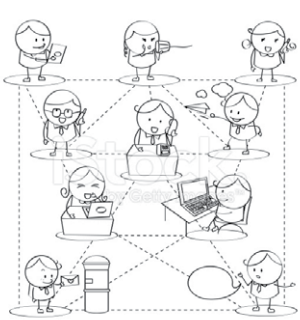
Lean Software Development– Building and Shipping Two Versions

CATERING TO DEVELOPERS' STRENGTHS WHILE STILL MEETING TEAM OBJECTIVES

KATE MATSUDAIRA

Once upon a time (and isn't that how all good stories start?).... I was managing a software team and we were working on several initiatives. Projects were assigned based on who was available, their skillsets, and their development goals. This resulted in two developers, let's call them Mary and Melissa, being assigned to the same project.

Mary and Melissa had been working together for a few weeks when I started hearing complaints in my one-on-ones with each of them about the other. Mary was complaining that Melissa was taking too long to do her part, and spending time on unit tests that didn't make sense because things were in flux with the project. Meanwhile, Melissa was complaining that Mary wrote sloppy code and didn't write enough tests (she even showed me comments in the code like "hack:...." and "to do: someone should add more error handling here"). Each of them had valid points and feedback for the other.



I spent the next few weeks coaching each of them on how to improve and address the other's concerns. With Mary I was focused on pushing her to write more tests and not

just throw things together, and with Melissa I was focused on having her prototype something quickly and then add the polish after it was working. Both of them tried really hard, and both were miserable. It just wasn't what they were meant to do.

These two women were just not meant to collaborate. This got me thinking: how could we change the process so they both could write code the way they liked to create (and that catered to their strengths) and we could still meet our team objectives?

From this was born the v1/v2 development process.

You see, certain developers love building the first versions or prototyping – they are the ones who love hacking things together to get something working quickly. They love (and are best at) building version 1 of a product. The other type of developers love building the second version. They see their code as a craft and write unit tests for everything. "Test coverage" and "beautiful code" are phrases they use a lot.

And of course this definition isn't black and white–there are people who fall on different sides of the line at different times, so it is more like a spectrum.

Typically, the v1 person doesn't like working with the v2 person, and vice versa – not for personal reasons, but because of the ways they think and create software. At the heart of it, the fundamental thing they enjoy about what they do is different.

By changing the way we thought about software, we were able to address this concern and actually make the team more agile and solve some business problems along the way.

There is a popular movement called Lean Startup that advocates fast iteration and data collection to home in on the right product and requirements as quickly as possible.

THE V1/V2 PROCESS

When it comes to software development there are many ways to build and create products.

In my experience it is difficult to build the right product the first time. By "right" I don't just mean something customers will want and use, and that hopefully will generate revenue, but also the right technical solution. How customers will use products can be hard to predict. For example, before you ship the first version of a product it can be difficult to answer questions like:

- How quickly will the data grow?
- How fast will writes need to be?
- Will it be feasible to write directly to the data store or do you need a queue to manage writes?
- What will the throughput be, and is it enough?
- Will the system scale with usage?

Even if you could reasonably answer these questions, or if you built a system that took all these things into account such that you could "just add hardware," the effort would probably be larger than if you had simply pushed to ship a first version quickly. And it is possible (if not always probable) that such a system would be over-engineered (since it was designed to solve problems or scale in a way that may not be useful), or even have other unforeseen issues.While there are certainly pros to doing it right the first time, in practice it is very hard to get it right, and building the end-all system will take longer, delaying the answers to all of the questions indicating you are moving in the right direction.

There is a popular movement called Lean Startup that

advocates fast iteration and data collection to home in on the right product and requirements as quickly as possible. It's all about figuring out what customers will use and then building those things. These ideas can also be applied to the technical ways we build products.

BUILD V1 FAST, THEN BUILD V2 RIGHT

Similar to creating a minimum viable product (MVP), build the minimum viable technical implementation that meets the business goals (meaning the product can meet current usage requirements and performance).

Most engineering teams have been taught to think about software design and architecture from the beginning and to build something that can scale. However, if you don't have any customers yet, scaling really isn't your challenge. So in some ways, designing for scale when you don't have to scale yet is solving the wrong problem.

Of course I am not advocating doing something stupid, or writing bad code, but don't over-engineer or solve problems before you have them. Here are some examples:

- forgoing the deployment of a multi-node Cassandra cluster and storing the initial data in a single database (with backup) that is fast and easy to make arbitrary queries against
- faster UI development using standard forms and charts versus interesting shapes or fancy custom graphs
- cutting corners on unit tests (because the units are in flux) or comments and documentation

All these shortcuts will help you get something out faster.

They may not be best practices but they do ensure you are building the right thing.

In many ways, this process is about looking at the problems differently and, instead of solving the "big" problem, focusing on speed and efficiency – getting the highest ROI (return on investment) for your development resources.

So you modify your development process to ship the first version quickly, and then once v1 is out in the wild, plan to start on v2 right afterwards.

Then your v2 can address all the problems with the first version of the product. You can improve usability, cut or add features based on customer usage and feedback, pick the right technology and libraries to power the product, and spend time writing acceptance tests and unit tests that are unlikely to change. From a business perspective, you get the product out there faster, so it is easy to see if this product is going to drive revenue and success.

Why This Process Makes Sense

It reduces the risk of over-engineering. As noted above, getting something out quicker and understanding how it is used (or how successful it is in the market) will help confirm that you are building the right product and solving the right technical problems for your business. This allows you to mitigate the risk of building or scaling out systems that may not address the production issues.

If you build the wrong thing you can correct it. Since the product is truly the minimum viable product, you reduce the risk of wasting resources creating the wrong product. Furthermore, you will get feedback earlier in the release

cycle and can pivot or make changes faster (since it is always harder to change a system with a plethora of moving parts).

Chances are you will get to market faster. Building something quickly will help you launch sooner, which will help you test and determine the viability of your product faster than waiting for the bigger launch. This means you won't spend time building software your customers won't use or you can't sell.

Staffing and personnel happiness. This is one of the biggest upsides to this method. Software engineers get to do what they love, in a way that caters to their strengths. Moreover, they aren't forced to work with people who don't share their approach, and they don't have to work in a code base that they can't sculpt into their desired piece of art.

So what are the downsides?

Total time to build is more than if you just got it right the first time. In this model it is likely that v2 development will be finished later and the total development time will be greater than if you built the product once. If you build the v1 using one technology (say, a MySQL database that won't scale the way you need it to), and then the v2 needs a different one (say, a proper key value store for faster queries), the team will spend time shipping and building expertise on one database, only to have to learn and manage another.

Operations can be a headache. Most seasoned engineers will tell you that building something really fast can result in operations nightmares if the usage or data grows in a way that wasn't considered. No team likes being operational and fighting fires. Of course, if this is a problem it probably

means that your product has achieved some adoption and success, so at least you are solving something that produces real value. However, even for the v1 it is still worthwhile to identify risks and potential bottlenecks, so in the event there are problems the team will have some ideas for solutions.

SO YOU WANT TO TRY IT AT HOME? ER...WORK?

Like any software process or methodology it is important to assess if this makes sense for your business, product, and company – it certainly isn't a one-size-fits-all. However, it definitely has its merits, so feel free to steal the pieces you like.

If you do embark on this journey, here are some pointers from my experience:

Get management buy-in. This is really critical, because a key to this approach being successful is always building a v1 and a v2. If you don't have buy-in and your boss is content to keep and operate the v1, your team is going to hate you. Seriously. No one wants to get stuck supporting crufty v1 software indefinitely. So make sure everyone knows and understands the plan ahead of time.

No one wants to get stuck supporting crufty v1 software indefinitely.

Measure everything. In order to build the right v2 you need to know what to build. Where are your bottlenecks? How is the data growing? Without this data it is very hard to realize many of the advantages of this model. Make sure you think about what you need to measure, and how you will measure it, before the product ships.

The v2 will take much longer than the v1. I have personally seen this process evolve across four projects (no, it's not a large sample set), and in each case the v2 took more than twice as long as the v1 to create and launch. Of course, every

v2 I have launched had a lot of extra features, and they didn't always leverage a lot of the v1 code. Some people assume the v2 will be faster, since "you have already built it once before," but I haven't found this to be the case.

Have a tight feedback loop with your customers. Make sure it is easy to get insight into what people use, what they like, and what they don't like – that way you can build a great v2.

Make sure v1s and v2s are celebrated equally. Sometimes teams or companies celebrate the first launch of a feature in a much grander way than the next version. But in this case the next version is what you need to grow your business. It is just as important as v1, if not more so. Make sure you value each of these equally, or people will gravitate to the project with the most fanfare, not necessarily the one suited to their strengths and talents.

Be open to people being good at both v1s and v2s. Sometimes I think I have someone pegged and then they prove me wrong. Be open to the fact that there aren't two types of people, and some people are just good at everything. (I just wish I was one of those people!)

I hope some of these ideas will prove useful for you and your team. If anything, let this anecdote inspire you to question the way you are doing things, and look critically at innovative ways to improve how you build software.

If you liked this or have other, better ideas, definitely post them in the comments – I am always looking for ways and ideas to make us more efficient and effective as an engineering organization.

LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Kate Matsudaira *is an experienced technology leader. She worked in big companies like Microsoft and Amazon and three successful startups (Decide acquired by eBay, Moz, and Delve Networks acquired by Limelight), before starting her own company Popforms (https://popforms.com/), which was acquired by Safari Books. Having spent her early career as a software engineer, she is deeply technical and has done leading work on distributed systems, cloud computing, and mobile. However, she has shown herself as more than just a technology leader by managing entire product teams, research scientists, and building her own profitable business. She is a published author, keynote speaker, and has been honored with awards like Seattle's Top 40 under 40. She sits on the board of* ACM Queue, *and maintains a personal blog at katemats.com.*