Google Prodcast Season Three Episode Five

[JAVI BELTRAN, "TELEBOT"]

STEVE: Welcome to season three of the "Prodcast," Google's podcast about site reliability, engineering and production software. I'm your host, Steve McGhee. This season, we're going to focus on designing and building software in SRE. Our guests come from a variety of roles, both inside and outside of Google. Happy listening. And remember, hope is not a strategy.

[JAVI BELTRAN, "TELEBOT"]

STEVE: Hello and welcome to the "Prodcast." This is Google's podcast about SRE and reliable software. This season, season three, we're focusing on software engineering and building reliability into your systems directly. We have a couple of guests here that will introduce themselves in just a minute. But first off, I'm Steve McGhee. I'm a reliability advocate in Google Cloud, and my co-host is Jordan. Jordan, why don't you introduce yourself.

JORDAN: Hi, I'm Jordan Greenberg. I'm a program manager in Google cloud platform security.

STEVE: Thanks so we have today—drum roll—Silvia and Niall. Silvia and Niall, could you please introduce yourselves.

SILVIA: My name is Silvia Botros. I am a senior architect at Twilio currently. My background is heavily rooted in relational databases, which tend to be the worst thing to make reliable. And so I've been working with teams in Twilio for a while now on expanding that trauma, that experience into reliability across the board and not just with relational databases.

STEVE: Cool.

SILVIA: Niall.

NIALL: Very good. My name is Niall Murphy. I've made a long career of failure, sometimes failure downward sideways, sometimes upwards. I come to you from years in cloud providers, including Amazon, Google and Microsoft. And most recently, I am the CEO and co-founder of a teeny tiny startup in the reliability/SRE/infrastructure space. And I'm probably most well known in the outside world as the SRE book instigator.

JORDAN: That is a lofty title to have. The SRE book instigator.

STEVE: Well done.

JORDAN: I'm sure that somebody that works on this specific podcast might have something to say to you. We'll make sure to carry that info over.

STEVE: Is that a threat, Jordan? What is going on?

SILVIA: It's like, oh, so it's you.

STEVE: Excellent. The thing that comes to mind when I hear Niall talk about like the olden days when the book is the meme from I think it was the first "Lord of the Rings" movie where he says, I was there Gandalf, , years ago when the war was first blah, blah, blah. I was able to use that recently. And I think it's my new favorite meme, so it makes me feel old, and I kind of like it actually. But I'm not quite as old as an elf. But anyway, why don't we get into it.

So reliability is often described as making the thing under your application just up more. But I think there's more to it than that. So I'm hoping today we can talk about the application itself, and of course, the application is not just the front end and the middleware. It's also the database. So this is where Silvia will teach us some things, but it's not just like get VMs with more nines, please. There's probably more stuff than that, I'm guessing. So I'm wondering if we can get started with those databases because frankly, I know very little about them when it comes to reliability. I've gotten away with being at a place where someone else takes care of that.

SILVIA: Lucky you.

STEVE: Which I highly encourage people to do, if you can pull that off. But like, how can they be bad? Like what is unreliable about databases? Weren't they solved at least years ago or is it still imperfect? Was that what's going on?

SILVIA: I mean, like everything in what we do, we keep hoping that it's about the technology, but it ends up being about technology and people. And the people part is always very big. One of the biggest challenges, especially with relational databases and databases in general, actually, not just relational, is for quite a while as a field, we leaned too hard towards what you describe, we're making someone else problem, but where the someone else is not a whole team, it's a one person. One of the first things-- I've done some amount of public blogging. And I'm sure some people here will be able to

find it. But then also it all culminated in co-writing "High Performance MySQL." And one of the very important things that I had to learn in my growth from a senior DBA with all of the context that that title always comes with, where she's running the databases, the schema changes go through her, the performance issues go through her. If we think there's a database problem, we base her on being basically a human single point of failure. I'm personally that little rock at the bottom of the XQCD comic. It was not good. There's a whole culture in the database engineering space that is thankfully now starting to change a little bit, where it was the DBA runs the database. And here the multiple risks there is it's not just the DBA, it's also the database. We no longer build products that way. We're way past the LAMP stack days except for one company that's still living that way. And even that one company is not even one database anymore. So big part of the risk there has been a culture shift before even the technology could come into picture, where you stop thinking about this as like there's this one person who has the magic incantations to figure out exactly what to do. And it's more about you predict failure, and you plan accordingly. And therefore, you don't want a single database failure to take down your whole product. One of the things that I made sure to do when I was given the big task of writing the fourth edition of "High Performance MySQL" after folks that I looked up to and learned from was to change that mindset where older versions of that book were about benchmarking and performance management and squeezing out the last bit of performance of your database and far more about looking at it as an SRE. It's just specific to databases. You assume failure, how to assess the return on investment on the work you're doing in the databases, how to figure out the point at which the performance is acceptable and it's not causing impact, and don't spend more cycles there. Spend them in other things instead.

STEVE: Yeah, nice.

JORDAN: When did you like notice that this cultural shift was happening? Like stop thinking about the witch in the forest that has all the spells, but she's been banished to the forest because magic's not actually allowed in the town. So when did you see this shift from that sort of like, hey, we have a problem with the thing that we don't know about. Can you fix it for us? To this is now within our suite of tools and this is the way we debug it. This is the way we monitor and manage it. When did you see that? How did you see it? How do you support it?

SILVIA: So I wish I could say that it's done. It's still an unevenly distributed change. I know of a number of companies where it's still very much like a small group of humans know how to do this thing, and they're the only ones who do it. Big part of it has been more and more sophisticated managed

database solutions. When you start introducing to the business, hey, you run this thing on that managed database, and you reduce your toil by this much, but the trade-off is now it's a managed service, and you can't just have one person go into the box and do a kernel patch, and now suddenly the IO is three times faster. It forces teams to think about this in a very different way because the shape of your trade-offs becomes very different. And you can't just be like, I'm going to go and do it directly. And then from the other side is, for me personally, I was very much-- like at the beginning of my career in databases-- I was Milton with the red stapler. I'm not going to lie. I spent a good amount of time-- because it feels good when you're up and coming in the field where it's like, oh, I own this. I'm doing this, and therefore, I will do everything in this. It was personally for me like a slew of incidents and a little bit of burnout to recognize that I can do this better if I don't do it the same way all the time. I need to lift boats. I need to educate a lot more rather than just constantly like being called in with my red cape to go fix it when it's already broken in production.

STEVE: So you went from a red stapler to a red cape. I like it.

JORDAN: I like that too.

NIALL: Not all heroes wear staplers.

SILVIA: Never thought of it that way. I like that.

JORDAN: Yes. We're going to mark that timestamp. That's going to be the short.

SILVIA: There you go.

STEVE: Nailed it. So I presume there are websites in the world that are more than a database. And I'm curious if Niall can talk us through-- I know. It's hard to believe-- talk us through how a system that maybe talks to a database might deal with adding reliability to itself, maybe in the way that it talks to that database or other things inside its own logic bubble. What are the things that teams can do directly to their code to make their system more reliable away from the infrastructure but actually in the code itself?

NIALL: Yeah, it's a huge question. And it's a very interesting one because of course, coming back to the thing that you led with at the start of this conversation, we often are primed to regard reliability as a question of interfaces. And so when you disappear into an interface, you're kind of like, OK, somehow the complexity of the world is being pinched into a bubble for me, and something or

someone is handling that stuff. And so it's all going to be fine. Which is, of course, not true. But the way in which it is not true is different to the way it is not true within your application. Everything is a potential source of unreliability. One of the particularly interesting consequences of Spectre and Meltdown, if you remember the controversy-- if that's the right word-- the event around discovering that certain processors had the ability to leak information if they were programmed in a particular way. That brought to the fore-- for a lot of developers-- the key point that, actually I'm not really running on abstract stuff that adds things for me. I'm running on a thing that has a definite structure and it has a consequence, and its design has meaningful consequences. So that was-- coming back to one earlier question-- I think an interesting mindset shift for a bunch of people. But anyway, we're accustomed to thinking of things in terms of interfaces. Actually, it's not quite that simple. Of course, you can have unreliability in the simplest of functions. I do a lot of test-driven development, and I'm always amazed by-- I write a function that I think is pretty simple. I write a test for it, and I'm like, oh yeah, actually, there's no way that could have possibly worked. And I have a lot of reflections upon my own meager talents as a software engineer.

Anyway, long story short, there's a lot of things you can do with respect to reliability within the application and looking at it as a software problem or a software discipline or an engineering questions of methodologies and viewpoints that humans might take with respect to how to write software correctly. They all start to come into the picture now. But it actually ranges from things which might be considered extremely simple. Have you considered checking the return code of your calls?
JORDAN: Wow.

STEVE: Wild.

NIALL: That is a huge-- exactly. That is a huge piece out there, which is even today not necessarily being followed or paid attention to. But it goes all the way up, up the stack to various levels of abstraction. And so you can see within your application, you can bring in frameworks that are trying to use-- or that are trying to make it easier to use stuff like Paxos or really high-level abstractions around reliability. My current area of interest is a middle ground, which is frameworks to make a dressing external and internal-- things on the other end of a TCP connection, essentially, more reliable. And this is one of the things that Stanza Systems-- my current company-- does. But the basic idea is when you are executing a function, the function has some set of preconditions, some set of post conditions, like things you expect to be true after you invoke the function. But depending on the complexity of what you're working on, you probably need to wrap your functions, check return codes, have if-then-elses

for strange failure cases, and so on. But you need to assemble these functions in a way that the programming leaves you with safe results. That's another way to look at it. You are trying to write software so that the output will be safe. You won't change state too much. You won't overwrite user data. You won't mistakenly assume a bunch of things are true and scribble all over storage when in fact they're not true, and so on.

STEVE: Silvia, did you have something to add to that?

SILVIA: I was going to say I really like that approach because when I work with teams, whether it's a database reliability problem or a larger, just like, more ambiguous reliability concern, what I've seen a lot is as engineers, we are, like Niall said, we're primed to think about the code we write as it does thing. And if you flip the script in their head more towards, I give it x. I'm expecting y, and therefore checking for like is it true? It feels like a bit of a mind shift for a lot of folks to just recognize that you may be writing this code. And it's doing these steps, but if the outcome is not what was expected-- and I feel like that is so similar in my mind to how we always tell people write tests first, but it's like at a much higher abstract level than write unit tests and then write the code, which I feel is more effective. Because unit tests are great, but they're still not going to give you the big picture.

STEVE: There's happy path programming and then all the other stuff.

SILVIA: Exactly.

STEVE: But that sounds like a lot of work. But it turns out it's important.

SILVIA: And it's funny because I grew up in my career from operations and into database things. And so I've lived all my life in like the shit is broken in production and then trying to align that mindset with engineers who are like, I write code. Is it going into production? So it's good. And it's like, oh no, but like you wrote code and went in production, and then other things happened. And like you say, the happy path it's a double edged sword, but, like, I try to always come in when I'm working with teams on like a design document or like a plan for how to build the thing. I'm usually-- like my reputation internally at Twilio and it past companies has been like, I'll come in. I'll be like, take this part out. It's more like much more paranoid planning than happy path planning. And this is the two things that are always slightly in tension, but you need them both to get better outcomes.

STEVE: Cool.

JORDAN: It's kind of like packing for a trip, and then you unpack, and then you repack in a different configuration, and eventually you end up with something. But next time you'll know better how to pack your stuff. Maybe next time you'll have a go bag in that way of thinking about what you're putting through the testing for going forward. For both of you, there are many reasons you might introduce these type of changes or improvements. Some could be born out of Niall's favorite failure or other reasons, motivations outside of your control, like do you have a stakeholder telling you to do these things? Is it a technical improvement to implement a change or improvement? How many times is stuff born from an outage versus out of a want to improve something? So open for both of you.
SILVIA: It's definitely a lot more outage.

STEVE: Or how should this work? Like how would you suggest people approach this?

JORDAN: Great question.

SILVIA: So what usually happens is out of outage. My attempts, what I try to make happen is to take it to the next level up where you're not just solving for the outage that happened. See how the outage that happened could happen in another way? Like you landed in this bad spot. It's not the only way to get into this bad spot. You landed in a spot where you realized your backups were not working. There's going to be very different ways, like many, many ways, to figure out that maybe apply some fire drill or backup testing. This is a very simple example. So it tends to be that way. And then there's the whole universe of things that happen from outside of outages where the first one that comes to mind is all of the compliance burdens and the data residency burdens that a lot of us have to deal with, especially as companies grow. So those tend to come far more as either a hypothetical or it's like, audit x tells us to do this thing, and therefore we should do the thing.

NIALL: So I think my answer is it depends, which is a great kind of staff engineer response to-- principal engineer response to more or less everything.

STEVE: Fine. He got us.

JORDAN: We walked right into that one.

SILVIA: We should have had a countdown for, it depends.

STEVE: Well, that's one so far. We can count them, at least.

NIALL: So my particular version of it depends in this case is related to all kinds of things around engineering culture. For example, you can be strongly product management-driven as an overall end

product org, for want of a better word. And you can have loads of features coming down the backlog where you don't have a whole load of choice about where they're coming from. Maybe there's things to respond to. Like, Spectre and Meltdown and crap ton of software engineers around the world were doing things quickly in order to respond to security incidents, and that has not changed in the six years intervening and neither do I expect it to. So there's a load of underlying reasons why this could happen. One interesting one, which I think often isn't discussed, is responses to complexity, because one of the things we end up doing in the reliability space is we stand back from an architecture diagram and we go, actually I don't understand this, and I don't think anyone can. Is there a way to simplify this? So simplification can be a response to certain kinds of reliability problems that flow from complexity. Of course, simplification can be done in a number of ways. One extremely common way in the software industry is to go, eh, a bunch of incomprehensible stuff. Now I will put it behind an API. And now I can forget about it, which has the problems that we've discussed.

SILVIA: The closet nobody wants to open now.

STEVE: Perfect. What could go wrong?

NIALL: The problems we discussed earlier. There's another way, which is traditionally speaking, not really done in the software industry, which is to turn things off or stop running things or deprecate old things, not just upgrade, but actually get rid of them. And obviously, again--

JORDAN: What does that mean?

NIALL: [LAUGHS] The details--

STEVE: Not understandable.

NIALL: The details of how this works, well, they vary a lot. But one of the things I'm noticing in the outside world is that actually cost control implications can run in alignment with simplification initiatives. And you can go-- actually this thing only reproduces % of our income, and it costs % to run goodbye or, like, equivalent.

STEVE: So one question I have is, let's say you have a service which is worth making more reliable. It is not in the %, % category like you just described, but it actually makes us money and somewhat problematic. So, like, we should maybe make it better. What do you think? Given something like that where it's not totally broken but it's also not the best, is there a magic spell that I can use to convince my bosses that this is time worth spending? Essentially like what I'm getting at is like, is reliability

feature work, is there a way to prioritize it or get it done when it comes to the prioritization mechanisms that companies out in the world tend to have? Like, I expect that it is not always obvious that this is the right thing to work on. Is that true or untrue? What do you think?

SILVIA: Casandra comes to mind. Not the database.

JORDAN: Have you asked the witch in the woods for the spell?

SILVIA: I've been called that a few times in my lifetime. But Niall had his hand up first.

JORDAN: Yes. Very good. Very good.

NIALL: So there are two answers to this, the industry standard answer and the answer I have settled on in the past year or so, which I like talking about a lot. The industry standard answer-- actually, I shouldn't say the industry standard, but it is a common theme is OK, what was the cost of the last outages of this type divided by ? Roughly speaking, not addressing this costs us x, addressing it would cost us y when y less than x, then OK. Do it. Which in fact is a rampant simplification of a very large number of typical kind of social interactions, and sometimes they are business related, et cetera. The answer I have settled on most recently, which I think is much more interesting, even though it's not necessarily more correct, is based on a piece of research by Microsoft, which showed in summary, that for a given product-driven backlog, approximately one third of all of the features were positive when implemented, which is to say that they yielded about the amount of cash or additional users or whatever it was supposed to do, it did it. OK, fine. One third were neutral, which meant we can't really detect any change in the metrics here. And one third were net negative, which is people stopped using the product, or it cost more than it yielded and so on and so forth.And so if we were in an evidence-based policy world, we would be able to go to folks in the business and in product, and so on, and say, % of the time your thing will not do what you think it does. Over here I have a fix for your infrastructure. It will do the thing that we think it does. How about we exchange your thing for this thing, which on average will be correct?

JORDAN: Yay.

NIALL: Those conversations--

STEVE: I fail to see the problem. This sounds like a perfect plan. Let's do it.

SILVIA: If only all VPs would listen to that.

JORDAN: I know. It'll work, right?

NIALL: So there's some nuance here about how you would address that. But I think the underlying intuition that actually-- quotes were not very good at figuring out what's going to happen with the software after we have written it, which of course, is true. Software is a complicated thing. The world is a complicated thing, and so on. All of that being true, you would like to say, generally-- not always-- but generally reliability work has the kind of scope that allows it to be interpreted-- its consequences to be interpreted way better than overall feature work. And so you could trade a much wider scope for a narrower scope or whatever. Now of course, then you get the implementations or the reliability work, which is actually build new system x or retrofit Paxos into your data model or screaming horror style stuff like that. But in general, I feel we should be making more petitions towards complexity, simplification and all of the other things I was talking about in the context of the Microsoft research.

SILVIA: I would say my answer to this is, well, first of all, again, I always go back to-- the research is I'll actually remember that paper, and I like it a lot, but it comes down to what are the system you're in. So assuming you have supportive leadership, which is an easy one to just say in a podcast, but it's a very real concern from one company to the other. So assuming you have leadership that truly wants to understand how to assess these trade-offs, one of the things I'd have seen in the last few years that have really improved the conversation about these things-- because ultimately we're trying to quantify squishy things like, if I change this, will it break later? It's we're trying to apply some kind of measurement to things that are basically trying to forecast the future. And one of the things I've really liked using a lot have been DORA metrics, which I suspect is the same. Shout out to Dr. Forsgren because I know that the paper from Microsoft was also her work, I think. I hope I didn't get that wrong. But DORA metrics are one of those things that I know have evolved past the annual assessment that we see every year come out. But at their heart-- and I highly recommend the book, "Accelerate," because it goes much more deep into why these are the metrics that it landed on. It can help leaders quantify things like it's not just that this service makes us money, the team that's on call for it is scared to change it. Like that's a squishy feeling that if you couple with something like change failure rate-- which is one of the Dora metrics-- you start actually seeing data around every time they do a deploy to it, something small breaks, and then every deploys or so something big breaks. And therefore, the team's confidence in making changes to the service that makes us a lot of money is declining. That needs to be addressed. And it becomes-- you start applying slightly better measurements to things that are far more squishy in feeling.

STEVE: Yeah, I think in the past, we've always had, well, every domain is its own special snowflake, so how can we possibly compare them? But I think DORA and "Accelerate" gave us some really good numbers, and they show, like well, yes, they're all special, but they're all stuff running on computers, so we can measure that part of it.

JORDAN: So can you talk about in the lens of building stuff into software to support reliability, what spells has the witch of the woods already given us to help us with that? So example, exponential reach highs, jitter, rate limiting, load shedding stuff. We have some magic here. Can you talk about some of this magic, maybe some of your favorites or an example of when you've used stuff like this to accomplish a reliability goal?

SILVIA: It will never cease to amaze me how many times and how many different ways a company can have an incident because they internally had a DoS dynamic. Like system A was doing this, and then system B just had no clue what the hell is about to happen. And boom, I mean don't let your--

STEVE: Stop hitting yourself kind of thing.

SILVIA: I know. It's just and it's--

JORDAN: The call is coming from inside the house.

SILVIA: Yeah. And the thing that I keep saying to engineers when these happen, it's like it's not that this particular way you do it is going to prevent this from happening. It's like this will happen. You just need to be far more paranoid about it. For example, don't ever let a logging call be synchronous. Writing a log line should not take down your service. Sending a metric should not take down your service. Now, there are going to be certain things that should take down your service. Like, you're about to write a line to a database and the database is not there, that's a valid reason. And then for that one, you start thinking about how fast you can fail over. Do you really need a single writer database? What happens if you switch this to a database that's more availability than consistency-oriented? Like there's other things. But like the way that the same smells keep happening, it still surprises me, to be honest.

NIALL: So I think of this a little bit like using one of the analogies from earlier. If you think of RPC calls as externalized call stacks for one machine software or one computer software, then every time you're calling another function, you're making an RPC, I mean, obviously. But I think people kind of tend to forget that actually there's a call chain which is a software stack, which could and should be regarded as no more complicated and difficult as pulling the registers or heap or whatever for your

particular application. But actually, as a matter of observed reality today, it's just really, really hard. And it's particularly really, really hard in the cases where you are DoSing yourself in the cases where traffic management should be a hugely powerful lever for you to control your call stack but actually isn't because you don't have the tools, et cetera. So we do rate limiting as a service as well as a bunch of other things. And so rate limiting in terms of being able to in real time control and cap and prioritize, in particular, various call stacks over each other is I think actually a much underused capability in software. There's actually a ton of open source things that do bits and pieces of this, but most people don't really think about the DoS cases. Of course, starting off you're like, hey, if I'm DoSed, my thing is popular. Excellent. That's fantastic. And then you get DoSed a couple times in a row, and people are like, I will go to your competitor, and I'll never see you again.

SILVIA: They don't trust you now.

NIALL: They don't trust you now. Also as an SRE nerd, I think we as a profession don't really understand customer trust very well, and we're not sure what causes it to be lost completely and what causes it to be sticky, et cetera. So there's a lot of nuance there. I think we don't understand. But traffic management, rate limiting prioritization, load shedding, particularly the client side load shedding where you can stop actually sending the work rather than the work just arriving at the machine no matter what, and it actually has to deserialize the protobuf and make a bunch of decisions, et cetera, to drop it, which is more work. You actually want to control the client side. All of these are hugely powerful techniques which I think are underutilized.

NIALL: Exactly. And I think one of the crucial differences between folks who are raised in the product management or product developer tradition is that often they do think very fine grainedly about the minutia of the thing that they're trying to deliver. But the holistic picture is often not present. And in fairness, they're not rewarded for the holistic picture necessarily. Now, of course, this picture is complicated and its staff engineer tends to get a bit more horizontal and so on and so forth. But broadly speaking, those kinds of differences exist, which means that folks raised in the reliability tradition often have more understanding of cascading failures, subtleties about load balancing. I remember being delighted when I discovered that a system could return s way quicker than it could return the legitimate data that it was in fact supposed to, which meant that those systems attracted more traffic from the lowest latency load balancer configuration that pertained at the time but not the day after or any day after that. So there's a lot of subtleties in these things.

SILVIA: One thing that just stood out for me in what you said, Niall, it's a good thing that now in this field-- I remember circa seven, eight years ago. I remember I wrote this blog post about being a principal engineer. And like the reaction I got from it was like, wow, that is cool. And I was like, really? Y'all think this is new, like this is how it should be? But one thing--

JORDAN: Nerd.

SILVIA: I know. But here's the thing. Like when someone gets promoted in their IT ladder from senior engineer to staff plus, it's not like the next day they suddenly can now see the whole system. I wish it worked that way. But what happens is like one of two things you're going to end up learning as an engineer, and my hope is that a lot of the folks who are interested to listen to this podcast are looking to learn these skills. One of two ways you're going to learn this, you're either going to live through some really traumatic incidents in production that teaches you how these things can happen, how those cascading effects can occur. Or you go learn from other people's experiences, drama, how they did it, and always keep an eye out on the patterns. And my hope is that more and more folks learn these skills the second way rather than the first way. Because if we all as a field only learn it the first way, then we're also going to be slogging through incidents.

STEVE: That's a great point.

SILVIA: And the hope is you don't.

STEVE: Yeah. So given that, thank you for the perfect lead in to my next question. My question really is, if we have all of these tools that are possibly available to us, should only we get them and use them and be proficient at using them? And we have these systems which we don't really understand perfectly well. Do we have to wait for incidents to happen to be able to know which tools to use where? Or can we look at other people what they've done and maybe make an educated guess? Or is there maybe something else we can do? I'm curious if anyone has seen more formal tools. So I've seen things like TLA+. There's a system called STAMP, not a system, but a methodology called STAMP out of MIT. Are these more formal systems ever helpful in helping you determine how the system might better be controlled in the future? This is like proactive, getting in front of it approaches.

NIALL: Actually, I've used TLA+ myself a bit. And the number one person to talk to about this is Hillel Wayne, who runs a consultancy and does training. And I actually have the benefit of a small amount of training with him. But he also does a newsletter on software and computer things generally, which I

highly, highly recommend. Anyway, plug over, reasoning formally about your system, the set of transactions that can happen to it, the pre and post-conditions, and so on, hugely, hugely valuable in terms of the complexity of the space that you cut off by making sure that your system can't get into undefined or weird states-- or well, maybe making sure, but reducing the probability a lot. I do believe there's serious things. And in fact, it's not a matter of belief. There is serious benefit to doing this kind of stuff. And in terms of your other question, which I'll deal with only briefly, with respect to, is there anything that we could use to help determine where we would spend our reliability dollars or infrastructure or frameworks or however you would think of it? There's a huge tradition in the incident analysis post-incident response space, et cetera that tries to, I suppose, address some of these questions. I am afraid, hand on heart, I'm definitely an it depends person in this context. There's lots of frameworks to use. They're kind of bicycles for the mind style things, typically speaking, but there is a cost benefit to them. And I don't think there's an algorithm to choose these. Anyway, that's where I am.

SILVIA: So I'll take this question back to like, it's not first about the tool. It's first about the people. This might be a shock to this audience because we're all steeped in this, but I still routinely encounter teams where they just don't think they need to spend time sort of preparing for incidents, whereas like they've just accepted that incidents happen to them. So getting folks to just get out of that mindset first is probably like job number zero. Having gotten to that, one number of the things that I recommend to teams that I've seen actually been very useful in the past is try to-- first of all, one of the things that are very up and coming in the field right now and a lot of people are sort of taking on is domain design. You start off like if you are a company that has grown so much that you have multiple products, you cannot continue with those products just like haphazardly flat network talking to each other because now the space of possible failures becomes untenable. Now I understand the risk of what Niall was explaining earlier of you can't just shove things behind an API and call it good. But there's a happy medium. You need to find the Goldilocks where you shove the parts that actually make sense together behind APIs. And so you can separate making those individually reliable and then separately figuring out the cross communication being more reliable. At minimum, there you start gaining things like blast radius mitigation when actually things break. One of my favorite things to do with teams are things like tabletops where once you know what your domain look like, you start talking about, OK, traffic is x. What's going to break first? We don't even know. OK, well, we should know. So things of that nature, like the hypotheticals that you probably will have a hard time testing in reality in production. But you should think about what are the parts that need load shedding more than others?

What are the parts that are like, yeah, this is probably-- Like, you start recognizing this service that is in the middle of every single call hot path is probably x away from having way more failures. What's the ceiling of its performance? And we start talking about this is how you start recognizing things like if we don't start planning for rewriting this thing now, in a year we're going to have a lot of incidents, and by then it'll be too late. Because a lot of times-- and this harps all the way back to customer trust, because by the time the customer walked out the door, it's too late. So you don't want to find out how many incidents it will take to lose half your customers. That's not a number you want to find out. You want to just never get close to that number.

JORDAN: We are unfortunately running out of time. So we have some time for some very quick questions for you. Where can we find you on the internet?

SILVIA: So I used to be fairly active on the formerly known artist, Twitter, but I don't anymore. I suppose at this point my blog is on dbsmasher.com. And a lot of what I explain, if you're interested in it specifically in a database lens, go find "High Performance MySQL" version 4. It was a lot of fun to write with my friend, Jeremy.

STEVE: Cool.

JORDAN: Awesome. Niall.

NIALL: I am still on the artist formerly known as Twitter. I am on LinkedIn. You can find my company at www.stanza.systems, and I also keep a blog on reliability issues, which is at a URL-- search for my name kind of. Like, maybe it's the 5th result or something. I'm really selling this [INAUDIBLE] this is cool.
JORDAN: Definitely.

STEVE: We can link it in the show notes.

SILVIA: Go read his book.

JORDAN: Read the book. Absolutely.

SILVIA: I keep telling folks, the SRE handbook, go get that too.

JORDAN: Awesome. Well, thank you to Niall and Silvia, my co-host, Steve and The Witch of The Woods for this episode of the "Prodcast." Have an awesome day.

[JAVI BELTRAN, "TELEBOT"]

JORDAN: You've been listening to "Prodcast," Google's podcast on site reliability engineering.

Visit us on the web at sre.google where you can find papers, workshops, videos and more about SRE.

This season's host is Steve McGhee with contributions from Jordan Greenberg and Florian Rathgeber.

The "Prodcast" is produced by Paul Guglielmino, Sunny Hsiao and Salim Virji.

The "Prodcast" theme is "Telebot" by Javi Beltran.

Special thanks to MP English and Jenn Petoff.