rest of this guide is pointless?

Hopefully so. But in reality, at some point some idea may gather such momentum or such force of conviction that the **“do not build it”** ethos will start to falter. At that point, ask these questions:

Are people already trying to do what the technology is supposed to help them do?

❯ **If yes, how are they doing it now, and are you sure you know why that does not work? And why will technology make any difference to the reason their existing attempts are frustrated?**

✖ **If not, why would having technology make a difference? Why would someone who did not want to do X now want to do X just because some tech exists to do it?**

Of course, it is easy to fake answers to such questions in a way that justifies building something, and that will happen most of the time. But suppose they are answered honestly and it turns out that, for example, people ***are*** trying to do whatever the technology is supposed to do, and it is not working because of some fundamental problem.

The strong temptation will be to immediately try to use technology to work on that deeper problem. But, again, **don’t build it!** First ask the questions above about this fundamental problem you’ve discovered. For example, before you build a piece of software to help a government know about a phenomenon (violence, or service outages), ask: Aren’t people already trying to tell them about this? If they are being ignored, why are they being ignored? If it’s because of power imbalances, will your well-meaning alert technology *really* do anything about that? Or might it be irrelevant, or even worsen the problem by allowing the powerful to pretend (to themselves and to other elite stakeholders) that they are doing something? If no one is attending government meetings, is that because they don’t know about the meetings, or because when they attend no one listens to them? Such examples could be multiplied almost endlessly.

If, after all of this not-building, you come to a problem where it is very clear that: a) people are trying to do whatever the tech is supposed to let them do, but it is not working; b) it’s not working *primarily because of some problem that technology can address*; and, c) the grounds for this are *clear* and *certain*... then it might be time to start considering possibly building something.

ALRIGHT YOU HAVE TO BUILD IT. BUT WHO WILL DO THE BUILDING?

↳ *PROJECT & TEAM*

So let's work on the premise that you have gone through the steps above and discovered a truly worthwhile technology project. The next question is: Where will you find, and how will you organize, the people who will design, code, deploy, monitor, and iterate the project?

Outsourcing is great as a tactic, but a terrible strategy.

The first impulse for most non-technical organizations will be to try to outsource most of the work. Outsourcing is great as a tactic, but a terrible strategy. That is, outsourcing individual components of work to highly-experienced, focused senior developers and designers can provide enormous bang-for-the-buck. They provide significant flexibility, can come in and out of projects without eating up budget on idle time, and modern top-end freelancer platforms allow you to find them for very specific needs. In contrast, outsourcing as a whole makes more or less the entire project's success dependent on the single decision point of what contractor you hire, and then you're stuck. If a single freelancer doesn't work, you can swap them out and find someone new, often without interrupting the rest of the team; if a whole outsourcer doesn't work, you have to replace the team, halt most of the work, and may even face a legal dispute over breaking the contract.

Don’t build it

The reason is that, unless an organization has a competent **Chief Technology Officer** (CTO) —more on that below— you will likely do a terrible job of choosing a contractor. Without the ability to judge the technical merits of proposals, or to break down a project into its abstract technical components, you will either under- or over-specify the requirements. Under-specifying will lead to lousy delivery with no contractual recourse; over-specifying will lead to even worse delivery (because no one will be able to adapt as you build), with the false promise of contractual recourse.

Perhaps a friendly technical specialist will provide input on the specifications, but it is usually obvious to an experienced eye when the primary drafter of a request for proposal (RFP) or other bid document is not technically versed. A whole cottage industry of consultants looks for such RFPs, and they’re not exactly who you want building the project you can’t avoid building (since, if the project is optional, you shouldn’t be building it).

Once the RFP is out, without a technical network, how will you search for contractors to invite? And once contractors bid, how will you evaluate them, both at selection and at the initial milestones? Is the friendly developer who put some track changes in the technical annex really going to do all that? If not, you'll be reduced to selecting based on a portfolio and stated expertise. Given the frequency with which non-technical organizations hire bad technical contractors, hiring on a portfolio without the ability to technically interrogate it is worse than random selection, because you'll be selecting on salesmanship, and that is, if anything, inversely correlated with technical ability.

Every now and then, some organization does luck out, and a limited project finds the perfect contractor and it works. But a strategy that fails except when extremely lucky is more or less the definition of a terrible strategy.

Chief Technology Officer (CTO)

Some organizations may worry about how they will bear the cost of a CTO between projects. If that is a worry, **don't build it**—your organization isn't ready to sustain a good tech product. Having that worry means you're assuming that technology is built as a once-off project, is complete once deployed, and will not need ongoing work to iterate and improve it. And if that's the case, then either the problem is not understood (go back to step 1), or the technology itself is not understood, and the project is doomed from the start.

Of course, resources may change over time. But for at least several years after a useful tech project begins, it will require at least half or more (most often more) of the time of a good CTO. Just like you shouldn't adopt a puppy if no one in your house wants to walk it or feed it when it's no longer cute, you shouldn't embark on a tech product unless you have a qualified CTO who can take care of it when it grows up.

In the rare circumstances that a project is so mature that a CTO has spare capacity, they can, for example, work on internal technology upgrades (the kind that many organizations during the pandemic realized they should have done years ago), or engage in cost-sharing collaborations with other partners. In most organizations, a skilled, high-quality technology manager will always be useful.

Full-time technology staff

For the rest of the team, let the skill profiles follow the product. Add full-time talent cautiously, at cost levels where you can keep them in the team and invest in their growing skills over time. Leave enough budget to bring in exceptional talent to do short-term, niche-technology, high-value chunks of the project. For example, you might combine a good but junior front-end developer with a month or so of an extremely skilled senior front-end person, who can lay down the structure of the code and the first core components, and mentor the junior staffer who can build out the remainder.

Having full-time people, even if very junior, for a long stretch of time, is worth the investment. Your organization can internalize knowledge and retain it over time. Senior contractors bring deep knowledge of their domain; in-house developers bring deep knowledge of the system; one without the other is likely to fail, like a clinic that tries to rely exclusively on in-house doctors, or dispense with them entirely.

There are many different combinations, but the idea is to optimize for continuity at the base, apply extremely high skills where it matters (even if in short bursts), and leave a healthy margin of error.

Add full-time talent cautiously, at cost levels where you can keep them in the team and invest in their growing skills over time.

DEFINITIONS

Technology stack: The combination of technologies that make up a product or project (which are, metaphorically, stacked on top of each other to create something useful or delightful).

For example, a great team structure might look like a raw junior developer, a very good technical lead on the most important parts of the stack , and a bench of senior freelancers specializing in a specific part of **the stack**, coming on for a few weeks or days at a time, with a budget reserve left for emergency help when needed. Likewise, solely to illustrate, a terrible team structure might consist of three mid-tier generalists, experienced in the wrong parts of the stack, and no reserve left (Figure 1). The one commonality is that no matter how the team is assembled, the project will likely end badly unless you have a CTO.

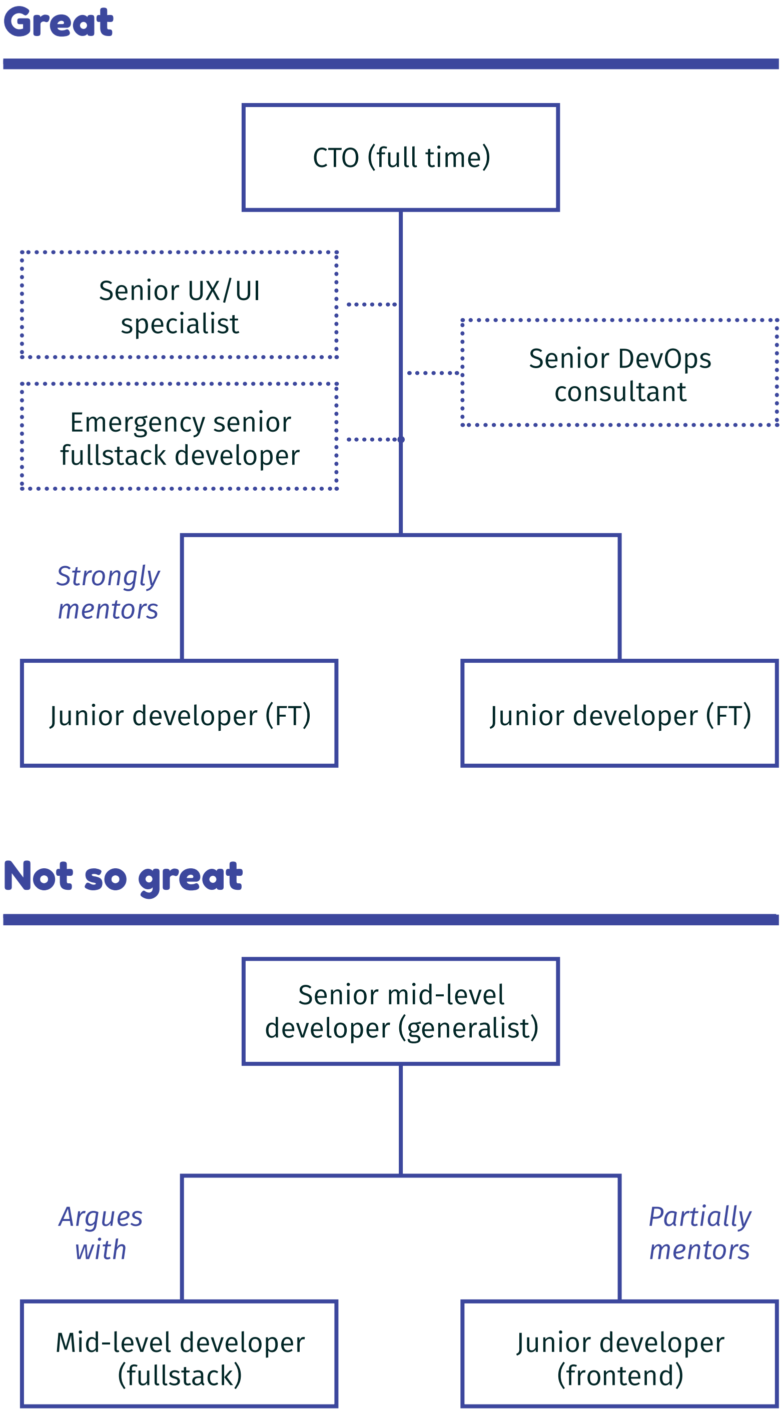


FIGURE 1 →

Connecting developers, user experience, and field staff

A final note on team composition: you need people who can build the technology, but you also need people who understand the context in which the technology will be used, i.e., the countries, communities, organizations or demographics that form its audience.

Part of that understanding should be built with “user experience” (UX) techniques, by skilled UX contractors (see the section in Hiring, below). In time, you may be able to justify hiring a full time UX specialist. But, especially at the start, it’s unlikely that you will have the base load of demand to justify hiring a UX specialist full-time, and probably not the budget to have one on-demand via a retainer. The more that the field team understands how features are built (and, ideally, have been trained in some UX techniques themselves), the more they can adapt their observations to what is likely most useful to the development team. The more the engineering team understands what is happening in the field, the more likely they are to propose new ideas or alternate directions for what they build (or have healthy second thoughts).

That said, contextual understanding should not be a hard and fast rule, especially in more senior hires. In general, adaptability and speed of learning are core criteria in every role. An adaptable, fast learner can design or build for a new context, even if they lack former field experience. Conversely, a slow learner might have a contextual understanding that is outdated, will be slow to update that understanding, and may be resistant to viewpoints that differ from their own.

Adaptability and speed of learning are core criteria in every role.

For very junior roles, you can and should weigh context heavily, by prioritising young developers from the communities you are building for. An old Confucian saying is that “the streets are full of sages”. The streets are also full of developers, if you put in the work to look. It’s worth sifting through CVs to find someone with the right abilities and mindset, but also a lived experience that can enrich team discussions.

Yet no one will be able to adapt or learn if they are not brought into regular contact with the project’s context, or with direct feedback from potential or actual users. Developers with a more traditional, large-team or corporate background may be uncomfortable thinking about context at first, since in larger and more mature teams writing the code is usually quite distinct from thinking about its use. But in small teams (and, some would say, even in larger ones), everyone should be thinking about users' needs.

In civic tech, that means forging a close link between the community outreach team and the developer team. But that also requires having a community or field team that is deeply embedded with and honest in their understanding of target users.

Invest in building links among the field and developer teams to the extent that information flows naturally among them. Have the developers accompany the field team on occasion; invite the field team in to contribute to a debate on features among the developers. When a developer comes and asks "I'm not sure about X", if it's appropriate, send the developer directly to a field team member; and vice versa, if someone comes back from the field saying, "can we have the app do X?", *and* it obviously makes sense, send the field team member straight to the developer who can implement X. There are fine balances at work here—the CTO, or senior management in general, still need to be in the loop often, and such activities can be pointless if artificial. But it's possible to find the right balance (Figure 2).

At Grassroot, we started off with a relatively senior-heavy structure, with a few too many generalist contractors. Over time we shifted to a structure of 1-2 junior developers at a time, some of them staying on the team for a few years, supplemented by senior specialists brought on for a few months at a time to achieve a specific task. I myself (wearing my CTO hat) reviewed the specialists' code and mentored the junior developers. Our community team had a mandate to spend two-thirds of their time in the field, and, when not in the field, to sit with the developers and relay what they had seen. We supplemented this structure with user experience specialists, who we brought on from time to time to train the community team in observing user behaviour. The structure also allowed us to flexibly incorporate other sources of talent, for example, through the great "computer science for good" programs that some universities have developed (in our case, working with Stanford).

Don’t build it

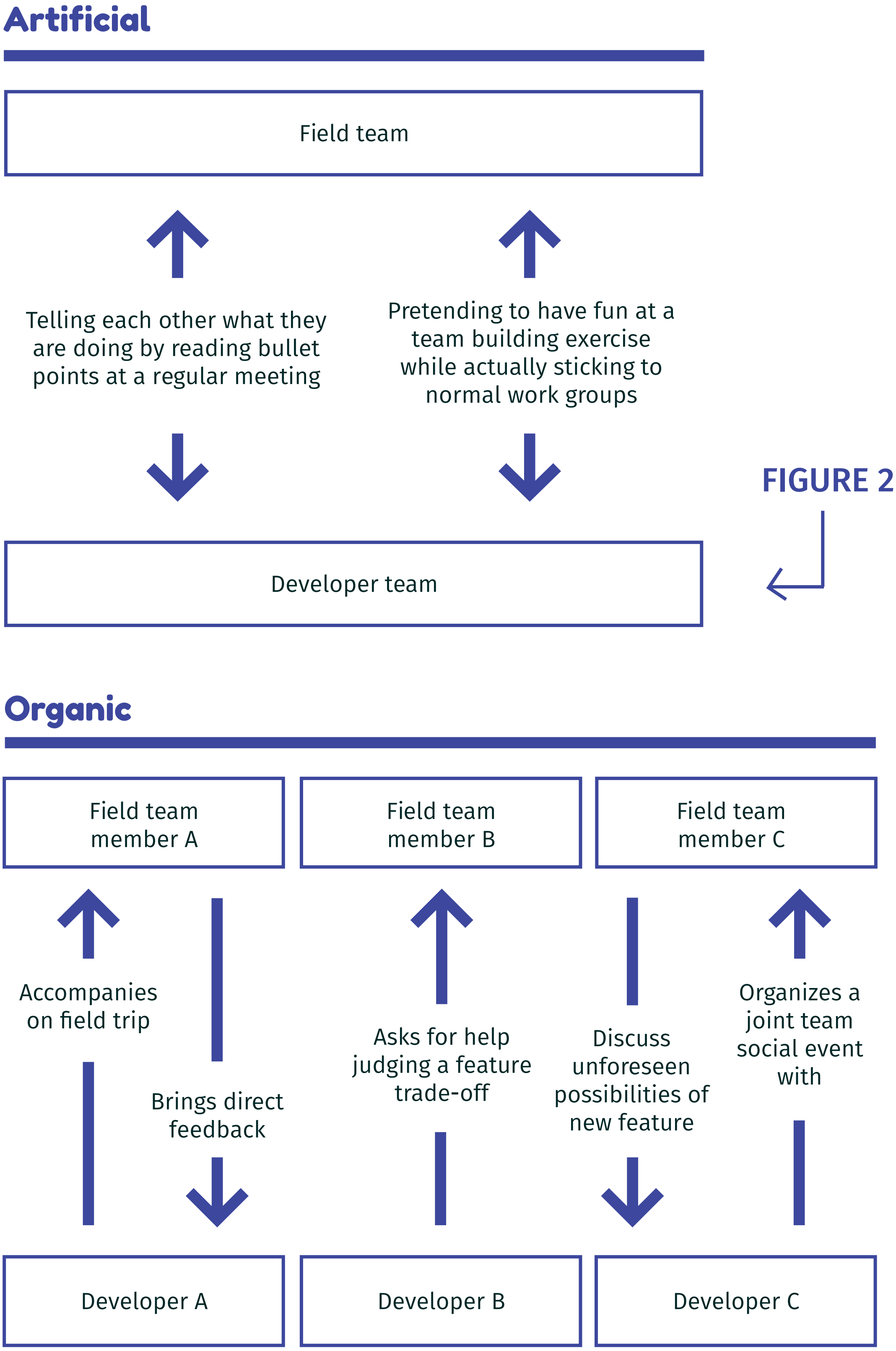


FIGURE 2

Artificial

Field team

Telling each other what they are doing by reading bullet points at a regular meeting

Pretending to have fun at a team building exercise while actually sticking to normal work groups

Developer team

Organic

Field team member A

Field team member B

Field team member C

Accompanies on field trip

Brings direct feedback

Asks for help judging a feature trade-off

Discuss unforeseen possibilities of new feature

Organizes a joint team social event with

Developer A

Developer B

Developer C

HIRING & CONTRACTING

Chief technology officer (CTO)

Term confusion can get in the way when hiring a CTO. In a large company, the CTO role can involve largely overseeing processes and hiring and overseeing middle managers. Good CTOs still get their hands dirty from time to time, but they primarily oversee a number of processes to build. That is not what the role means in a startup, non-profit, or even a government digital service unit. In such teams, the CTO builds, and has to love building things..

An example background for the right person could be a senior developer and “engineering manager” in a strong tech company, who wants out because of the procedural weight of the latter role; or a freelancer with a track record of working on and delivering ambitious, complex projects. The primary criteria are the ability to adapt to new environments and teams, to combine good process discipline with flexibility, and, above all, a love of building things and seeing people use them.

Don’t build it

In finding a CTO, recruiters will likely be helpful. However, it will be important to brief them that you are open to unconventional paths and “jagged resumes”. Once you have candidates, ask them questions like:

- *Is there an example of a project that you successfully killed?*
- *What’s the best team member you’ve ever had, and how would you find more of them?*
- *What’s the worst team member you’ve had, and how would you avoid them?*
- *Have you ever managed to mentor someone from being on the “worst” track to being on the “best” track? Have you come close? What did you do?*
- *How do you decide whether a technology belongs in the stack for a project?*
- *How do you react when a project is (very) late? Or when no one is using it?*

There are no perfect answers to these questions. What you are seeking is someone who has thoughtfully been able to save their team from a colossal mistake, or a lot of wasted effort; who knows how to compose a team, in a positive and negative way; who knows how to guide young team members; who is constantly attuned to new technology but also to the practicalities of getting it into the field; and who will respond well when life happens. If you can find someone at least somewhat familiar with the context in which you operate, weight that highly, but not definitively—an excellent mentor who is highly adaptive and balances rigour and creativity, but who has only limited familiarity with your context, is much more likely to work out well than someone who knows the context deeply but cannot adapt or mentor.

Last, there are a few red flags. Avoid people who are no longer learning. Avoid people who self-promote. Be cautious of someone who says something like, “I don’t see myself writing too much code, because I create more value building and managing the team than writing code”. Of course, if the team grows significantly, writing code will no longer be the best use of the CTO’s time. But the kind of CTO who works well in small teams will likely need to write a lot of code themselves. If they don’t enjoy that activity, if they want to manage more than build, that’s unlikely to end well.

Senior contractors

The best senior contractor I worked with told me, once: “You pay me more not for the code I write, but for the code I don’t write”. That was an exaggeration, but it did encapsulate a core part of the value that a senior engineer brings—the experience to know when a path that seems promising is likely to end up in a tangle, or is not worth it compared to another route, shorter and not exactly what is wanted, but close enough and at a fraction of the time and energy.

you pay me more not for the code I write, but for the code I don’t write

Managing a technology project is a sequence of trade-offs. Very good senior contractors help make the terms of the trade-off better, by expanding the range of possible options and seeing less complex ways of achieving outcomes (or increasing confidence that no less complex path is available). It is difficult to fully know in advance how good a contractor is at this art beforehand, but you can test for it in interviews, asking, for example:

- **Describe a time you succeeded in preventing a team from making a bad choice?**
- **What is the worst project experience you’ve had? What were the worst trade-offs made in it?**
- **How do you personally know when you’re making something too complex? Or when you’re making it too simple?**

One red flag here is if a contractor expresses a dogmatic view against a popular framework or technology. If you have junior team members, and the senior contractors will need to mentor them or do some knowledge transfer, ask how they've handled similar situations in the past.

To source senior contractors, there are some very good platforms that specialize in finding and filtering high-end talent (for example, Toptal). They can charge hefty margins, but for short stints the costs are manageable. The quality filtering they perform, alongside their ability to understand and find niche skills, make them worthwhile. Of course, using them depends on a technical team able to flexibly integrate and use high-quality talent quickly. That will be impossible if your processes are too rigid, the rest of your team too dogmatic, or, of course, if you do not have a CTO overseeing the whole process.

UX/UI designers

Good user experience and interface designers should be able to adapt to any context, by following user behaviour wherever it leads. The challenge with hiring designers is that it is very hard to identify what in a completed project is theirs. By the time a project is done, so much iteration should have happened that the original designs are long buried. A product that looks wildly different at the end than at the beginning could mean a terrible starting point, or it could mean a flexible, creative designer working tightly with a development team to take new directions.

There is probably no real solution to this dilemma. In interviews, probe examples of prior work to understand how a designer learns and adapts. A friend of mine likes to hone into a single wireframe or screen, and probe it in extreme depth. Doing so shows how deeply they thought about the design, how much responsibility they really had for it, and how good they are at engaging in conversations about it. If you are hiring for a significant length of time, it is fair to set a short, contained take-home assignment. It is also usually a good idea to avoid designers who are very rigid in their methods or dogmatic about their craft. If someone cannot explain why personas are sometimes a bad idea, they are a good fit for a large, corporatized organization, but probably not for a small one or for a team operating in ambiguous contexts.

More generally, when you find a good designer, do everything in your power to make them enjoy working with you. Given how hard it is to hire in this field, and given just how enormous a difference it makes to a project, there is almost no resource quite as valuable in building technology as a superb designer who likes to work with you.

Young engineers

The world's universities, bootcamps and online courses produce a steady stream of people every year who have heard that coding is in huge demand, pays well, and provides a rewarding, skill-intensive career. Shortly after completion, they get a dozen job rejections. Maybe they get to a "technical interview", get asked some wholly pointless questions about things they will never need to do, and get rejected. Those who do make it through this absurd funnel eventually find themselves given insignificant tasks in giant and often meaningless projects. Then everyone wonders why the world produces such terrible software, in such vast quantities, every year.

This situation would be comic if it weren't so tragic. It is, though, very useful for the person assembling a development team on a budget. It means that you can hire very talented young developers, almost always from the context where the project operates, and give them a route past the entry-level absurdity by stamping two years of experience on their CV, while offering a modest salary. They will leave after 18-24 months, and will need 6 months to get fully up to speed, but in the period between you will be able to build at a pace and for a budget that people from badly assembled teams will not consider possible.

To do this, you will have to go through very, very many CVs —possibly hundreds. You cannot automate or outsource the process, or you will throw away the best young engineer you could have ever hired because someone who didn't care enough wanted to just get through the pile. So you and your CTO will have to go through them. It may be the highest value work you can do. If you don't have the time and energy to do it—well, just don't build the project.

The criteria in both the CVs and the interviews that I have found have worked best are:

- **Hunger to learn.** *Some traditional questions—about side projects, or subjects learned outside of studies/work, or failures—are genuinely useful. It's also helpful to ask about a team decision made on a prior project, and why it was made—someone who likes to learn quickly on the job will have sought to understand the decision, and its rationale, whereas someone who doesn't will have just noted it and moved along.*

- **Commitment to quality.** *Coding often involves short-cuts and practical trade-offs. The great temptation is to take an easy way out and justify it as such a trade-off, without having properly explored the options. It is often very hard to tell from the outside which is the case, so the great temptation facing young developers is to use practicality as an excuse. You can test that a little with code tests, but not on a whiteboard. Here, take-home coding tests are useful, but only if discussed afterwards. Otherwise, questions about managing trade-offs in their prior life, and framing hypotheticals about responding to imagined (but concrete and plausible) scenarios can be helpful.*

- **Basic technical ability.** *This is obvious, but (even with supposed experience) it is sometimes remarkable how many interviewees will not be able to code. So, some simple coding questions and a few technical questions are handy.*

The single best hire I ever made was someone who had first tried to be a chicken farmer. The South African government gave him a grant to raise poultry, but forgot to tell him anything about disease management. While his chickens were dying, he taught himself to code. By the time I interviewed him, he had built a couple of mini-apps for his friends, and could talk coherently about the choices he'd made building them and how he'd build differently in the future. I found him through a similarly young, talented and unorthodox recruiter, who I'd told to find unusual candidates.

DEFINITIONS

Stack Overflow:
A site where developers ask and answer questions. Arguably, after Wikipedia, the single most valuable public good on the internet.

TIMELINES & BUDGETS

Set a budget that gets you off to a quick start, but allows you to keep iterating over time

It is almost always the case that a simple version of the technology you're trying to build could get out the door and in the hands of some real people (known as "shipping") quicker than you think. It is also almost always the case that bringing the technology to a state where it's heavily used and has few bugs ("maturity") will take much longer than you or your timeline will estimate. For that reason, set a budget that gets you off to a quick start, but allows you to keep iterating over time, while retaining knowledge within the team.

Most things go wrong in software. Nobody wants to use what you've built; or the product fails in strange and unanticipated ways once it's in the real world, because software and life are both complicated. Or the product fails because it's badly built. There will always, always be some bugs, but you can reduce them by an order of magnitude by (1) hiring a good CTO, and (2) using a practice known as "DevOps", and (though this is more controversial if it's not already a team habit) "test-driven development".

But even technology built to the highest standards will encounter unanticipated bugs once it hits the real world. This can extend the real timeline for building technology long past its shipping date. Software consists of many small pieces connected together, and is deployed in a world full of complexity. Users behave in ways that you don't understand at first, and could not have anticipated. Between unanticipated behaviour and complex interactions between components, it can be very, very difficult to identify why something is breaking.

Don’t build it

In sum, shipping timelines can always get shorter, and maturity timelines will always stretch. In specifics? A rule for this sort of guide is never give specifics that people can hang around your neck. A rule for software development is “it takes as long as it takes”. I’m going to break both rules, and say that for a good team of 3-4, following good practices, you should be able to ship a moderately complex product in about 3 months, and get it to some form of stability and maturity within 12 months. After that, of course, you start again—if you have users. If you don’t, stop at 6 months, or better yet, 4 months, or best yet—**don’t build it.**

A rule for this sort of guide is never give specifics that people can hang around your neck. A rule for software development is “it takes as long as it takes”. I’m going to break both rules

What those timelines mean for budgets varies with the team structure. Over four years, Grassroot spent a total of around USD $200k on development. This covered the build out of the core platform, its extension to running large-scale national campaigns, experiments with WhatsApp bots and machine learning systems, and several Android apps. That is slightly distorted by the organization having a combined Executive Director/ CTO (me) at a much lower than market rate, which I don’t recommend. Nonetheless, I have known some organizations to quote at multiples of that amount.

On a unit basis, in developed markets a good engineering manager (to be CTO, see below) would be about $10k per month (more in big US cities), and a good, inexperienced junior, roughly $4k or so (again, more in New York/San Francisco). In developing markets, if not wholly distorted by “impact fund” or VC money, those numbers would be $5k and $1k-$2k. Good quality senior contractors from Eastern Europe, for example, come at around $80-100/hour if sourced through a good matching platform, or $40-60/hour if done directly (it is almost always worth it using

DEFINITIONS

Test:
A piece of code that asks your software to perform certain functions, and then checks whether they’re done correctly.

a platform, unless you have worked with a contractor already or have a strong referral). Overall, unless you have technical expertise in-house, it should cost between $50k-$100k (excluding CTO cost, as that will spread across many projects) to build and get to maturity a moderately complex project. Much less and you are kidding yourself; much more and someone is probably pocketing some very healthy margins.

Two last points here, one for the start of a project and the other near maturity. First, at the start, investment in what's known as "tooling" usually reaps rewards down the line—that means not just getting the "DevOps" pipeline in place (see definitions), but also some basic documentation, coding style conventions, scripts for getting new developers up and running, and automating things like code quality reporting and vulnerability scanning. Upfront investment in such tools will all increase the speed at which the team can work in the future, not just in routine matters but also when there's a fire to put out. The time pays itself back, often many times over. There is a limit though—if someone starts suggesting something called "Kubernetes", tell them to stop.

Near maturity, perhaps the oldest law in software development is Brooks' Law: "Adding manpower to a late software project makes it later". More generally, if the project seems on the verge of becoming a catastrophe, adding budget and people may be the worst thing you can do. Bringing new people on board, working out what to do with the new funds, establishing new reporting, all of that will take the scarcest resource—CTO and senior team time—and deflect it from the most important task, which is diagnosing the core problem and fixing it. When the project has turned the corner, and has changed from chronically late and chronically broken to proceeding quickly and in a sound way, albeit behind schedule, then additional resources can fuel an acceleration. Turn the corner, then add gas; if you add gas before the corner, you just hit the wall harder.

DEFINITIONS

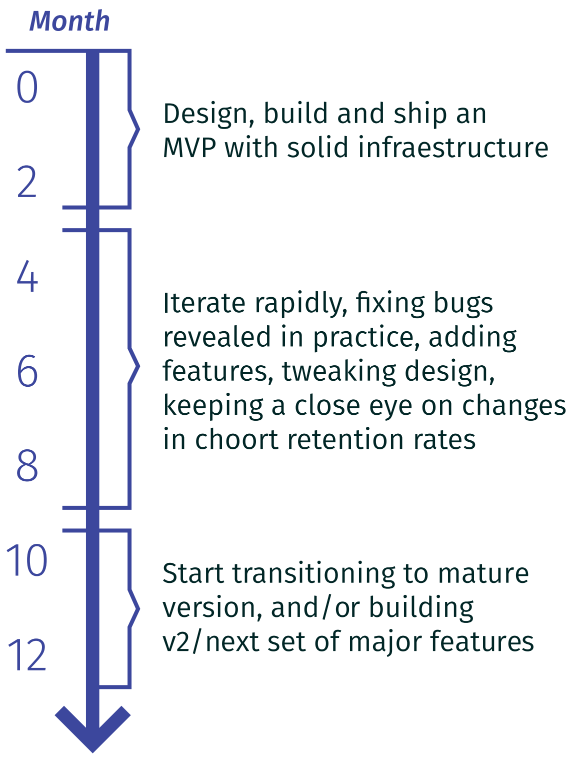
Test-driven development (TDD):
Writing the tests that software must pass to be accepted, before writing the software itself. TDD saves a lot of time in the long run and makes it immediately clear when a key function has changed, at the cost of additional time upfront.

DevOps:
A set of practices to enable new code to be continuously integrated into existing code and deployed swiftly and reliably. The specific tools for integrating code and deploying it are called "continuous integration/continuous deployment" (CI/CD).

Turn the corner, then add gas; if you add gas before the corner, you just hit the wall harder

Don’t build it

Good timeline

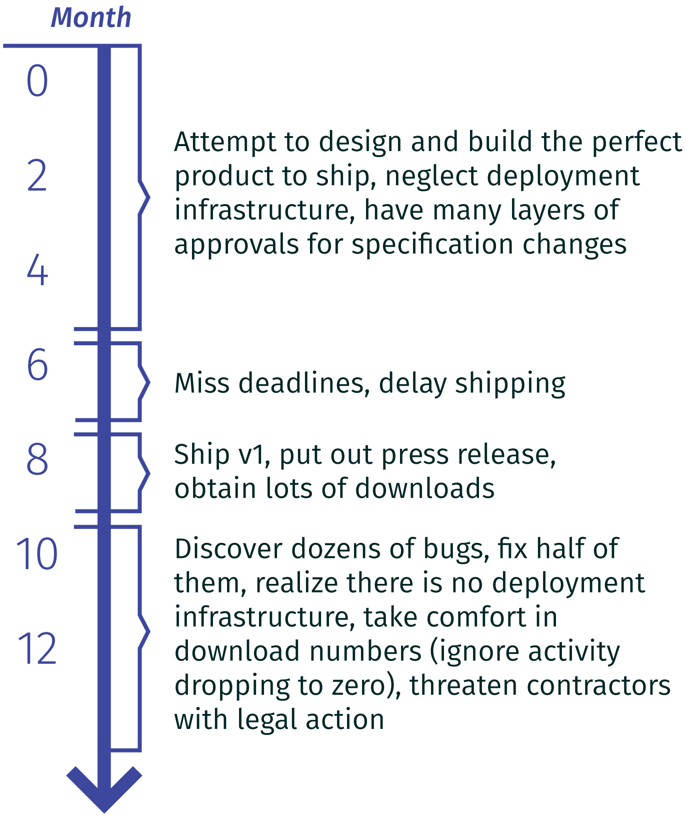


Design, build and ship an MVP with solid infraestructure

Iterate rapidly, fixing bugs revealed in practice, adding features, tweaking design, keeping a close eye on changes in choort retention rates

Start transitioning to mature version, and/or building v2/next set of major features

Not great timeline (planned or actual)



Attempt to design and build the perfect product to ship, neglect deployment infrastructure, have many layers of approvals for specification changes

Miss deadlines, delay shipping

Ship v1, put out press release, obtain lots of downloads

Discover dozens of bugs, fix half of them, realize there is no deployment infrastructure, take comfort in download numbers (ignore activity dropping to zero), threaten contractors with legal action

DEFINITIONS

Script:
A file specifying a sequence of commands or operations to perform, for example to create a mock replica on a developer’s machine of the application being built. Often used to automate certain tasks, like setting up a new developer, archiving log files, or checking for new versions of code.

Tooling:
A program that developers use to create, document, debug, maintain, deploy and otherwise support other programs and applications. For example, a source code editor, or a tool for detecting and flagging potential code quality issues.

Kubernetes:
A complex system for managing the deployment of very large scale applications (millions of active users)

USERS VS VANITY METRICS

Some of the lore in Silicon Valley is useful. The most useful is the idea of “product market fit”, and its antithesis, the “vanity metric”. Having product market fit means that you have built something that people find valuable to use regularly and will tell other people about. In other words, do people actually want to use the thing that you are building? What is important here is that your opinion of whether people want to use your product, your partners’ view, even your focus group’s view, are all inadmissible as evidence. Evidence is data from within the app, collected over a meaningful number of users, over a meaningful amount of time.

Evidence is data from within the app, collected over a meaningful number of users, over a meaningful amount of time

Equally important is that “people using the product” does not mean “total downloads”, or “landing pages”, or even “accounts created”. “Using” means either taking the core action regularly and repeatedly, or —even better— paying (even a micropayment) for the technology. Those other measurements are “vanity metrics”, meaning a metric that makes you feel good or presents your technology in a good light, but does not reflect reality. The most common example is to measure total user numbers, which will always go up except in a complete catastrophe.

Don't build it

Vanity metrics are especially dangerous for non-profit and government projects, because free press makes them easy to pump up, and the lack of profit and loss means that such metrics can short-circuit learning indefinitely. Moreover, donors and senior officials will create strong if tacit incentives to use the metrics that make targets easiest to beat—and some nice free press can usually follow. But for a project itself, nothing is more important than rapid early learning, and vanity metrics short-circuit such learning. If possible, don't send that press release, until confident that you are near product-market fit, and, even if you need the vanity metrics to appease other stakeholders, ban them from internal team discussions. Vanity metrics are extremely seductive, especially in difficult periods. One team member just needs to say, "We shouldn't be so down, total users are still really high!", and the rigorous honesty on which progress depends will come to a halt.

Nothing is more important than rapid early learning, and vanity metrics short-circuit such learning

Instead, build a funnel, from downloads through sign up through to regular use or subscription, and watch how new users progress through it. To do that, you will need fine-grained data from your app on how people are using it, obviously anonymized and aggregated. This is called "instrumentation", and it will allow you to watch how average behaviour among groups of users (cohorts) changes. Tools like Amplitude make it simple to set this up—you do not need to build it yourself—and allow non-technical team members to monitor how use is evolving. Practically, set up a chart showing what percentage of people drop off from use at what stage, and open that chart every single morning. If it's not getting better, you need to fix things, no matter what the latest social media blast did to the vanity metrics. If cohort retention is not going up, your project is not fitting its intended use.

If that is the case, what do you do? A set of techniques in user experience and user interface design (UX/UI) can help you understand where products are falling short (just asking people is not a good idea—they will be too nice, and may not remember). If you don't have a UX/UI specialist, or are

DEFINITIONS

Funnel:
A sequence of steps that a potential user goes through on the way to becoming a regular user, from landing on your website or app listing (top of the funnel) to being fully set up and regularly engaged (bottom of the funnel).

very short on budget, read up on the techniques (I hope my UX/UI friends will forgive me) unless you are in really deep trouble, in which case find the budget or appeal for a UX/UI volunteer. If you invested in good tooling and instrumentation upfront, it should be easy to deploy small tweaks very quickly, and observe the results in real time. You will still need some patience and flexibility—but not too much.

Finding the right balance between perseverance and stupidity is a hard art. It is normal to see conversion rates go down as often as they go up in the first few months after launch. But if you aren't constantly getting closer to product-market-fit after 3 months, go back to the drawing board. If you aren't very close by 9 months after your first launch, it's likely time to pull the plug and free up the resources for something else. Another red flag is if, a few months after launch, the week-by-week cohort retention rate goes down for a long stretch of time, and repeated (3+) attempts to fix the problem have all failed to produce any effect.

Once your cohorts are consistently getting better and a healthy proportion of your users are converting into regular or paying users, the next metric to watch is a "net promoter score" (NPS). That measures how likely people are to recommend your product to others. By far the best way to measure that is by embedding a referral code system in the product and monitoring its use, but occasional and/or automated surveys provide a proxy.

The single worst measure of how you are doing is press coverage. Be immensely suspicious of any project with a lot of press coverage. It hints at a team trying to pump vanity metrics instead of making their product work.

DEFINITIONS

Cohorts:
A group of users that started using the product at roughly the same time. This is used to analyze if product-market fit is getting better or worse.

Cohort retention:
What proportion of a cohort is still using the product at a later date.

Conversion rates:
What percentage of users at a certain step in the funnel move onto the next step, e.g., what percentage of people who download an app or visit the homepage then create an account.

Instrumentation:
Tools embedded in a software product to measure in real-time how it is being used and when and where it is failing.

UX/UI:
The profession of designing user experiences (informally, the "flow" of a product) and then turning that experience into a user interface (the specific design).

TECHNOLOGY SELECTION

I get asked quite often by non-technical friends embarking on projects, or just wanting to learn to code themselves, "Is language X a good idea?" To a large extent, the CTO you've hired should make these decisions. But just as, if you run a health organization and aren't a doctor, it's good to know enough about health to follow high-level discussions about medical techniques, it is useful to know a little about technology selection.

The little that's most useful to know is that, within some broad parameters, specifics don't matter too much. Do not spend too much time on what programming language to use for a project, what exact database, what to use for the mobile app or the web page. Instead, use a couple of broad principles, and try to avoid anyone trying too hard to convince you that their technology preference is a perfect solution —either they are trying to sell you something, or they have insufficient experience. The basic principles are:

(1) **Stick to open source.** *For non-profits, this choice is largely enforced by budgets. For government projects, anecdotal evidence suggests that giant enterprise vendors are trying to sell aggressively in the developing world to make up for being increasingly replaced by open source in the developed world. Do not listen to them. Just use Postgres (or MySQL or MongoDB, if your team has more experience with them).*

(2) **Pick a popular language.** *As a rough rule of thumb, use Python if machine learning will be important, Java or C++ if you anticipate tens of millions of users, and the Javascript family (NodeJS/Typescript) otherwise. But as long as you avoid PHP (I know I said not to be dogmatic, but PHP is actually terrible), it doesn't really matter what you use. Picking one of the more popular languages just means you have a larger pool to hire or contract from, more frameworks to use, and general problems in underlying packages are likely to be fixed more quickly.*

As described in various sections above, you will want to require test-driven development. You will also want to make sure everything is instrumented, i.e., you are able to monitor real-time usage statistics and detect crashes and failures, and, if you are using contractors, that documentation is up to snuff.

Importantly, the choice of technology has almost no impact on security. Good security requires process discipline and a healthy dose of paranoia. If someone believes a technology choice will provide security, they will be hacked, unless the product is not used, in which case no one will care enough to hack them. Security starts with design. A practice known as "security by design" will design the overall system, so far as practical, to prevent and contain breaches. For example, the tables containing any personally identifiable data should be kept on a separate server to the one used for processing normal user activity, and the functions that handle such activity should never have permission to access the other server. That practice intersects with a general principle of good software engineering — known as "separation of concerns". Similarly, make sure that you have DevOps in place (see Timelines & Budgets above), and make sure that security scans are incorporated in the DevOps process, so that code is automatically scanned for vulnerabilities. In general, clean code is good code, and good code is (relatively) secure code; good teams produce good code, and incorporate secure practices.

And yet, you should assume that, at some point, you will be hacked. On a project with a large budget, you could commission penetration tests. Whether you can or can't afford that, have regular sessions with the development team to ask: If someone hacks us tonight, what's the most data they can get, depending on how they enter? How would we know? What has changed that might open up new vulnerabilities? Your CTO should lead these discussions. But non-technical leadership should participate too —if not in the very granular technical discussions, at least at the level of system design and understanding the potential and means of containing breaches.

Finally, it should be noted that if you have decided to contract out the entire development, even though it is a terrible strategy, you can insert all of the above as requirements in the scope of work. Your CTO will have to check that the developer is keeping to them. If you have contracted out the entire development and have not hired a CTO, put the requirements in and hope that you get lucky and the contractor pays attention.

DEFINITIONS

API:
Application programming interface. A way for one application (e.g., the app on the phone) to communicate with another (e.g., the cloud-based function performing processing).

Framework:
A base of pre-built code packages that allow developers to quickly create new applications. For example, React is a framework for web development.

Penetration test:
When a hired or friendly hacker (sometimes called “white hat”) tries to penetrate your system to reveal vulnerabilities so that they can be fixed.

CONCLUSION:

↳ *REALITY*

Despite the best practices and best intentions, we live in the real world. It is more or less inevitable that the forces of press attention, donor preferences, team preferences, and institutional inertia result in a decision to build some technology for bad reasons. You may be pressed to “do something digital”, to undertake “digital transformation”, or the like. Vanity metrics will proliferate. Someone will suggest press coverage. Before you know it, you’ll be connecting unrepresentative organizations to unaccountable institutions via unusable chatbots, justifying it by a few thousand people who tried it out and never used it again. Or, you may find a project genuinely worth doing, but will be unable to hire a good full-time technology leader, and so will end up with a part-time, mostly uninterested contractor and a few junior engineers lacking mentorship, and a project that ships late and then is never fixed.

Right now, many donor grants and expectations aren’t structured for successful tech products. At Grassroot, I had to spend a lot of time working out how to make a flexible, adaptive team structure work with donor grants that made a distinction between “full-time” and “part-time” salaries, even when funding “core” activities. The idea that a junior developer might leave, a contractor comes in briefly, a wholly new user issue is discovered, a different contractor is brought in, and so on —this is not a model that’s familiar to most donors. Similarly, when we realized that even our deeper metrics of user activity and engagement —which were going off the charts— were not leading to impact, some donors still hoped to run up vanity numbers by expanding into new geographies. When we did find projects that had strong results but weak vanity metrics, we could not find takers. And many times when I was asked for advice by friends or peers, and I suggested not building it, whatever “it” was would be built anyway, obtain a press release, and languish.

Don’t build it

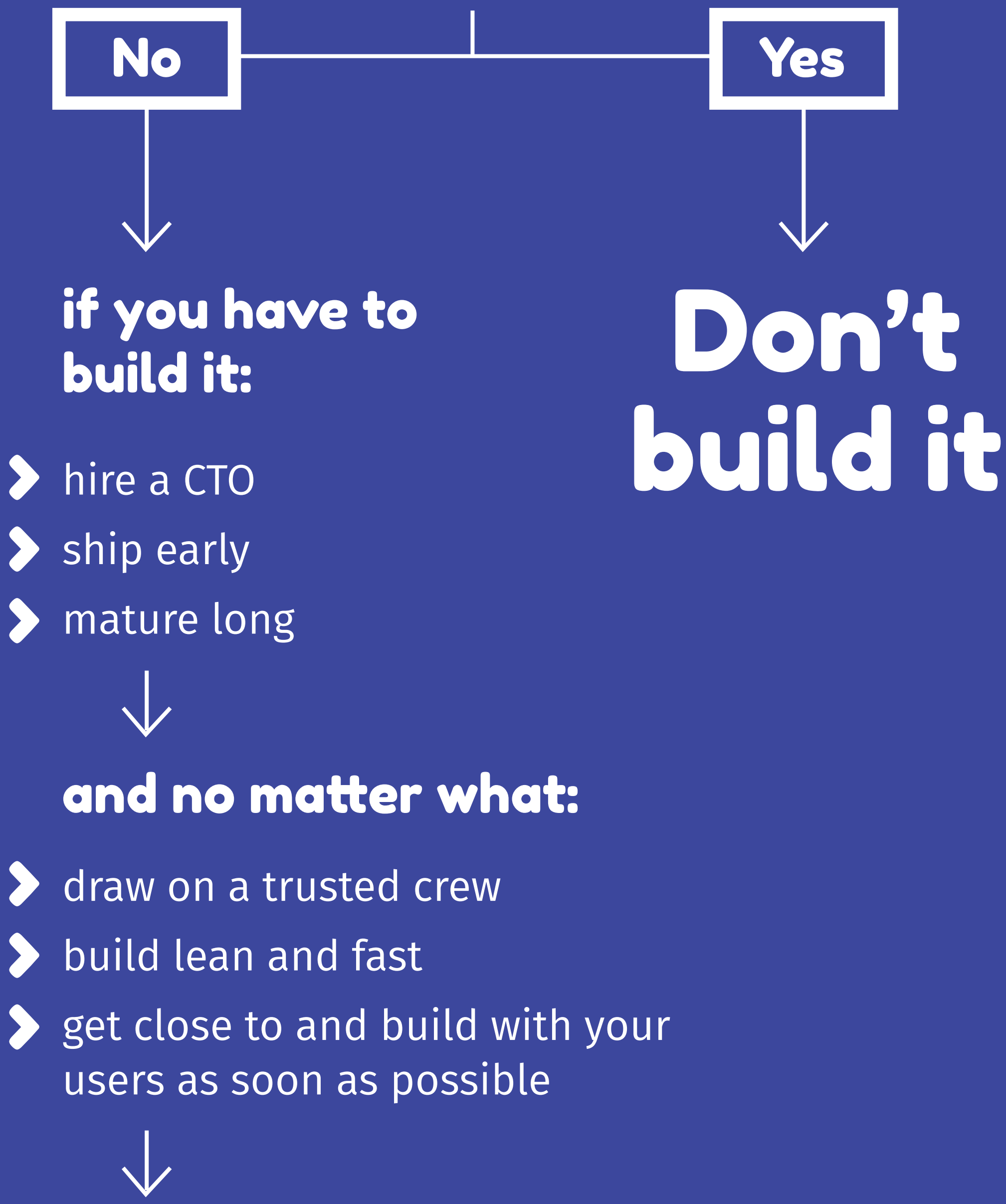
Get close to your users and to do so fast. Getting close to users means having a dedicated community engagement team, being present in the field as much as possible,

There are only a few routes out. Hopefully, the arguments in this guide will help at the margin—to avoid bad projects, structure the team right, ship and learn quicker, and mature longer. If so, the guide will have served some purpose. Otherwise, there are a final couple of tips for reality-adjusted damage limitation. The first is, so far as possible, cultivate a group of good, trusted engineers and designers and team leads, and rely on them whenever you can. If you have to do a project, and can’t structure it right, but you have good and trusted people that are not available right away, then delay until they are. Connect the field team and development continuously. Request reporting extensions because “We have to use technology X, and the only person who can do this project with technology X is not available yet”, or just build the time into the budget. Then, whenever a project comes up, spend a bunch of time with those trusted collaborators developing a strategy to obtain the maximum of strategic flexibility and the ability to execute the project well, within the constraints that reality has imposed. If you’re conscious and cautious about it, and have built up some experience, it’s sometimes remarkable just how much space you can create.

Finally, the most important tool is to get close to your users and to do so fast. Getting close to users means having a dedicated community engagement team, being present in the field as much as possible, and feeding direct observation into your development process all the time. The quicker you do so, the quicker you will be able to iterate to something people want to use, and the quicker you will be able to adjust a bad idea towards at least being partially useful.

In all:

Can you avoid building it?



No

if you have to build it:

- hire a CTO
- ship early
- mature long

and no matter what:

- draw on a trusted crew
- build lean and fast
- get close to and build with your users as soon as possible

Yes

Don’t build it

beyond that:

as the Bhagavad Gita says,

> **“you have a right to the work, not to the fruit”**

Don’t build it

A GUIDE FOR PRACTITIONERS IN CIVIC TECH / TECH FOR DEVELOPMENT

GRASSROOT

MITGOV/LAB