Scaling your Customer Experience

Presented at the 2019 Texas Scalability Summit by Chris "fool" McCraw (@fool on twitter)

Summary (4 min)

In this talk, I'll tell you some real-world stories about effective and scalable technical customer support. This talk is targeted at anyone who works on a product or service that has customers - which is just about everyone, even if your customers might be "colleagues in finance" instead of "the unwashed masses of internet consumers". This is the first time I'm presenting on this subject, so I'd be excited to hear what you thought was best and worst in this presentation, and if there was anything you wish I'd expanded on more.. Note to the concerned: talk will be technical, but it will be human, too. It's not an overblown bullet list; it is a true story from the trenches.

First I'll explain a bit about myself and how I got here, which may or may not convince you that the problems we're solving at Netlify are similar to yours. But if not, hey, there are some other sessions going on right now that you'll then be

empowered to head over to in case my world doesn't turn out to remind you of yours.

In the main part of the talk, we'll explore some ways that Netlify's tech support team works with our Infrastructure, Development, and Product Management teams to both provide better tech support (our goal), and to improve and scale operations, development, and even feature implementation (their goals) - without hiring any new employees. While we talk, I'll explain in detail how our team does it and help frame some ways you could work with your own company's Support organization to get similar scaling effects. To communicate the stories in a way that makes sense, I'll also set the stage about our infrastructure and monitoring stacks, then I'll tell the story of a bad Friday afternoon deploy, and how the company reacted. After that, I'll move into some other Customer-impacting areas like QA, development, and roadmap planning. In the end, it is my hope that you are inspired to turn your own Customer Support team into collaborators and trend-finders to help other teams in the business (from product to operations to development) thrive, and you'll have some homework to do with your own Support or IT team if you want to help them level up to be the allies you could almost certainly use.

Time permitting at the end, we can do some Q&A about your situations or follow up on something interesting from the talk.

Intro (4 min)

I'm Chris McCraw, also known as "fool". I went to the University of Texas for a Computer science degree in the 90's, and while I was working on that degree I accidentally became a system administrator, because I turn out to like systems and networks more than programming.

In 1998, while working for the computational and applied math department at UT as a sysadmin, I built the first commodity off the shelf compute cluster on UT campus from 16 pentium 2 Dell desktops, in my office. Since then, I've been working on scaling computational solutions to increasingly large problem sets. Mid-career, I decided I was tired of running networks and saying no ("no, I can't make a hole in the firewall for you. No, you can't just have a copy of the database that has others' data in it. No, you can't run network attacks from my network.")

So, 8 years ago I pivoted into a developer support role, with a focus on "technology that helps people work smarter", which is where things get interesting. I've been providing developer support for a few tech-focused startups such as New Relic since then, as I enjoy helping startups figure out a better way for their Support and Engineering organizations work together. Today, I run the tech support team for web hosting platform Netlify, and we have built an extremely product-focused team of support engineers, and herein you'll some tales of how that is working out for us in a land of terabytes of traffic volume per day.

Netlify's stack (5 min)

This talk isn't a shill for my company. I love what we do, and you'll learn more about it as I describe the architecture and situations. At a high level, we're a web hosting company with many integrated developer-friendly, features that are intended to reduce the need for you to worry about how your continuous integration or webservice scales, or is configured. Our data isn't traditionally big data, but we have a lot of customer data and a TON of log data that is a primary resource in analyzing changes in our platform's behavior over time plus debugging individual customer problems as well.

serving

We serve over a hundred billion of web requests a month at a rate of ~40k requests per second, all through Apache Traffic Server with a custom plugin for it which we've written to handle a lot of special features in our implementation. We run our own CDN using 7 separate cloud providers - from AWS to yandex, Digital Ocean to Rackspace, Google Compute to Host Virtual, and we will add new cloud providers like Alibaba by the end of the year. ATS nodes on each provider in dozens of PoPs front a several-layer-deep custom distributed caching and proxying implementation written in a combination of Go and C++ and Rails. Assets may be served directly from a Mongo database (rarely, in case cache isn't available yet), a short term memcache store, a longer term cache at GCE which is mirrored at AWS and Rackspace, or a per-node memory and disk cache. We may also serve content that you've configured us to reverse proxy to, from another site of ours or any service on the internet (such as heroku or your own data center). Any piece of this infrastructure can cause a request to fail, including the parts we

don't control directly like reverse proxies. **So, being able to understand and debug the system requires a fair bit of training, experience, and a great toolkit.**

monitoring

Speaking of toolkits, the main tools we use for monitoring and alerting are used by both our Support team in diagnosing, and our infrastructure team in their day to day work:

- datadog for OS-level stats like CPU/mem and other APM,
- humio for loglines, ~2 terabytes a day
- pingdom for uptime/RUM monitoring,
- pagerduty for alerting about all of the above
- slack integrations for all of the above so hopefully you see the problem before the page is created.
- statuspage.io for telling the world
- and one of the most important monitoring tools of our infrastructure: our helpdesk and twitter, where customers let my team know very quickly if something is wrong.

ci

There's a fair amount going on in our infrastructure already, so let's complicate things! We ALSO run a CI environment for most of our hundreds of thousands of customers, and this is a full linux shell inside a docker container, spawned by our API via kubernetes. When it runs, it pulls your code from a git server, and fetches your site's build settings from our Mongo database via our rails API, and then executes your build command(s) for up to 15 minutes to create your site assets from source code, before deploying to our CDN. This is the source of a lot of our support questions, and it can be a bit difficult to determine customer-facing service degradations vs "lots of customers just using unix wrong", so we need toolkits that help us filter, find trends, as well as examine specific errors in depth.

Story of the bad proxy deploy on a friday, and how big data helps! (15 min)

Our engineering teams at Netlify are great, and we do a lot of test-driven development. Still, as you might guess from the longwinded description of our stack and traffic volumes, there can be unforeseen issues with production deploy based on simpler non-production testing, at volumes many orders of magnitude lower than our production traffic levels. Plus, customers are the best at creating unanticipated edge cases with their interesting configurations. I'm guessing there is another talk here about scaling your testing, using canary deploys (we do this, but it can't catch every problem) and feature flags, so if you want to learn how to solve this from that direction, I suggest you do that in parallel, since the best time to find a problem is before full production deploy obviously :)

Regardless, since our Support team gets the firehose that is our customers' feedback, we are the second line defense in case any engineering team misses some problems during testing of all stripes. **This is already the case for your Support team** - no matter how you provide support - chat, forums, twitter, a dedicated help desk, walk up office hours - they hear from the customers when something changes for the worse. So - you could treat them effectively as another alerting channel - "something's wrong from Support!", and that is a part of what I'm proposing, but let's explore a bit the best practices for enabling GOOD trouble reports, rather than just passing on that firehose of customer sentiment!

First, your support team needs to do some basic filtering. Many or even most customer contacts will be due to a problem, so they will be a very noisy alert stream with a lot of false positives if you take the unfiltered "feed" from them. So, it's in everyone's best interest to ensure that the Support team understands a few things like:

- how your system works. This includes not just what successful operations look like but also an understanding of what failures are related/identical, and what the effects of failures are ("breaks one website" vs "breaks every website")
- how to understand when changes were made to a system, so analysis can reflect whether it is a new problem caused by a system event, or one that has been there forever and just got noticed by someone who happened to tell you.

- how to troubleshoot a problem. Essentially, this isn't just "seeing something is wrong" but figuring out causes and effects where possible, and coming up with reproduction steps. (maybe use a slide from md's talk here?)
- how to communicate these issues in a useful way to other teams. Filing a bug is one path, paging an on-call engineer over the weekend is more impactful and immediate. Either one should have a set of reproduction steps since otherwise we're back to "Something's wrong from Support!". If the fireman shows up and you can't lead him to the fire or at least the unexpected smoke, he's not going to be much help and you might be in trouble for wasting his time.
- how to determine the frequency of a problem. At over a million requests per second, any one request's behavior is very much noise. Even a hundred are still a very small fraction of a percent and probably still not statistically interesting. Orders of magnitude are quite important!

Now, for the story!

One Friday in 2018, we had a somewhat urgent fix to a problem in our reverse proxying service, wrote some tests, and then some code to implement a fix for the tests, and deployed to production at 4pm. Our office closes at 5pm, for the weekend - reopening when the Support team starts up with the Asia/Pacific business week on Sunday evening US time. We generally don't deploy that late in the day on Friday since while all the monitors and paging systems do their jobs over the weekend, unless the problem is **alertable**, nobody will notice that something is broken, except the customers. And since my team isn't there to observe & relay the customer sentiment, that effectively means that ONLY our customers notice.

Fortunately, one of our VIP customers noticed the issue, and fortunately for all of our customers, our Support team is on call over the weekend for VIP customers, so I was paged while at the bar at about 6:30pm. I pulled out my laptop to take a look at the situation so I could respond to the page. What I could see were some unexpected HTTP status code responses for a few requests on one website. For most of our customers, even the bigger ones, their traffic is not statistically significant, and these errors were a fraction of a small (but valuable) customer's

single website. So this was far from a statistically significant problem that the customer observed, but of course this customer was upset that his e-commerce site's checkout wasn't working, and paged us for help. His reproduction steps' results didn't match mine, which meant I needed to keep digging to either understand the difference or do something about it.

Casting a broader net, I looked for an uptick in the error message that accompanied his site's handful of failed requests across our service using humio. What I found was that this error wasn't unique to this customer's account - but had instead happened to dozens of customers. Still - a small mystery on a Friday night isn't a good cause for bringing someone else back to work.

Looking for more clues, in correlated events, I checked our deploys listing and was reminded of that high urgency fix a few hours prior. The charts of frequency of the error started right at deploy time, accelerated as the roll out covered more and more of our network, and finally were happening on every host on one of our CDN's after the deploy + restart had completed. Now, I had a great theory to share with someone!

Fortunately, I've spent years now developing a great relationship with our Infrastructure lead - we've taught eachother a lot (but mostly him teaching me about our systems and how to use the debugging and monitoring tools). The net result of this training was that not only was I able to diagnose, correlate, and assess impact, but I had a great report for him including links to the charts showing the correlation and frequency of the problem, so my page looked less like "Support needs help!" and more like "Support sees that your last deploy of the proxy service has caused a new class of problem affecting all customers who use AWS lambda functions with our service on this CDN. Here's the evidence; we request a rollback ASAP".

He logged in, read what I had to say, concurred with my judgment, and rolled back the service, and we both went back to our beers after I informed the customers who had written in (by this point there were a few in addition to the VIP who could page me). While I was informing folks, a couple people reported success so that showed that things were functionally/obviously repaired from the customer point of view. From humio, I could see the chart of errors had dwindled back to zero, so I felt comfortable having another beer knowing that disaster had been averted.

Anatomy of a recovery (6 min)

But, the story isn't over yet. What happened on Monday?

A lot of our practices look like the practices at other tech companies: We have a retrospective meeting (not a post-mortem; nobody died!) where we talk about what went well and what went poorly, and also, how we'll avoid repeating the same problem with the same cause in the future, and what if anything we should tell our customers about the incident. Our Support team, as first responders and often alarm sounders, and keepers of copious notes about ever situation courtesy of our heldpesk, almost always participates in these meetings.

We started by looking at how our normal testing had missed the issue: unexpected "edge case" that did affect less than one percent of our customers (kudos to humio for giving us a certitude in the numbers!). The edge case is now a test case for the fix.

Then we tried to understand how our alerting had missed this issue: the error message was new, and the status codes from the HTTP error code percentage never went over a threshold of criticality, a risk we always have to balance for: between alerting for errors from customer's proxy'd content servers, vs ones caused by something unexpected our own service did.

Then we talked about how we would prevent it from happening again, and besides the reminder about the "friday afternoon no deploy" rule and the new test case, it was one of the use cases that led to us having canary deploys that we could monitor more carefully and choose not to leave in rotation off hours. We also put some time around engineering a more alertable system onto the product roadmap, and support prioritized it over many more frequently encountered bugs, since the symptoms of this bug were larger even if rarer.

Finally, we talked about what went well, and this is where we find all the winning elements of our Support/Engineering collaboration:

1. Our Infratructure team had taken the time to train Support on how the system worked, and paired with us on prior bug reports, so we had a sense of how to look for information and what kind of information was likely to be useful, and how to use all of our great tools like:

- APM and other discoverability tools, like datadog, so we could spot trends and share the details of what we'd found.

- The log ingest and search tools were well configured, so I could slice and dice by customer, by URL, by status code, by error message, by server, by service, by time, etc.
- discoverability of deploys across our infrastructure in several slack channels.
- The rapport we'd built w/platform that led to me being able to escalate successfully and painlessly, even on a friday night.

2. Developing an engineering relationship with customers - when they are our allies in debugging, and when we're transparent with them about changes and system behaviors (even misbehaviors), their reports are more useful in our own debugging.

3. Finally we needed to think about whether and how we announce retroactively to customers? In light of the company value of transparency, and wanting real customer reports about behavior other than what we know happened, we tend to open incidents more often than not, even if most customers aren't affected. Our Support team is empowered to make some decisions on our own and we drive the policy on what we talk about publicly. (slide of Moss putting the fire with the other fire?)

How you might develop a successful collaboration with your Support team (4 min)

- Helping your support team think like engineers. Train them to embrace their engineering traits (reproducibility, proof, backing data, trend detection, filtering, etc). This leads to better escalations.
- Working to develop a model of actionable feedback (instead of "this is broken", "this could meet more use cases we've heard about if behavior changed ")
- teaching the support team what matters to you, so they can be on the lookout for the things that are most important - performance? bugs? feature requests? But also "billing-related bugs" or "places where people struggled to

succeed at configuring DNS" that might be reported in many different contexts using many different words.

- develop and document the troubleshooting path (slide with internal-docs titles for debugging cache consistency for support and for ops)
- teach and keep current a thorough understanding of your system and its internal architecture and interactions by the people who receive the most feedback about new problems the most quickly.
- developing an engineering relationship directly with your customers ?

Some other examples if time permits.

few minutes: Story of launching new relic community and turning off free helpdesk support - "what do your customers think" - your Suppor team knows!

---

few minutes: value of providing good, personal, thoughtful support for even free customers

1. free user feedback and testing. We have no dedicated QA staff, but we do have a few hundred thousand folks who find the rough edges and use cases we hadn't considered. Then our support team can collate those trends and report on them, while characterizing "customer temperature" or in essence, impact. Your product team can do the same but it's hard for them to work at the same scale as a helpdesk or your logstream!
2. guidance on product growth: "we're trying to improve microservice A. these are our plans! none of those address any customer pain points which the top ones are X, Y and Z - and you seem to think Z works well since you're basing your improvements on that part of the codebase, but there are a few major bugs and gaps in implementation that will block widespread adoption. Let me show you some of the use cases from customers..."