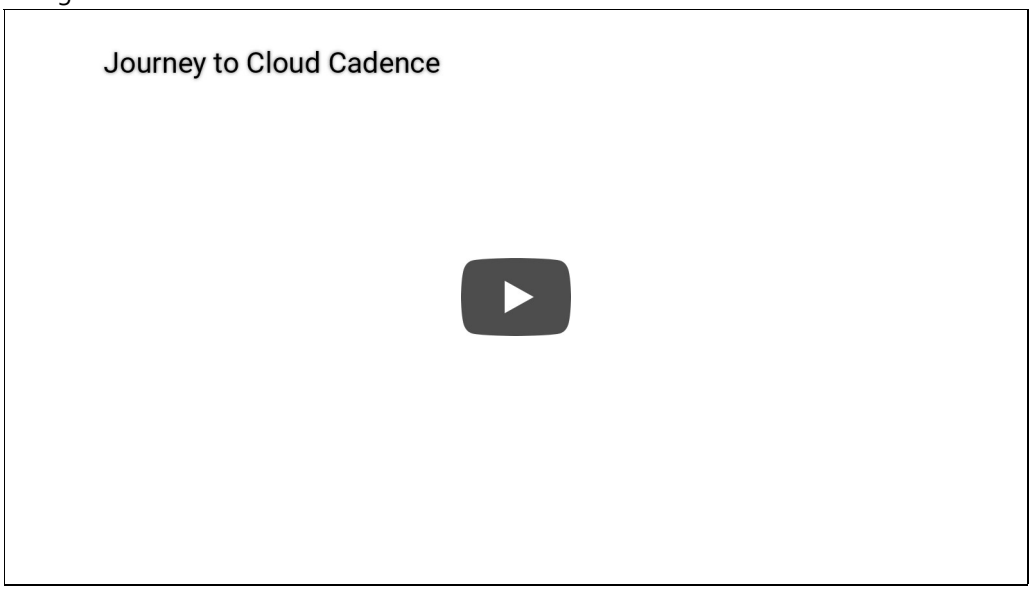
Moving to Cloud Cadence

11/08/2017 • 12 minutes to read

In this article

By: Lori Lamkin

At VSTS in Microsoft, we've been on a DevOps journey. We're not perfect, we didn't do this in six months and we're not done. We had major milestones along the way and the pattern that we followed is very similar to the pattern I see customers following. They start with, "Gosh, I want to reduce my cycle times. I want to do this DevOps thing. I want to move to that cloud cadence. My business isn't keeping up." Yes, that's why we changed too.



First Change the Cadence

What do you do first? **Change your schedule**. I want to ship faster. That is the first step on the journey. To me, the big ah-ha moments for this was when we tried at first to change the schedule and we created a stabilization period because we were scared that we weren't going to have a stable product. That delivered the wrong behavior. Some teams did the right thing and burned down their debt, other teams kept building their debt because of course engineers want to ship features so they're going to cram as many in as they can. Then team A had to fix team B's bugs. Yeah that was rewarding the wrong behavior. The key to us was just jumping off the cliff, getting rid of the stabilization milestone. Then all the sudden the system rewards the right behavior, because if you build up debt, then you have to spend sprints burning it down, it's just you. You don't get to ship your features until you do it. That was kind of the key ah-ha thing.

Team Autonomy and Org Alignment

The next thing is **letting go**. As a management team there's a lot of security in planning up front, and feeling like oh we're going to get these dates, but it's just false security. You've got to let your teams be able to run with their own backlogs, their own plans, and somehow find a way to align them.

I often talk to executives especially it's about how we run our feature chats and our sprint demos. Those come across as big ah-ha things because they ask, "Can you just give me the KPI's so you can measure productivity? Can you just give me the KPI's so I know which teams are doing well or which ones need help or my features are on track?" Again, you've got to be in the product. You've got to be talking to the people. The tools facilitate that, but conversation is way better in that transparent rhythm.

Ship Continuously

The other is to **always deliver features**. That schedule, it's not like, yeah we do a three sprint plan and we try to stick to it. We're listening and we're learning.

Work in Master

We used to have engineers working in separate branches. The merge debt is hidden until you try to do that integration, and the more teams that you have, the bigger that integration becomes. How can you get that integration to happen in small chunks and faster, more continuously? The key is, **work in master**. One of the reasons why we moved to Git is the lightweight branching. The really big benefit to our internal engineering was getting rid of that deep branch hierarchy and the waste it introduced.

Walk the Walk

Because we **use the tools that we build**, our one investment yields benefit both in our productivity and improving our product. For example, it's really important that we have a release management system that we ship to everybody else and that we use ourselves, instead of something secondary that then siphons off a bunch of velocity from the team.

Continuous Deployment

The less frequently you deploy, the harder it is to deploy. The more you have time between deployments, the more stuff piles up. Suddenly, the code isn't fresh in anyone's mind, so you get deployment debt. The more

you can **work in the smaller chunks**, the easier the actual deployment will be. I say to people, on our journey, when we avoided deploying because it was so hard and that just made it harder. Yeah. Reserving that, it seems like it makes sense but it's a little counterintuitive when you're in the middle. Deploying more frequently made us prioritize making the tools we use to deploy more solid.

Instrument Everything

Of course, debugging needs to change. You can't attach a debugger on a production machine. That means we're flying blind. Then our practice became that we needed to **instrument everything.**

Now Test Continuously

At this point, we were still moving slowly, but why? "I wrote all my code but that guy over there can't seem to test it in time." It's really easy to throw things over the fence. To actually make the change, we had to **change the accountability**. I no longer know how to live in a world where devs aren't accountable for tests. Before this change, we were in a nightmare situation where tests that testers run were different than tests that developers ran, so you didn't even know what was going to be tested. By the time they got around to testing it, the dev says, "Well I wrote that code so long ago I can't even remember how to fix it." That key pivot, making the dev accountable for the whole thing, made everything change. **It made the test tools get better. The test results got better.**

And Test Reliably

The quick, reliable signal that you get when you're trying to do a pull request, when you get the CI runs and when **you know red means red**, is indispensable. That only happened when we made dev accountable for the test, and not dev with some testers working for dev, we tried that too. It was really devs doing the testing. Other businesses, they might make a different choice, but as soon as you start throwing things around and you have all these handoffs, you're creating delay and waste in the system and the business gets too far ahead. We can't ship fast enough.

Same Tests Run Everywhere

Let's make sure that we get great signal-to-noise ratio. We have **zero tolerance for flaky tests**. Tests can be run quickly, they're run close to the code, 55,000 tests in six minutes, yeah that's what we're talking about here, so that every time you're submitting a pull request you know if it's good.

Move to The Cloud and Evolve in Flight

The key thing here was we had to evolve in flight That's why we took issues one at a time. We can't just do architecture improvements. We were shipping the product and shipping features to all of our customers all the time. What I see is that **feature flag saves lives**. That was a huge thing for us because once we started deploying all the time, we had a bunch of people on the team saying, "My feature can't be done in three weeks so I'm not going to merge and I'm not going to deploy."

The answer to that was feature flags, which was controlled exposure. Just turn it off and you get to merge and deploy, and you don't accrue debt. This is all about that speed of the machine.

Use Resiliency Patterns

Then of course we each had our on-prem architecture up there, and we had to **prevent the cascading failure problem**, which we found in single points of failure. We're eliminating those piece by piece. We're still in the process of doing that now, but we're learning how to prevent noisy neighbors and make sure people aren't misusing resources.

Security

The big thing here is to **make vulnerabilities real and personal** and then you're going to care. When the red team shows, I got into your code and I turned your whole dialogue upside down, and it's like, wait a minute, you can't do that to my code. That's much more real than a warning about XSS. We created that kind of culture and dynamic through that red team, blue team exercises that we do, where people take pride in hacking into each other's code or being able to block the attempts. That instilled a secure code culture.

We can't plan for every attack vector, but what we can do is assume that there's going to be a breach, and plan **how fast we can react to that breach**. A lot of the security work has been around that for our team.

Then lastly, humans make mistakes. They store passwords on file shares and we can tell them not to and we can send them to security training and we can do all these sorts of things. Many people learn but nobody will ever always learn. You can have all the sorts of lists of best practices but unless you're making that real, you have to **assume that people are going to make mistakes**, you have to go and check that they're being followed. and that you have ways to test that through. This is the same learnings as the testing problem.

Live Site Culture

Make live site the engineering team's problem. That was huge for us because in the past, I could go deploy something, sleep all night, have a nice weekend, come back Monday to 900 customer issues that the customer support team was dealing with and the ops team was trying to solve and all that sort of stuff, and I didn't pay the price.

If I don't pay the price, then I'm not incented to build in all the systems to never get to that point again. When I'm called at 2:00 in the morning, I notice.

We had to build on that and say, "No stop everything. **Live site is the most important thing that we do**." It's the customer experience they have right now and it's not just a tax or something like that. It's actually something people count on for us and we take pride in. It's a feature of our product. It's something that differentiates us.

Production Telemetry

In order to survive in that world, we need great alerting systems. Having unactionable alerts, redundant alerts, or overwhelming alert volumes makes you ignore all the alerts. Again, we said, "We better clean up this whole alert thing because it's gold. **Alerts need to be actionable.** This is the key to making sure we're jumping on the right customer issues and being able to do it as quickly as we can. Then we evolved to creating systems that can fail over and self-heal, so that all the sudden not only are we getting the alerts, but we can deal with them in the morning, not all night. All of that wouldn't have happened if we threw things over the wall to an ops team over on the side. It was because the dev team is sitting there. They're fighting to balance these improvements as a part of not just feature velocity, but engineering improvement velocity.

Running the Business on Metrics

About three years ago the company said, "We want to have a data driven culture." Don't we all want to be data driven? We had already started, just being in the cloud, learning that we needed a lot of telemetry and analytics instead of debugging in production. What was most important to us first was the health of our service. But then to change the culture and say, "We need to do this in all aspects of our decision making." We were already good at the kind of qualitative feedback loops, but not very good at the quantitative feedback loops.

Getting Metrics Right

I learned a couple of things through that journey. The first one was, **designing metrics is as hard as designing features**. I better not just say, "Oh here's a metric. We want engaged users." We redefined what engaged users on the service meant, three or four times in the first year and a half of measuring it. I don't need to measure vanity metrics, always going up. I need to measure the velocity ramp instead, is the velocity going up or is it just staying the same? And we learned that wow, different months have different days and need to be normalized.

Even now, every time we define a metric we have a conversation like a spec review. What are we trying to drive and are there sub-metrics we should be watching to make sure it's right? I didn't expect that much overhead as a part of becoming data driven. Lastly, we had to bake it into our culture with a monthly service review.

You're Not Done Until Telemetry Confirms You're Done

We bake metrics into our reviews up to Scott Guthrie, our executive vice president. We tell him every six weeks how we're doing on health, our business, our scenarios, our customer telemetry. We discuss it with the executives and then bring it down to the teams. We look at those same engaged user metrics and ask, "What does that mean for your feature?" People all through the org can say, "**Oh I don't just ship the feature, but now I go and look and see, Are people using it?** Or do I need to adjust the backlog to work on this feature more to make it achieve its goals?"

It's a Journey

It wasn't a straight line to get from A to B, nor is B the end. It wasn't rainbows and unicorns. We needed to move faster, not just do the same thing as before in shorter sprints. It means working differently. This is changing your rules and responsibilities for dev, test and ops. **It's a journey of continuous evolution and mistakes and learnings and we're still on the way.**

Lori Lamkin is the Director of Program Management for Visual Studio Cloud Services. She leads a team of program managers to drive the online developer experience to create cloud connected apps, collaborate with their team members, connect with community, all while adopting modern development practices. She joined Microsoft in 1990 supporting customers with the C/C++ language product, became the Unit Manager of C/C++ technical customer support and in 1993, and became Group Program Manager on version 1.0 of Visual Studio Team System in 2003. She holds a B.S. in Mathematics with an emphasis in Computer Science from the University of Washington. Lori enjoys reading, cooking, hot yoga, dance, tennis and spending time with her husband and twin boys.