Chapter 1. Context Versus Control in SRE

A discussion with Coburn Watson, Microsoft (formerly Netflix) and David N. Blank-Edelman

David: We've had the pleasure of talking about a lot of things in the time we've known each other. One of the most interesting things I've heard you speak about is a way of doing SRE that focuses on providing context instead of using processes that are centered around control (the more common way SRE is practiced). Can we dig some more into this? Can you explain what you mean by context versus control and what a good example of each would be?

Coburn: I think of context as providing additional, pertinent information, which allows someone to better understand the rationale behind a given request or statement. At the highest level, availability-related context as shared at Netflix with an engineering team would be the trended availability of their microservice[s] and how that relates to the desired goal, including availability of downstream dependencies. With this domain-specific context, an engineering team has the responsibility (and context) to take the necessary steps to improve their availability.

In a control-based model a team will be aware of their microservice[s] availability goal, but if they fail to achieve that goal there might be a punitive action. This action might involve removing their ability to push code to production. At Netflix, we err toward the former model, sharing context on microservice-level availability, then working with teams when needed to help improve availability.

The challenge is making sure sufficient context is provided to teams. When someone makes a nonideal operational decision at Netflix, the first question to ask is whether that person had sufficient context to make a better decision. Quite often, the SRE team will find that a hit to availability is a result of insufficient context being passed to the team, in particular context related to reliability. Those are the gaps we seek to close as an SRE team to improve overall availability.

In a very large organization, it can be challenging to provide enough context such that, based on context alone, people can achieve the desired availability goals for their services. In organizations of this scale, you often have to fall back on more processes to achieve your availability goals. One example is the Google error budget model.[1] Another case for a more control-based model is when lives are on the line. If someone frequently writes unsafe software for an airplane autopilot system, that person (and company) probably has a very low tolerance for a primarily context-based approach. They don't want to get together and figure out how to improve their availability through additional context if planes are falling out of the sky. It's up to each SRE organization to determine how much risk they can assume as one factor in finding the split between a context- versus control-based model.

I believe there is a difference between information and context. In systems monitoring, information could just be a bunch of availability metrics I jam into a dashboard and email to the team. A typical engineer receiving such an email would ignore it because, 1) they are in charge of writing business logic for a service, and 2) they lack the expertise to digest and understand resource and availability metrics presented as time series.

At Netflix we have hundreds of thousands of operational metrics available to us. In order to support a context-driven model to improve availability, we have to bring specific domain knowledge to bear on the data. This requires taking the information and massaging it into a format that tells a story about availability. By applying such a transformation, we are able to push this context to teams on an as-needed basis so they can measure if availability improves for the given microservice. As an example, one key availability metric is the trended success rate from dependent services on a given microservice (as measured from the client side and breaking down failure rates based on cause).

My team doesn't own availability, but our job is to improve it over time. Why? Because someone can always blow a tire off the car. Oftentimes teams reach out and say, "I'm not quite sure why my availability dropped; can we talk about that?" While investigating the situation, it could be discovered that someone modified a client library or changed a timeout setting. As mentioned earlier, it is important to start from the principle that people are not operating in a negligent manner; they're

just lacking the context to make the better decisions. We also don't forget that the systems can be overly complex and the operational bar required to avoid incidents is both too high or unnecessary. An example in this latter category is the tuning of static timeouts in a dynamic system.

Though a context-driven model is ideal, you can drift toward the needed control when you see a team having repeated incidents. They have been provided the same effective availability context as other teams, yet their availability continues to suffer. In some companies, control takes the form of prohibiting code from being pushed to production, and then of course engineers come to management because they need to push code to production. At Netflix I'll lead with the "you're on my list" discussion with an engineering manager, which involves in-person additional level-setting around expectations of service availability. If there is a common view, then I'll mention that they are developing a lot of features, but not addressing the necessary changes required to improve availability. I end by asking, "How do we get those on your list?" In my mind, that's about the extent of control I really want. I am lucky to work in a company with a mature set of individuals who recognize the importance of availability to the business and prioritize efforts appropriately. I find this type of discussion typically gets the needle pointing in the right direction.

Mentioned previously, but worth calling out again, before I make context-versus-control seem like a utopia that everyone can achieve, just remember that we run a business where if we have a failure, maybe streams don't get delivered. We don't have planes falling out of the sky or people's heart machines stopping. That gives us more flexibility about where we fall on the context-versus-control spectrum.

David: Does this mean that a context-based approach works at smaller scales but not nearly as well at larger scales?

Coburn: I believe the "scale" at which context over control works has little to do with the size of the production environment or customer base. It's more about the organization size and how they operate between teams. As you grow an organization, the effective use of context for availability can be challenging. A key factor may be less one-on-one communication or face-to-face time as the company grows.

At Netflix, I have the luxury of all the engineering teams I engage with residing on one campus. I can have coffee with any manager at Netflix. I've worked at large companies like HP with teams sitting on the other side of the globe. In that global situation, I still get them appropriate context, but it requires more work to give effective context. My assumption is if a company leans in and starts with control it's primarily because (regardless of scale) they're much more comfortable leading with process and control.

David: Is there a way that infrastructure plays into reliability as related to context?

Coburn: We consider ourselves in our push-model perspective to have great reliability because we're immutable in nature. I think we've all worked at companies where we upgrade something, and it turns out it was bad and we spend the night firefighting it because we're trying to get the site back up because the patch didn't work. When we go to put new code into production, we never upgrade anything, we just push a new version alongside the current code base. It's that red/black or blue/green—whatever people call it at their company. The new version of code goes in, and if something breaks in the canary, we just take it back out and we immediately roll back, or we don't even go through with the full deployment. This brings recovery down from possibly hours to minutes.

Although we have immutable code deployments, we also have something called Fast Properties. This means we can go into production with a later ability to dynamically update an application property. Since this breaks our immutable model but is often required, we see this capability being overused and leading to production incidents. Much like other common problems in production, if we see a team or collection of teams stumbling over the problem of dynamic property management, we look for ways to remove the risk through improved tooling. In our production environment, we now use Spinnaker, our continuous delivery platform, to manage staggered rollouts of dynamic properties to minimize blast radius and identify issues early. We refer to this general strategy as a "guardrail."

Even once we've done that, sometimes teams still go change something manually instead of going through a fast property pipeline and break

production. If it happens again, we're like, "OK, they clearly haven't gotten the message. We need to go meet with them and say, 'please don't push this button'." We try at all costs to not take away the button, but at some point, we probably will.

David: This brings up a question about feedback loops. It sounds like some of your feedback signals are "did the site go down?", "did something trend positive?", or in terms of money "did it trend negative in the way you wanted it to?" Is there a more direct way to get an understanding from the humans whether they got the context they needed besides observing the system in a black box fashion and seeing whether the indicators you were looking at changed?

Coburn: That's one of the evolutions we've made as our company's grown. At Netflix we probably have 2,000 engineers, and that covers all different product domains. I could be building an internal tools UI, an encoding engine, standing up an infrastructure stack, or something to give you better recommendations. Some of those efforts have a more significant impact on customer-facing availability than others. From a numbers perspective, probably 50% of our engineering teams on a given day can do whatever they want in production and it has no possible risk to our service availability in any way whatsoever.

Initially we used to walk around the hall banging the virtual pan and yelling, "Hey, our availability is at three-nines; this is a problem!" The problem is it's a very diffuse message and 50% of the people are like, "OK, great, um, you're saying availability is not good, but what can I do about it as my services can't impact availability?"

Then, we changed our messaging to focus the banging of the pan on the teams that actually affect availability, which might be the other 50% of the population. So, now, at least our message about service availability is hitting the right audience. The next step is to funnel context, which is specific to each engineering team and allows them to evaluate if they have substandard availability.

But even figuring out who to provide this information to isn't always easy. In a microservice architecture, there might be 40-plus teams that actually have microservices running at any given time, which can impact availability on the service critical path. If we're measuring availability at

the edge of our service and determine that one-tenth of 1% of users couldn't play movies in a given hour, it's quite challenging to identify which team might have actually driven that outage. Even more challenging is that, in many cases, it's externally driven. As one example, think of an online gaming platform that needs to be available to play a title on the service. In that case perhaps I have to go to the UI team at Netflix that depends on that service and see if they can build resiliency to the external vendor service failure (which we have done).

I find it helpful to be clear within your organization when referring to the terms "availability" and "reliability." As an analogy to how we use these terms at Netflix in the context of our streaming service, I refer to a disk array which might be part of a Storage-Area Network. If the disk array represents the service, the underlying disk drives in the array would represent microservices. The design of the disk array takes into account that individual drives might fail, but the array should still function to both serve and persist data. Using this model, you would calculate availability as the percentage of time the disk array (the Netflix streaming service, in my world) is able to provide service to a client. The rate at which drives fail in the array represents the reliability of the drive (in my case, a microservice).

Much as with disk array configurations (RAID configuration, caching, etc.), you can apply operational patterns in a microservice architecture to improve overall service availability in light of possible microservice failures. One such example in use at Netflix is the Hystrix framework, based on the bulkhead pattern, which opens up "circuits" between microservices when a downstream microservice fails. The framework serves a fallback experience when possible to still provide service to an end user.

In conclusion, setting and measuring reliability goals at the microservice level lets you move toward a desired aggregate service-level availability. Different organizations might use the terms availability and reliability differently, but this is how we define them in light of our operational model.

David: So, what sort of info can you give to that team that would be actionable?

Coburn: Another company I talked to had a microservice availability report. They looked at the rate at which services were providing successful calls to their neighbor. If I'm a service that handles subscriber or member information and I have 30 services talking to me, my primary measure of availability should be the rate of success my dependencies have. Many times, teams will focus on their service-side view of being up and running. This doesn't guarantee you are servicing requests successfully at the desired rate, but it all comes down to the measures.

We liked the model from the other company and looked to see how we could apply it to our own infrastructure. Luckily, we have a common IPC [interprocess communication] framework and we have metrics around Hystrix and other commands that record the rate at which a client (dependent service) is having successful calls. We aggregate this information for all our critical microservices and compare the trend to the previous 30 days. It isn't meant to be near real time, it's meant to be more operational—are you getting better or worse.

Let's say a team owns three microservices, and the goal is for the microservices to have four-nines availability for calls from clients. If these microservices stay above four-nines, the team will never receive an email (availability report). If the last 7 days compared to the previous 21 have a deviation in a negative manner, or if the microservices are failing to achieve the four-nines, a report will be sent to the team. The report shows week over week as a bar chart. Green indicates you're achieving the goal for a given week. Red or yellow means you're failing to achieve the goal. Yellow means improvement over the previous week; red indicates degradation. Clicking on a bar in the graph will deliver a detailed report that shows for the entire window call rates of upstream (client) services talking to the microservice and the availability to those calls. On the right panel is a view of downstream dependency call and success rates, which is helpful as the microservice availability might be reduced by downstream dependency. We call this internal implementation the "microservice availability scorecard framework." At this point, the SRE team (working with Data Analytics) has provided information to a team about the microservices they own, and in particular their microservice availability to dependent client services, independent of Netflix service availability. This should be very actionable information. When dashboards are continually pushed to engineers, they can stop looking at them relatively quickly. The SRE

solution at Netflix is to only push a scorecard when something changes that a team might need to pay attention to.

David: So, this sounds pretty *reactive*, right? You send them a scorecard because you'd like to see something that already happened be different in the future. What's the *proactive* side of this? How do you help someone who is trying to make a good decision well before they would get a scorecard?

Coburn: A good analogy for the scorecard is a message on your car dash indicating "your right front tire is losing pressure." It doesn't tell you what to do about it, just provides information that a trend is going in the wrong direction. When thinking about how to best inform people, I leveraged experience from past work in the performance domain, where we're not as worried about catching large performance deviations. We have canaries that evaluate a system looking for significant deviations, so if CPU demand happens to jump 20% on a push, alarms start going off. What ends up getting you in the long run is the five-millisecond increase week over week for a six-month period. At the end of which you say, "Wow, I'm suddenly running with three times as much capacity for that service; what happened?" The role of the scorecard is to catch smaller deviations as well to avoid that slow and dangerous drift. I think the analogous situation for reliability is if you ignore the small deviations and fail to proactively address them, you're just waiting for the big outage.

David: OK, so what do you do from a contextual perspective to allow people to make the right decisions in the moment? In theory, the error budget concept means that at any time *T*, I have a way to determine at that moment whether I should do or not do something like launch a new version. What is the contextual version of that? Is the theory that people look at a scorecard synchronously and then make the right decision?

Coburn: If someone is having a problem where they're significantly impacting actual production availability, such as consistently having production incidents that result in customers not being able to stream content, then the microservice availability scorecard context is probably not the way to solve that problem. In that case, they have most likely been receiving the report but not acting on it. The way to solve that problem is to gather in a room and figure a path forward. It still doesn't

mean they should stop pushing code, because they have a lot of other services dependent on them.

To your question about more real-time inputs to help engineers make better deployment decisions in the moment, we strive to put the information inside of the tools they are working with. Spinnaker now exposes availability trends at the cluster (the AWS term is Autoscaling Group) and it is right in front of the engineer if they are making a change via the UI.

When we look at continually improving availability, the target falls into two primary categories: avoiding all incidents, regardless of type, and a specific failure pattern someone has that's causing an incident or repeated outage.

From a context perspective, if they're having a problem where they're making a set of decisions over and over that result in production-impacting incidents, that requires a human-delivered versus a system-delivered report. At Netflix, the Core SRE team doesn't have the responsibility to step in and troubleshoot and resolve microservice-specific issues, but the team is instead considered the "central nervous system of Netflix." Taking the central nervous system analogy a bit further, this is the only team at Netflix that sees all the failures, whether external, internal, CDN [Content Delivery Network], network, etc., with the resultant responsibility to determine which teams to reach out to and engage.

David: Viewing your group as the nervous system of Netflix, what process do you have for propagating information throughout the org so that people in one part can learn from the other parts (either a good way to do something or not to repeat a bad experience)?

Coburn: Based on the specific shortcoming in operational risk we would like to address, we have a couple of channels to get the best practice applied: drive to get the necessary enhancements into our operational tools (for example, Spinnaker, canary analysis) to seamlessly expose it to all teams or socialize the suggested change via an availability newsletter which is published monthly. In some cases, the suggested best practice is one of the tooling extensions previously incorporated into tooling.

Both of these are methods to help move the needle on availability in an aggregate manner.

Some changes that we incorporate into tooling might be referred to as a guardrail. A concrete example is an additional step in deployment added to Spinnaker that detects when someone attempts to remove all cluster capacity while still taking significant amounts of traffic. This guardrail helps call attention to a possibly dangerous action, which the engineer was unaware was risky. This is a method of "just in time" context.

David: What about cases where somebody uses one of your tools like Hystrix in a way that is a perfectly valid way to use it, but experience has shown that this way yields negative results? It's not the case that you want to change the tool because the tool is just fine. How do you propagate experience like this through the org once you've learned that lesson? It sounds like the availability reports would be one of the examples of that?

Coburn: That's right. The availability report is actually challenging to interpret. Once you receive it, you need to click in three to four strategic (and often hidden) places on the report just to get to actionable data. This was a fairly significant barrier to adoption, and so we created a four-minute video to function as an onboarding tutorial for report usage. When people get the report, it says please watch this four-minute video to understand how to use this report. It didn't move the needle fully but definitely improved the actionability of the report for those who took the time to use it.

David: Let's talk about the limitations of context versus control. Do you think that a context-based system works as well in an environment where the product and the metrics for it are more complex than Netflix? Is it the simplicity of the Netflix product that makes it work so well for you?

Coburn: First I would point out that in aggregate the Netflix service (in terms of its interacting parts) is as complex as you might find anywhere else. There are a number of factors that allow context to be more broadly effective at Netflix versus other companies. It ties directly to a lack of complexity in the engineering organization and the structure of our service. The following are some of the key factors:

- We have all the engineers sitting in one location. Common engineering principles are easily socialized and additional tribal knowledge of best practices is implicitly socialized and discussed.
- We have one dominant version of the software running in a given region at a given time. We additionally have control over that software stack and we can change it whenever we want (not shipped to customers, with the exception of device applications).
- We have a service where people don't die if it breaks.
- We have a little bit of runway and we have an SRE team who is willing to take a bunch of bullets if availability gets worse and say "my job is to fix it."

For this last bullet point, it can be somewhat of a difficult situation for the reliability organization to say they're responsible for improving availability, but at the same time don't have total control over the factors that impact availability. A lot of people aren't comfortable with saying they're responsible for improving something they don't necessarily have absolute control over.

At a company where you ship a product to the field (for example, self-driving car software), this model might not work. You wouldn't want to wait and say, "Wow, look, that car drove off the road. That was an error. Who wrote that? Well, let's get it right next year." In contrast, the space we operate in allows a fair amount of room to leverage context over control, as we don't put lives at risk and can make changes quickly to the aggregate product environment. That said, I think you can see situations when companies start to scale where they started with context but move over time to have a bit more control as warranted by the needs of the customer and reliability. You could generalize this perspective in that you will eventually move more toward the other end of the spectrum from which you started, if even only a little bit.

David: In the past we've talked a bit about context versus control from an engineering investment perspective. Can you say a bit more about this?

Coburn: I think about four primary dimensions: rate of innovation, security, reliability, and efficiency (infrastructure/operations). Depending on how your organization is structured, you could own one

or more of the dimensions. Even if you don't own a given dimension, the decisions you make in your own space can have significant impacts on the others. I'm in a situation where I own at least two, reliability and efficiency, because I also own capacity planning. When I think about how hard I want to push on teams to improve those dimensions I want to try as little as possible to reduce drag on the others (rate of innovation, security). Keeping this model top-of-mind allows us as an SRE organization to be more thoughtful in our asks of other engineering teams.

Somewhat related, we also make explicit trade-offs in order to improve one dimension at the cost of another. We have cases where we run significantly inefficient (over-provisioned) for a given microservice for the purposes of increased rate of innovation or reliability. If we have a service that needs to run with twice as much headroom because of bursty traffic, I don't care if that costs me an additional tens of thousands of dollars a year, because all I need is one big outage to quickly burn that amount of engineering effort/time.

David: OK, so how do you connect that back to context versus control?

Coburn: With control as an example, let's say I actually take away the ability for someone to push code because I need to increase this dimension of reliability. On the one hand it stops the near-term breakage, but it slows down innovation. The goal is to try to provide more context to avoid forcibly slowing down other dimensions and let the owner of a given service determine which dimension to optimize for, taking into account their current demands. In contrast, control has a much more binary effect on other dimensions.

David: Does context have a positive or negative effect on any of those dimensions? Control clearly does, as you just mentioned, but what about context?

Coburn: Context does have a strong upside and track record, at least in our case. In the past five years, our customer population has grown six times, our streaming has grown six times, our availability has improved every year, our number of teams has increased every year, and our rate of innovation has gone up every year. That's without applying control to improve availability, but rather predominantly by improving through

context combined with improvement of the platform so engineers spend less time working on operational aspects that shouldn't be required in their role (such as determining how to configure pipelines to have staggered pushes across regions).

David: If control can have a negative effect on those dimensions, can context can have a similar adverse effect?

Coburn: It can, but I have yet to see where the context we provide to teams results in an undesired behavior that actually harms availability. One area in which this can be a risk is infrastructure efficiency. Although we provide detailed infrastructure cost context to teams, we don't have or enforce cloud cost budgets on teams. Case in point, engineers simply deploy whatever infrastructure they need and receive a cost report each month that provides them context about their cost. With this context one team chose to try and optimize efficiency by staying on an older and less reliable instance type, which was of course much less expensive. We then had an outage due to poorly performing and less reliable instances, often running underprovisioned. In this case, we provided cost context, the decision was to overindex on efficiency, and as a result reliability took a hit. We decided to socialize with teams that we would rather run less efficient than compromise reliability, aligned with our explicit prioritization of domain concerns.

Coburn Watson is currently a partner in the Production Infrastructure Engineering organization at Microsoft. He recently concluded a six year adventure at Netflix where he led the organization responsible for site reliability, performance engineering, and cloud infrastructure. His 20+ years of experience in the technology domain spans systems management through application development and large-scale site reliability and performance.