What is Site Reliability Engineering?

In this interview, Ben Treynor (VP, Site Reliability Engineering) shares his thoughts with Niall Murphy (Site Reliability Manager) about what Site Reliability Engineering (SRE) is, how and why it works so well, and the factors that differentiate SRE from operations teams in industry.

Editorial note: an earlier version of this interview referenced the MTTR equation in an ambiguous fashion; this has now been corrected. Thanks for reporting the problem.

Niall: So what is SRE?

Ben: Fundamentally, it's what happens when you ask a software engineer to design an operations function. When I came to Google, I was fortunate enough to be part of a team that was partially composed of folks who were software engineers, and who were inclined to use software as a way of solving problems that had historically been solved by hand. So when it was time to create a formal team to do this operational work, it was natural to take the "everything can be treated as a software problem" approach and run with it.

So SRE is fundamentally doing work that has historically been done by an operations team, but using engineers with software expertise, and banking on the fact that these engineers are inherently both predisposed to, and have the ability to, substitute automation for human labor.

On top of that, in Google, we have a bunch of rules of engagement, and principles for how SRE teams interact with their environment -- not only the production environment, but also the development teams, the testing teams, the users, and so on. Those rules and work practices help us to keep doing primarily engineering work and not operations work.

Niall: How is this reflected in the day-to-day work and responsibilities of an SRE team?

Ben: In general, an SRE team is responsible for availability, latency, performance, efficiency, change management, monitoring, emergency response, and capacity...
Many operations teams today have a similar role, sometimes without some of the bits that I’ve identified. But the way that SRE does it is quite different. That’s due to a couple of reasons.

Number one is hiring. We hire engineers with software development ability and proclivity.

To SRE, software engineers are people who know enough about programming languages, data structures and algorithms, and performance to be able to write software that is effective. Crucially, while the software may accomplish a task at launch, it also has to be efficient at accomplishing that task even as the task grows.

During our hiring process, we examine people who are close to passing the Google SWE bar, and who in addition also have a complementary set of skills that are useful to us. Network engineering and Unix system administration are two common areas that we look at; there are others. Someone with good software skills but perhaps little professional development experience, who also is an expert in network engineering or system administration — we hire those people for SRE. Typically, we hire about a 50-50 mix of people who have more of a software background and people who have more of a systems engineering background. It seems to be a really good mix.

We’ve held that hiring bar constant through the years, even at times when it’s been very hard to find people, and there’s been a lot of pressure to relax that bar in order to increase hiring volume. We’ve never changed our standards in this respect. That has, I think, been incredibly important for the group. Because what you end up with is, a team of people who fundamentally will not accept doing things over and over by hand, but also a team that has a lot of the same academic and intellectual background as the rest of the development organization. This ensures that mutual respect and mutual vocabulary pertains between SRE and SWE.

One of the things you normally see in operations roles as opposed to engineering roles is that there’s a chasm not only with respect to duty, but also of background and of vocabulary, and eventually, of respect. This, to me, is a pathology.

Niall: Outside Google, we often observe that there isn’t parity of esteem between the SWE and operations teams, which combines poorly with the fact that they often have different incentives. That’s how we end up with the model that exists in the industry today, where SWE teams write something and throw it over a wall to the operations teams, who then try to make it work, and can’t, and throw it back, and so on.

Ben: It’s interesting in this context to also look at the organizational differences that make SRE what it is, not just the individual work habits.

One of the key characteristics that SREs have is that they are free to transfer between SRE teams, and the group is large enough to have plenty of mobility. Additionally, SWEs are free to transfer out of SRE. But, in general, they do not.

The necessary condition for this freedom of movement is parity between SWEs in general, SREs who happen to be SWEs, and compensation parity between those and systems engineers in SRE. They’re all groups that are held to the same standards of performance, the same standards of output, the same standards of expertise. And there’s free transfer between the SWE and the SRE SWE team. The key point about free and easy migration for anyone in the SRE group who find that they are working on a project or a system that is “bad” is that it is an excellent threat, providing an incentive for development teams to not build systems that are horrible to run.

It’s a threat I use all the time. I say, “Look, basically, we’re only hiring engineers into SRE. If you build a system that is an ops disaster, the SREs will leave. I will let them.” And as they leave and the group drops below critical mass, we will hand operational responsibility back to you, the development team.

In Google, we have institutionalized this response, with things like the Production Readiness Review (PRR). The PRR helps us avoid getting into this situation by examining both the system and its characteristics before taking it on, also by having shared responsibility. This is the simplest and most effective way I know to remove any fantasy about what the system is like in the real world. It also provides a huge incentive to the development team to make a system that has low operational load.
Niall: It is important to get the incentives right. To that end, why is it important that SREs be scarce?

Ben: Early on, we didn’t have much in the way of choice. We could not hire SREs as fast as the demand for them, so there was always scarcity. I used that to our advantage by simply saying, “We will assign SREs to the places where they’re going to do the most good”. Operational only projects have relatively low ROI. I don’t put SREs on those. There are other projects which are much more mature and established, and SREs on those projects would have a much higher marginal benefit. Historically, what we have seen is one SRE will replace two SWEs doing the same work, partly because they tend to be expert at production-related technologies and the support model. Roughly, that’s the equation that has helped SRE demand stay high. That demand in turn being higher than supply has helped us avoid being pulled into some engagements where the engagement itself would have been sketchy.

Niall: How does SRE make sure that an engagement, once started, is correctly maintained?

Ben: SRE has quarterly service reviews, which are intentionally designed to measure the amount of ops workload that a team has. You can get used to this workload and spend most of your time working on ops only stuff. The reviews are an external check to make sure that if you fall into that mode we notice and take corrective action, which can sometimes mean dissolving the team!

We care very deeply about keeping SRE an engineering function, so our rule of thumb is that an SRE team must spend at least 50% of its time actually doing development. So how do you enforce that? In the first place, you have to measure it. After the measurement, you ensure that the teams consistently spending less than 50% of their time on development work get redirected or get dissolved.

Niall: I’m not aware of the 50% division in comparable jobs in industry.

Ben: Indeed; in terms of differentiation between SRE in Google and other, notionally “SRE” jobs, this, would be one of the easy ways to figure out what you are getting into. So when you interview with other groups, and talk to the folks in the team who you prospectively may be joining, try to find out how many lines of code they have written in the recent past, and what fraction of their working hours is spent on writing code. If their answer is “I wrote three functions last month,” well, you have your answer. Another question you could ask is, who are the more senior developers that they’re working with either inside the SRE team or outside the SRE team? How often do they end up getting promoted? Is their promotion rate the same as the SWE team?

In Google SRE, we pay close attention to the promotion rates by level for everybody irrespective of systems or software background, and compare that to the overall Eng and Eng SWE promotion rates to make sure that they are statistically identical. And they are.

http://www.site-reliability-engineering.info/2014/04/what-is-site-reliability-engineering.html
the overall Eng and Eng Software Engineering promotion rates to make sure that they are statistically identical. And they are.

Niall: Can you talk about some more classic conflicts in operational and development groups?

Ben: One classic conflict in the industry is that the ops team comes up with long checklists that the development team must go through before they say something is supported. This is an important conflict, because it's one of those places where you see clearly that the incentives of the two groups are actually different. When you strip away everything else, the incentive of the development team is to get features launched and to get users to adopt the product. That's it! And if you strip away everything else, the incentives of a team with operational duties is to ensure that the thing doesn't blow up on their watch. So these two would certainly seem to be in tension. How do you resolve that?

The solution that we have in SRE -- and it's worked extremely well -- is an error budget. An error budget stems from this basic observation: 100% is the wrong reliability target for basically everything. Perhaps a pacemaker is a good exception! But, in general, for any software service or system you can think of, 100% is not the right reliability target because no user can tell the difference between a system being 100% available and, let's say, 99.999% available. Because typically there are so many other things that sit in between the user and the software service that you're running that the marginal difference is lost in the noise of everything else that can go wrong.

If 100% is the wrong reliability target for a system, what, then, is the right reliability target for the system? I propose that's a product question. It's not a technical question at all. It's a question of what will the users be happy with, given how much they're paying, whether it's direct or indirect, and what their alternatives are. The business or the product must establish what the availability target is for the system. Once you've done that, one minus the availability target is what we call the error budget; if it's 99.99% available, that means that it's 0.01% unavailable. Now we are allowed to have .01% unavailability and this is a budget. We can spend it on anything we want, as long as we don't overspend it.

So what do we want to spend the error budget on? The development team wants to launch features and get users. So ideally, we would spend all of our unavailability budget taking risks with things we launch in order to get them launched quickly. This basic premise describes the whole model. As soon as you conceptualize SRE activities in this way, then you say, oh, okay, so having things that do phased rollout or 1% experiments, all these are ways of putting less of our unavailability budget at risk, so that we can take more chances with our launches, so that we can launch more quickly. Sounds good.

This approach also has another good consequence, which is that if the service natively sits there and throws errors, you know, .01% of the time, you're blowing your entire unavailability budget on something that gets you nothing. So you have an incentive in both the development world and the SRE team to improve the service's native stability so that you'll have budget left to spend on things you do want, like feature launches.

The other crucial advantage of this is that SRE no longer has to apply any judgment about what the development team is doing. SRE measures and enforces, but we do not assess or judge. Our take is "As long as your availability as we measure it is above your Service Level Objective (SLO), you're clearly doing a good job. You're making accurate decisions about how risky something is, how many experiments you should..."
run, and so on. So knock yourselves out and launch whatever you want. We're not
going to interfere." And this continues until you blow the budget.

Once you've blown the budget, we don't know how well you're testing. There can be a
huge information asymmetry between the development team and the SRE team about
features, how risky they are, how much testing went into them, who the engineers were,
and so on. We don't generally know, and it's not going to be fruitful for us to guess. So
we're not even going to try. The only sure way that we can bring the availability level
back up is to stop all launches until you have earned back that unavailability. So that's
what we do. Of course, such an occurrence happens very rarely, but it does happen.
We simply freeze launches, other than P0 bug fixes -- things that by themselves
represent improved availability.

This has two nice effects. One, SRE isn't in this game of second-guessing what the dev
team is doing. So there's no need for an adversarial relationship or information hiding or
anything else. That's what you want.

The other thing, which is one I hadn't anticipated but turns out to be really important, is,
once the development team figures out that this is how the game works, they self-policing.
Often, for the kind of systems we run at Google, it's not one development team; it's a
bunch of small development teams working on different features. If you think about it
from the perspective of the individual developer, such a person may not want a poorly
tested feature to blow the error budget and block the cool launch that's coming out a
week later. The developer is incentivized to find the overall manager of the teams to
make sure that more testing is done before the launch. Generally there's much less
information asymmetry inside the development team than there is between the
development and SRE teams, so they are best equipped to have that conversation.

Niall: The moral authority for SRE to say no is currently well established in
Google.

Ben: And that is crucially important.

Niall: How would an external organization spinning up an SRE function help it
acquire this authority?

Ben: The moral authority is a physics question. It is crucial that you establish what the
target SLO is upfront, because that is the standard against which you are agreeing that
the service will be measured. SRE is at that point simply measuring and enforcing
something we've already agreed we want. Very handy position to be in.

In terms of how you actually generate moral authority, it's easy. You, the development
team, have already told us what the SLO for this service must be, and now we're below it.
There's nothing else we can do; this is a physics problem. You can't change your
mind now. We already decided that this figure is what's in the users' best interest --
currently, the data clearly indicates that we're below that. So we just have to wait until
we get back to that availability level. You can use some of this waiting time to make
sure that your next release doesn't blow it again.

Niall: Let's talk about the lifecycle of an SRE team. How does the engagement
change during that lifecycle?

Ben: The way this is commonly done today is via a capability maturity model. There
are many, and they all mostly say the same things. They all start out with the basic
question of, do you know what you are doing as a team? Or do you just have a
collection of individuals, each of whom knows some fraction of the problem space? Most
teams start out at that point; you have a bunch of people, who each knows some stuff,
and when you need to do something, you try to get people with enough combined
expertise to be able to accomplish what you need to.

In this way of doing things, when something goes wrong with the service, the outcome
is dependent on who the people are. That would be what we call a chaotic situation. As
the team matures, you move from chaotic to defined; i.e., you start saying "Here are the
standard things we do, and they are also documented so that anybody on the team can
do them." You no longer depend on the random selection of the individual who happens
to be there. From there you can move to optimizing things, where you're actually
measuring them. Now that you've said, "This is what should happen," you look at what
actually happened and you compare it to what should have happened. Then you either
change the instructions or change how the people are behaving or both, and repeat
forever or until done.

One of the things we measure in the quarterly service reviews (discussed earlier), is
what the environment of the SREs is like. Regardless of what they say, how happy they
are, whether they like their development counterparts and so on, the key thing is to
actually measure where their time is going. This is important for two reasons. One,
because you want to detect as soon as possible when teams have gotten to the point
where they're spending most of their time on operations work. You have to stop it at that
point and correct it, because every Google service is growing, and, typically, they are all
growing faster than the head count is growing. So anything that scales headcount
The second is, again, this is an engineering team. If you look at the people on the team, their careers and their goals are not furthered by running around closing tickets or provisioning resources. They have engineering skills and expertise that they and their manager want to develop. And that is developed, because they’re software engineers, by writing software and working with people who are more senior and experienced and capable than they are, and learning. So you must ensure that they’re actually doing those things.

Niall: Let’s discuss monitoring, a core SRE responsibility. Can you talk about the philosophy behind SRE and monitoring?

Ben: A classic way of doing monitoring is, you have something that’s watching a value or a condition or whatever, and when it sees something interesting, spits out an email. This is the most common monitoring I know. But email is not the right approach for this; if you are requiring a human to read the email and decide whether something needs to be done, you are making a mistake. The answer should be, a human never interprets anything, in the alerting domain. Interpretation is done by the software we write. We just get notified when we need to take action.

So there are, in my view, only three kinds of valid monitoring output. There are alerts, which say a human must take action right now. Something that is happening or about to happen, that a human needs to take action immediately to improve the situation.

The second category is tickets. A human needs to take action, but not immediately. You have maybe hours, typically, days, but some human action is required.

The third category is logging. No one ever needs to look at this information, but it is available for diagnostic or forensic purposes. The expectation is that no one reads it.

I have never seen monitoring output that does not fall into one of those three categories. I have very frequently seen people make mistakes in implementing monitoring so that they generate logs but treat them as tickets. That’s a big mistake. One of the places you normally spot it is: “Oh, yeah, we had to spend a whole bunch of time reviewing all of this stuff that our monitoring system spits out.” Well, don’t do that. That doesn’t scale as you have more users and more instances, the quantity of that stuff will increase and the quality will decrease. You need to develop a system, whether it’s monitoring configs or a parser or whatever, you need to write a system that will turn that output into one of the three categories.

Niall: So monitoring helps you with noticing when things have failed, and also helps you fix it quicker?

Ben: Yep. More generally, when we talk about overall system availability, there are two basic components to it. There is mean time between failure — how often does the thing stop working. And then there is mean time to repair — once it stops working, how long does it take until you fix it?

Some function of those two is your availability. Falling out of that, there are two ways to make a highly available system. (Of course, anywhere between these extremes is also ok, if the numbers stack up.) You can make it fail very rarely, or you are able to fix it really quickly when it does fail. Google has a well-deserved reputation for extremely high availability. And the way SRE gets that is by doing both.

At the first level — and this is where we spend most of our time — we build systems that will tolerate failure. We talk about that in terms of graceful degradation, as well as defense in depth. They are two different variants of the same thing. Things will fail. What's important is that the user experience is not meaningfully degraded when things fail, giving you enough time to fix them without actually having a user-visible problem. So the MTTR is milliseconds for most failures, because it's automated. Typically, no human will respond in less than two minutes to something that goes wrong. So if you want things that are going to fail without a user impact, the best way to get them is to have them automatically fixed. We do that by defense in depth.

All the different layers of the system are designed to tolerate point failures, even data

http://www.site-reliability-engineering.info/2014/04/what-is-site-reliability-engineering.html
center-sized point failures, without the user experience being affected. All that happens automatically. No human lifts a finger and no human often even needs to know about it. It just occurs. That's defense in depth.

Graceful degradation is the ability to tolerate failures without having complete collapse. For example, if a user's network is running slowly, the Hangout video system will reduce the video resolution and preserve the audio. For Gmail, a slow network might mean that big attachments won't load, but users can still read their email. All these are automated responses that give you high availability without a human having to do anything.

Niall: What about the cases when a human has to do something?

Ben: Once a human actually has to do something, then MTTR matters a lot. In particular, the "R" part of that means not that a human has gotten the page or that the human has triaged the page, or even that the human has gotten to a keyboard to do something. It is that the human correctly assesses the situation and takes the appropriate corrective actions, versus diagnosing incorrectly or taking ineffective steps.

In other industries, you have operational readiness drills; we have operational readiness drills, and that is how we ensure that people know how to respond correctly to a variety of emergency conditions. However, if you get a group of software engineers together and say, "We're going to do operational readiness drills," the nictating membrane will slide down over their eyes, and that will effectively be the end of the conversation, whether you know it or not. One possible way to address that is to take inspiration from role-playing games. Plenty of people in software have played role-playing games at one point or another, and the rest have generally at least heard of them.

So while nobody wants to do operational readiness drills, everybody is up for a game of Wheel of Misfortune. In this context, Wheel of Misfortune is nothing more than a statistically adjusted selection mechanism for picking a disaster, followed by role-playing, in which one person plays the part of the dungeon master -- in this case, the "system" -- and the other person plays the part of the on-call engineer. The documentation folks listen in, we record what happened in the scenario, what the on-call engineer said to do, and we compare this against what they actually should have done. Then afterwards we go adjust our playbooks -- our term for operations manuals -- to provide additional information or context for what the ideal responses would have been.

Part of what makes SRE work in the operations world is you drill people on the correct response to emergency situations until they don't have to think about it. And we do it in a way that's culturally compatible: if you've seen SRE groups do this, people actually look forward to these exercises, because it's an opportunity to kind of show off what you know and it's fun.

Niall: We talked about some of the responsibilities of SRE, but not all. What about capacity planning?

Ben: Demand forecasting and capacity planning can be viewed as ensuring that you have sufficient defense in depth for projected future demand. There's nothing particularly special about that, except most companies don't seem to do it.

Another differentiator question between Google SRE and other places to work would be, so what N+M do you run your services at? A common answer is some variant of, "I don't know, because we've never assessed what the capacity of our service is."

They may not say it exactly that way. But if they can't tell you how they benchmark their service, and how they measure its response to 100% or 130% of that load, and how much spare capacity they have at peak demand time, then they don't know. In the absence of demand forecasting or capacity planning, you can expect frequent outages and lots of emergencies.

Niall: In the traditional model, IT is a cost center. They are basically about preventing loss and because they are perceived as a cost center, there is no particular incentive to change much. In the SRE model, we are empowered to improve efficiency, extend capacity, and contribute to the more efficient operation of the product. In this model, what are the incentives for senior decision-makers?

Ben: So there's two layers of incentives for efficiency. One is that the SRE head count isn't free. A Product Area gets a certain number of head count. They can spend them on developers or they can spend them on SRE. In theory, they will spend only as much on SRE as is necessary to get the optimal feature velocity while meeting their service SLO. In fact, that's actually how many we want them to spend on SRE -- no more, no less. So that's an incentive for them to be frugal with their SREs, and also be careful about the code that their teams write so that it doesn't generate a lot of work that SRE teams need to deal with. Plus, if you build bad code, then all the good SREs leave and you end up either running it yourself or at best having a junior team who's willing to take a gamble.

The second point is that once you realize capacity is critical to availability then you realize that the SRE team must be in charge of capacity planning, which means they also must be in charge of provisioning and change management. Therefore, you get utilization as a function of how your service works and how the provisioning is done. So...
SRE, by definition, must be involved in any work on utilization, because they ultimately control provisioning. So you get a very, very big lever on your total costs by paying close attention to your provisioning strategy.

**Niall:** Has Google SRE suffered as a result of the increased competitive environment? What about the use of the term “SRE” by other organizations?

**Ben:** In general, we've found that when people depart for other organizations, they generally come back. The things that currently distinguish Google SRE from how other companies do things today I would expect over time to be adopted by those companies. It's just a good way to run things.

It does of course require a lot of management support and a reliance on data to make decisions. This has happened a few times when a problem made it up to senior management; in this case, the VP for Technical Infrastructure, and, eventually, the CEO. They always back the SRE team, so people don't even bother trying anymore. You may not get that kind of management support at all companies.

**Niall:** On the topic of differentiation, are you familiar with the term "DevOps"?

**Ben:** I am familiar with it. I started coming across it a few years ago. It appears to describe people who are doing things similar to what SRE does, and it does hit the idea of let's have folks who are developers be on our operations team, which I think is excellent.

**Niall:** The term does not enjoy a uniform definition. It's a good idea to have the development people work closely with operational people. The problem is that it reifies operations, and if you buy into the way SRE does things, that is the wrong vision.

**Ben:** Yes, there appears to be a lot of variance in what “DevOps” means in practice. We've iterated to the current SRE definition over the last 15 years, and key pieces include status parity, free transfer, scarcity, operational load caps, error budgets, and so on. I've seen this definition work very well in practice here at Google, and I expect we'll continue to evolve it to make the role even more attractive to developers while at the same time making it more effective at running efficient, high-availability, large scale systems.

"We've iterated to the current SRE definition over the last 15 years...

I expect we'll continue to evolve it to make the role even more attractive to developers while at the same time making it more effective at running efficient, high-availability, large scale systems."

---Ben Treynor, Google VP Engineering

Posted by Niall Murphy at 10:56 AM