Site Reliability Engineering at Google

Taken from: https://www.youtube.com/watch?v=Cxb7a8lTv8A&ab_channel=GOTOConferences

Speaker

And Bjorn has mentioned a lot of things from the SRE book, specific to monitoring, thanks for the sales pitch. I will talk more about the philosophical background, the idea of site reliability engineering in general.

So site reliability engineering, it's kind of a weird name, so what does this site part come from? Well, essentially, Ben Trainor had a team of seven people running all of production and their mission back in 2003 was to keep the site up, where the site was Google.com.

And if Google.com is down, even back in 2003, that was big news and was very bad for the company. So that's a critical mission and that was their mission statement. Nowadays, site reliability engineering is much broader, there are lots of different services that are run by SRE teams, internal services, internal infrastructure, many different things, so maybe service reliability engineering is what you might like to think about it.

But then there's the second word in there, reliability. Well, I guess everyone agrees that reliability is a nice property, but is it really that important? I mean, as a developer, you can probably come up with a huge list of properties and things that you want to see in your software, and reliability is one of them, sure, but is it top priority, well, you could argue about that.

But let me prove it to you. So you probably all know a service from Google called Gmail. It's a web map thing. You probably have at least seen or maybe even used it. And if you look at that Gmail 2016, it has tons of nice features.

It sorts stuff into social and promotions automatically. It has this very powerful search and lots of things under the hood that you don't want to miss. Now imagine I would strip all of these nice features away and would give you a five-year-old version of Gmail.

That would be really disappointing. You get used to all of these nice features. You don't want to lose them. But what if the alternative to a Gmail 2011 or so would be that of a product that I call? Gmail 500.

Would you prefer that? Because it does not matter which features you have, which features you offer, if they're not available, if your users can't use them. So it's better to lose any kind of feature if you can make your system available and reliable.

So that is the top feature that you want. But the problem here is reliability is a bit like oxygen. It's all around us, and yes, everyone knows we need it to survive. But because it's so ubiquitous, it's easy to take for granted.

You quickly forget about it. And the problem is, when you notice that it's gone, it's kind of too late. And you are in a very dangerous situation then. And it probably will take you a long time to work yourself out of that situation again.

So when a system systematically fails, there's probably not just one thing that went wrong, not two things, probably three, four, five things in parallel that went wrong, one after another. And then they caused this perfect storm that took down your whole website or something.

Now, to get it up and running, often it's not sufficient to just fix one of those five things. Often you have to fix all of them. And until then, everything is on fire. So the problem here is you have to plan ahead to make sure that you stay reliable and not...

But it's not something where you can just react and fix things when they break. Now the problem here is, as I said earlier, as a typical developer, you have so many things on your plate. And reliability is one of them.

But there are other things that people pressure you to do, things to implement, things to take care of. You have to launch. There's a deadline. That's the monitoring later, right? We can add that. People will let us know when the service is done.

So the real important thing here is that you have someone in your organization who is dedicated to reliability, whose sole purpose and mission is to make things reliable and to keep them reliable. And that person can speak up, that team can speak up and say, look, We need to take care of this, and we will take care of this.

We own this topic. And that is why Site Reliability Engineering inside of Google is its own organization up to the SVP level. And that makes us very autonomous, and it gives us the voice to speak for reliability.

Now, what do we do? This talk is about Site Reliability Engineering at Google, but there are lots of other companies that do Site Reliability Engineering. Some of them do it slightly different. You can read a lot about it in the book that was mentioned multiple times.

And one thing that is very typical for SRE at Google is that we work at a large scale. Systems at Google tend to be pretty huge. So that gives us a lot of opportunity, but it's also a huge challenge.

Because as our system grow from a million users to 10 million users to 100 million users to a billion users, you cannot just increase the size of the operations team to keep that running. You cannot increase the size of that team a thousandfold.

You don't want to pay for that many people, and also your organization wouldn't scale that way. That said, even if you run a much smaller organization, it's very important to look out for these things early on.

And they can pay off a lot, especially when you do become successful. Because then often you come into a success disaster situation where it's like you set up yourself for failure. The moment all the users come and love your product and want to use it, it fails.

And you get the bad press and the bad press. worst moment. Now, Site Reliability Engineering has reliability in the name, but that is not the only thing we do. It's also about efficiency. If you run a lot of service, saving a few servers can save a lot of money.

And we work closely with the development teams on how to build new systems and make them reliable from the start, make them scalable, make them efficient, make them fast and make that a very painless process for the development teams because they want to focus on the features.

They want to build this crazy new thing and they don't want to bother with the infrastructure so much. And I guess the key to Site Reliability Engineering is that is an organization that was designed by software engineers.

So we treat operations as a software engineering problem and that fundamentally changes the perspective on how we treat those things. But I said, okay, Site Reliability Engineering runs operations, so it's an operations team.

And if you have been in the business for a while, you probably have learned that dev teams and ops teams, they kind of always fight. You can trace this back to the sex these two, probably before the times where we're computers.

Like probably the people who designed pyramids and the people who built pyramids, they were always fighting, I guess. Now, where does this come from? Well, they're fundamentally different incentives.

A development team, I mean, if you're a developer, and I've been a developer myself, you have this awesome idea how to build something new that will change the world, make the users happy, and make this planet a better place.

And you want to get this thing out of the door. And you want it to be successful. On the other hand, the ops team, well, they want to assist them that they're responsible for to blow up. Not on their watch, not on anyone else's watch for that matter, but definitely not on their watch.

So you don't want to get paged in the middle of the night. That's like a terrible situation. Nobody likes that. And do you know what the number one reason of breakages is? It's change. And change the sex sex.

exactly what developers want. And change is exactly what ops teams want to prevent. Yeah. So you have to find some middle ground, right? So traditionally what happens is the ops team says, yeah, sure, launch this thing, but first let's have a look at it.

Do a launch review. You have this cool new thing. You will present it to us. We will do a review. We want design documentation. And well, there were a lot of things that went wrong in the past. So we created this checklist here.

And let's go through that checklist and make sure that you don't do any of those things that have gone wrong in the past in this company or any other company we've ever heard of. And it will keep adding things to that checklist.

Until this checklist is a huge spreadsheet and you need a project manager to like, administrate the process. And it's very painful. And then you're like, it isn't enough really. You have to look at the system itself, like the white box approach.

You have to do technical deep dives and stuff like this. And it really slows down the dev team. And when they think they're ready to launch an awesome system, you ask them to do this, this, this, this, and this, and this, and this, which is probably another six to 12 months of work.

It's basically like a post submit code review. Everyone hates those. But developers are smart people, right? So they have their counter tactics. It's not really a launch. It's just like a maintenance release, small risk UI change, flag flip, 20% experiment, right?

We don't need a review for that. And now the problem is they will always outsmart the ops team because they have built the system. No matter what clever engineers you have on the ops team, they are always at a disadvantage.

They cannot never know the code base as well as the people who had just written it. So the team which knows least about the code is the one who will suffer if things go south. Now, you know where you have to have any crumpy susceptance.

Is that inevitable? No. So this kind of traditional very tight launch checking does not work. So at Google Site Reliability Engineering typically does not try to do that. Try not to avoid all outages in the first place or even tell the dev team what to do, what to release.

That is the responsibility of the dev team. Now, you might wonder how it is that Google services are still available. There is a very simple rule. But to explain that rule, we first need to explain error budgets.

To explain error budgets, we must think about SLO, service level objectives. So you have certain metrics, say how available your service is. or how fast your service is, what the latency is, you measure them, you need monitoring for that.

And that is why monitoring is at the lowest layer of that pyramid that we saw in the last talk. And then you have a bunch of thresholds where you say, okay, this service needs to provide two nines of availability.

You measure that, you check the threshold, and if you are not within the threshold, you're out of SLO. You haven't met your service level objective. And you put all of those SLOs into a contract called a service level agreement, and if that is not met, typically something like your customers get your money back or so.

So it's very important to establish that culture of SLAs and SLOs in the first place, and to have everyone in the room agree that this is the metric by which we are judged. And money trading hands is typically very good consensus building.

Now, what is the right SLA for a product? Obviously, 100%, right? Things should not break. That is true for certain kinds of products. So if I ever need a pacemaker, I certainly do not want something with 99.5% availability.

Also, if you heard the keynote this morning, if you're sending a two billion spaceship to Mars, you better get it right the first time. Because even if they give you two billion dollars again, it will probably take you another five years or so.

So those are cases where lives are at stake where you deal. with a once-in-a-lifetime chance where you might want to get 100%. But that comes at the cost and the question is, is that cost justified? And if you look at devices like this one or like this one, and those are the things that we typically use to access such services, they don't come at 100% reliability themselves.

If I pull out my phone or my pocket, switch it on and access a random web page, chances that that request will reach that web page are what? 99 out of 100 times, probably. In other cases maybe something is wrong with the phone, something is wrong.

with the cell signal, with the ISP, with something, something happened and your service didn't even see the request coming in. And so if one out of 100 requests will fail anyway, it doesn't really bother any user if one out of a thousand requests fails because of you, because your service wasn't available at that time.

That is just background noise that no one will even notice. So of course you can increase it to four nines to five nines, but the cost is exponential. It's getting harder and harder to make systems that are available, both from the engineering time and from the resources that you need to spend.

And essentially at the end of the day it buys you nothing because the users don't notice. They have other problems. So the question is, what is the right goal of an SLA for your product? And I can't give you a general rule of thumb that is like up to the product management of your system, what is the right percentage that your users are happy.

And typically what SRE does is we look, when we got alerted because we were out of SLO but our users were happy, then probably if that happens a lot, the SLO is too tight. The other way around, the SLO is too loose.

So it is a bit of an iterative process to find the right, strike the right balance. Coming back to the error budget. Now imagine you have a service that serves, say, a billion queries per month. and you have set your SLA to 99.9% availability.

That means you can serve a million errors per month and no one will have any problem with that. So that's a budget that you can use to serve errors. But how do you use that budget? Well the one thing you can do is you just built a shitty infrastructure that breaks all the time.

Your service, they take a few queries down with them, your error budget trickles down over the month and 80% is just spent on, is just leakage of your infrastructure. Or you could say the infrastructure is fine, everything is reliable.

We can take risks by doing changes, by trying something new, of which we don't 100% know if it will work, but it will allow us to innovate faster. Now, the rule that I mentioned is as simple as that.

If your service is within SLL, the dev team can launch whatever they want. Knock yourself out. If it's not within SLA, nothing goes out. Because the SRE team cannot tell, was it this feature, or that feature, or that feature from last month that was causing some slowly building up problem?

Only the dev team can find out. Well, the SRE team can help them with that, but the dev team needs to fix that. So while your service is out of SLA, it's kind of an all hands on deck situation, where the dev team needs to bring the service back into SLA, maybe rollback releases, maybe do cherry picks, do something like that, together with the SRE team, with the sole purpose on fixing things, and not adding more features to the mix that might break other things.

And the beauty of that rule is, it removes that discussion. Nobody needs to argue, is this a risky feature or not? Can we do that? Is it really that important? It's just, we look at the dashboard, is it green?

Launch away, is it red? You're out of luck. And it's, everyone has agreed on it, SLA, and it's not someone, one site against that. the other, but both sites are working together to keep the system within the SLA.

And the other more subtle thing is a developer team is many different people with many different plans. So it's not like they act as one and they produce one feature after the other, but many sub-teams are working on several projects at a time.

And now if I work on this world-changing feature that will increase our market share so much that the competition will start heavy-drinking and give up and eventually change companies, I don't want your flaky, under-tested feature to go out two weeks before, blow our error budget and I won't get promoted.

So I won't make damn sure. that you only release stuff that is good. And I'm also on the dev team, so that's fine. The dev team is polishing themselves, each other, and they don't have to fight with ops or something because they know it's in their best self-interest.

Okay, so you created a situation where the dev team and the SRE team worked together on building a super reliable service that is meeting just the right SLO that you want. But there are multiple ways on how to do that.

You can invest all the engineering and design stuff that you need to build an architecture that is well-tested, very efficient and scalable and autonomous and everything. But that takes a lot of development time.

Or you could just file hundreds of thousands of tickets that the ops team takes care of and with one heroic act fixes the system. And yes, you can do that. There will always be someone on the ops site who feels very responsible for the service and will be very heroic to keep it running.

Sleepless nights, no weekends, a lot of exhaustion. Just feed the demon machine with blood, tears and sweat. Another thing is heroes don't have a very high life expectation. And while it might be very gratifying in the short term, in the long term, it just leads to burnout.

And the worst thing here is it is very repetitive work. It's just... It's like, go through the error lock, fix this thing, fix that thing. Look at the ticket queue. Oh, my gosh. It's huge. Well, take it from the top.

Site reliability engineers are coders, are engineers. We are bored easily. We don't want that. And our management doesn't ask us to do that. It's not about boring, repetitive work. But then, what does the dev team do?

I mean, for them, it's very tempting. Everything they can throw over the fence to the upsides of things makes their life easier. And that sounds mean and malicious. But actually, if you are in that position, and you're very focused on your project, and you think, you did a good job, you put in a few lock statements, and...

and add a few unit tests, it should be fine, right? And then you launch it, and then you forget it, because you work on the next feature. And you don't actually see how that plays out in practice, because you're not running the system.

And all I see is all there is. I don't see that other stuff, so I forget about that. And there's a number of those incentive problems, but unfortunately there is not this one beautiful, simple rule to fix it, but we have six different fixes that work together.

Number one is do not just make it very cheap for the dev team to shift work over to the ops team. And it works like this. There is one headcount pool that the dev organization gets, and they have to pay SRE out of that pool.

So if you get 10 headcount for this year, and you need seven headcount just to crunch through the ticket queue, that means you have only three people that you could put on the dev team. That doesn't sound good.

So typically what dev teams try to do is they try to minimize the headcount they give to the SRE team based on how much work they need to do on infrastructure to make things reliable. They find the optimal balance themselves because it's in their best interest.

You could say, but wait a minute, that makes the SRE team kind of small, takes headcount away from that. Isn't SRE unhappy about it? No, actually what it does take away is the stupid work. It's the repetitive work that is boring.

And we don't want that. We want to be left with the very hard problems that the dev team cannot just. fixed by having another hour of design discussion and SRE hires people from a very broad spectrum of skills a lot of us have been some kind of system administrator operator at some point many different backgrounds but one thing that everyone has in common is that they can write non-trivial code they are programmers and they want to write code not everything we do is coding a good chunk of it is but we do know how to do it so first of all we speak the

same language with the dev team it's on it's it's a discussion among equals And we also understand what they tell us,

they understand what we tell them, we can explain what we need and we know what we can reasonably ask them and what is like a software system cannot do or cannot do easily. And the other thing is that thing with that repetitive ops work.

If you have a coder and you give them one ticket, they say, yeah, fine, I'll do that. Second ticket, they say, I think I know how this works. Third ticket, they're like, oh, gosh. Fourth ticket is I'm going to replace myself with a small shell script.

And that's exactly what we want because what I mentioned earlier was we want to scale the systems and this kind of repetitive work that typically is based on the traffic that the system sees. The more requests you get, the more tickets you produce, you cannot scale that.

But you don't want to scale the number of people on the team, so you have to automate yourself out of the job. And that's what we do all day long. The systems keep growing, so we're still in business.

The other thing is if you want to be able to do that, if you want to have time to improve the systems, to write automation, to do all of these things, you need to reserve time for that. So there's a very simple rule that you do only a maximum of 50% of your time on the team with what we call toil.

It's like stupid, repetitive ops work that can be easily automated. It's very hard to define precisely, but they say it's the stuff that you know when you see it. The problem here is if your system is successful and it scales and you get a lot of traffic then your system dies, I said this before and now the automation helps you to keep grow but you need to reserve the time because otherwise you will be in that downward spiral and things will only get worse.

Now, the fourth approach is to keep the development team in the on-call responsibility. They will get tickets, they will get paged, typically only for a small fraction of things. Every team has a different approach on how to do that and a certain fraction goes to them, a certain type of problem goes to them.

The SRE team escalates certain things to the dev team manually but the dev team has to be exposed to running the system to fight that all I see is all there is problem. And it helps them to learn so much more about the system.

So you might see a lot of resistance from the dev teams to do that. Because they don't want to carry a pager. They don't want to be paged at 3 a.m. either. But be persistent. Escalate this all the way up the management chain because it's super important.

And typically when they have done that for a few months, they will understand and they will be very happy and no one will ever go back. Because they learn so much more about the things they have done in the coding part and the design part and how they play out in practice and how to do it better next time.

And another thing is that on the dev team, if one of the developers has spent the last three nights trying to keep the system running because something, someone else, like pushed back and back in their backlog, fix it later, I have more interesting stuff to do.

Well, they will build consensus on bug prioritization. Now, as I said, we have a 50% cap on the ops work that the SRE team does. In practice, it is a cap. It's a maximum. We don't do 50%. Most teams do, I don't know, 20, 30%.

It really depends on the situation, can change over time. But there is this cap. So where does the access work go? I mean, we can't just like leave it there and let the system break down. Well, we can, but that's not what we want.

So that also goes to the dev team. So if you design a system where you don't spend enough time. on making it reliable and you're giving the SRE team a hard time, well, it will get back to you. You will have to chime in and help getting the system fixed and that will help you to understand the effects of what you've done and you will probably try to avoid to do that next time and also it will keep you in the dev team from writing more features that can potentially further break the system,

another self-regulating system, isn't that beautiful? Last but not least, nobody keeps me on my team. If I decide this is a terrible service and I don't want to do this stupid work anymore, I can go.

I find any other SRE team or... Because I'm a developer, I'm a software engineer, I'm qualified for software engineering position at Google, I can switch over to the dev team, to some dev team, probably not the one that built this terrible service, some other dev team.

Maybe to that dev team because I think I can help them to do a better job, I don't know, but they will slowly lose their SREs if they're not providing a good working environment for them, and eventually it might happen that no one wants to work on their team anymore.

Then what? Dev team is like, what happens to our service? Well, it's easy as that, it's the pager, it's the ticket queue, go deal with it. Now that is a nuclear option, you don't typically want to do that.

And it doesn't really happen that often. We make sure that dev teams and SRE teams get along very well, and typically works out quite well. But it does happen, and this threat has to be real. You must make sure that your SVP, your CEO is backing you on this.

They must understand that they cannot keep other people doing a job that they don't want, just because they don't want to fix their system. And then if they understand that, it typically will not happen.

Okay, third part. So we've built a system that is designed to be reliable, and the dev team and the SRE team are working close together to keep improving the system, but there will be outages, that is certain.

We did specifically not aim for a hundred percent reliability Because we don't want to It doesn't matter to the user and it will cost us a lot. It will slow us down It will keep us from writing cool features competition heavy-drinking, you know the drill so There will be outages and it's not fun a little bit maybe but Really if you get paged at 5 a.m.

In the morning and You know your customers have a bad time Nobody is really happy about that, but it's okay Now what we want is we want to get the system back on track as quickly as possible and The first thing we can do is we minimize the impact so Your system has a certain time between failures and TTF mean time to failure.

And then it has an MTTR, mean time to repair. How long does it take you to get it back up and running? And of course we can increase the mean time to failure to make the system more reliable, but it costs us.

But we can also reduce the mean time to repair. How long does it take us to get it up and running again? And one thing that we do is the SRE team is directly in the loop. There is no no cooperation center whatsoever.

The SRE team is in charge of the service. And we have very good diagnostic information. If you get a general problem with your service down message, that's not very helpful. You have to figure out what is actually wrong with the service, where does it come from.

So if you have good white box monitoring and you've been told that you have a latency spike and RPC is going from this cell to this back-end cell, you have somewhere to start. You can look at the network between those two clusters.

You can look at if any releases were rolled out in one or the other cluster, configuration changes, things like that. Quote out problems. So that helps you to get started very, very quickly. And a third thing is practicing.

So if you have a service that is very stable and you make sure that it's improving a lot, you might not have had an outage yesterday or this week. So you might say, oh, I just need to read up on that.

I have no idea what that part of the system is. It didn't have a problem in the last two years. So what you should do is you should practice up front and study have shown that this decreases the time of an outage by something between 50 and 70 percent.

But practicing doesn't really sound like fun. How can we make it fun? The approach we have at Google is called a wheel of misfortune. And it's basically like a role playing game. Dungeons and Dragons, if you remember that.

So basically you come up with a list of things that typically break in your system. You can make a pie chart like that. They won't allow me to show you what actually goes wrong. So we replace that with things you can imagine that would be terrible for your system.

Sharks with lasers, shark NATO, shark avalanche. I have a popular bear cavalry. Don't want that to happen to your service. And then you pick one of those, you can roll a D20 for your D&D

player, and one person on the team is the game master, runs the scenario, says, you get paged, the message says this and that, went wrong.

And then the victim, the players, tried to figure out how that happened. They say, okay, I will look at that dashboard, what do I see? And so on and so forth, and tried to find how to fix the problem and what has actually caused it.

Because that is a very important part, we do not want to just get the system up and running again. That's our very first priority. But after the system is patched somehow, duct tape, but it's running.

We want to figure out what was the root cause. And if it was a bad outage, we want to write a postmortem, and that postmortem should document what happened and why. But writing such a document takes a significant amount of time.

Doing all the investigation for the root cause takes time. So you cannot do an arbitrary amount of these all the time. So we have to rule that you should not have more than a median of two events per on-call shift.

You should, typically, what was going on? Magic. You should not typically have more of these events because then you would either not be able to really do a deep investigation or doing other things. like project work.

So that is why we typically have SRE with at least eight people on an on-call rotation. We do prefer having two sites on different sites of the planet so you can have a follow the sun rotation. You're not getting paged at 3 a.m.

because the person on the other site of the planet is responsible and have at least six people so you have six weeks to do project work and so five weeks to do project work or less four weeks of project work and one or two weeks of being on call as primary or secondary and then you have enough time to actually write a good postmortem.

So what is a postmortem about? Postmortem documents, what has happened, what we found out and what we should do about them. So it has a list of action items and don't let those action items rot in that document.

Put them in your bug tracking system and make sure they get prioritized. But the most important part about postmoderns is that they're blameless. One of the many companies that I worked at, whenever something was wrong and there was afterwards a discussion, the first and only question was, who's to blame?

And you know who was always to blame? The person who was not in the room at the time to participate in the discussion. So you better not take a vacation. That is not what you want, obviously. What you want is that people come forward and tell you what they did, how it went wrong and not getting blamed for it.

So we look at the process and the technologies. So if Kristoff pressed that shiny red button that took down the system, the question is not, why is Kristoff so stupid? The question is, why did we have the button in the first place?

And what can we do about it? And then we prevent that same thing from happening again. So hopefully you will not get paged for the same problem months and years over and over again. You will get new interesting outages that are fun to debug and will make a good wheel of misfortune.

So how are we doing on time? Okay. So you hire only coders. You let them do whatever they want to improve the reliability of the system. You have to have an SLA and everyone has to agree that this is the right SLA and that's the most important thing.

Use error budgets to gate your launches. You have a common stepping pool. Five percent of your ops work. Go to the dev team. The operational load on the SOA team is kept by 50%. You have to have enough people on your own co-rotation so they can actually do deep investigation without being exhausted.

You practice, practice, practice. And your post volumes are blameless. Now shameless block, the blue book, again. Get a copy of it, or if you don't want to spend the money, there is actually a copy of it online as creativecommons at google.com.sre, so you can have a look there and then get your good old paper copy.

Thank you very much. Thanks. I bought a book, by the way. It has too many pages, I think, so I have 800 or something. five minutes I must admit I haven't read all of them okay but most okay we have one minute for questions or actually for yeah let's try a few questions does delaying releases increase the chance of errors for subsequent releases as these tend to be larger you should release as often as possible so the smaller release gets the more reliable it is and the easier it is to debug because the delta smaller so if you think back at the 90s where you ship CD-ROMs like you had a dev team to change everything in the system over 12 months so basically anything that you had tested for the last release has become worthless if you had only like an eight hour eight hour work day for them to change and break things it's much easier to find out what went wrong then again if you are in a situation where you're out of SLR and you're not launching they should not keep changing the system they should fix the system and not just delaying later releases by doing project work on the site you want to keep that window as short as possible to get the system fixed so the dev team can get back to future development but only once the system is fixed is there a conflicting incentive regarding the diversity of technology used by developers for example every new language or platform used in the landscape increases the work of the SRE team so that they sorry increases work of the SRE team more than using something already in place I guess there's a trade-off here of course you don't want to like manage this like crazy sue of things then again the programming language that something is written and doesn't matter that much at the end of the day if you set certain standards And the best thing is to get the SRE team involved very early in the design process so you can agree on the technologies that you use.

And if something is just a better fit for your service and will make the service better, go for it. If it's just because that's the fancy new thing that you always wanted to try out, maybe SRE will not be thrilled.

Then again, reducing the number of critical dependencies that your service has is probably the best you can always ever do for increasing the reliability and also the headache that maintaining that code will give you.

Okay, last question, oops, questions come in so quickly. What are typical SLA budgets on different products? No comment. But, there was a public announcement this week that Cloud Spanner now offers five nines of reliability.

That is a lot. Okay, thank you Christoph. And thank you for coming.