EPISODE 613

[INTRODUCTION]

[0:00:00.3] JM: Over the last decade, cloud computing made it easier to programmatically define what infrastructure we have running and to perform operations across that infrastructure using code, and this is called infrastructure as code. Whether you want to back up a database or deploy a new version of a service or introduce a new tier of the load balancers, the changes that we make across our infrastructure can be done programmatically instead of through a series of manual steps.

As infrastructure has gotten turned into code, operations people have started working more like developers, and developers have begun to work like operations people. This convergence is often known as DevOps. At Google, the DevOps movement was manifested into a role called a site reliability engineer. DevOps and SRE, site reliability engineering, are not exactly the same thing, but it is a response to related changes in the way that infrastructure is managed.

In previous shows we've explored DevOps and site reliability engineering thoroughly. Laine Campbell is a senior VP of engineering it Fastly and the author of the book *Database Reliability Engineering*. In this book, Laine describes how the ideas of site reliability engineering can be extended to databases. Laine joins the show today to discuss the book and how engineering teams can build effective workflows around databases.

Before we get started, I want to mention, we're looking for a videographer. We're also looking for a writer. We're looking for several other jobs. If you're interested in checking those out, go to softwareengineeringdaily.com/jobs. If you want to get involved with us in a lower commitment way you can check out our open source community at github.com/softwareengineeringdaily. We've got several different apps for iOS, android, the web and they all have open source contributors. So if you're interested in getting involved, then we would love to have you as part of our open source community. You can check that out at github.com/softwareengineeringdaily. You can of course check out those apps at the app store for iOS or for android. With that, let's get on with the episode.

© 2018 Software Engineering Daily

[SPONSOR MESSAGE]

[0:02:34.1] JM: Nobody becomes a developer to solve bugs. We like to develop software because we like to be creative. We like to build new things, but debugging is an unavoidable part of most developers' lives. So you might as well do it as best as you can. You might as well debug as efficiently as you can, and now you can drastically cut the time that it takes you to debug.

Rookout rapid production debugging allows developers to track down issues in production without any additional coding. Any redeployment, you don't have to restart your app. Classic debuggers can be difficult to set up, and with the debugger, you often aren't testing the code in a production environment. You're testing it on your own machine or in a staging server.

Rookout lets you debug issues as they are occurring in production. Rookout is modern debugging. You can insert Rookout non-breaking breakpoints to immediately collect any piece of data from your live code and pipeline it anywhere. Even if you never thought about it before or you didn't create instrumentation to collect it, you can insert these nonbreaking breakpoints on the fly.

Go to rook out.com/sedaily to start a free trial and see how Rookout works. See how much debugging time you can save with this futuristic debugging tool. Rookout integrates with modern tools like Slack, Datadog, Sentry and New Relic.

Try the debugger of the future, try Rookout at @rookout.com/sedaily. That's R-O-O-K-O-U-T.com/sedaily. Thanks to Rookout for being a new sponsor of Software Engineering Daily.

[INTERVIEW]

[0:04:38.0] JM: Laine Campbell, you are senior VP engineering at Fastly. Welcome to Software Engineering Daily .

[0:04:43.2] LC: Thank you very much. It's great to be here.

[0:04:45.3] JM: So it's 2018, infrastructure as code and cloud computing are allowing us to update our software architecture more quickly, and more flexibly, more aggressively. How does this change how we think about database infrastructure?

[0:05:01.0] LC: Well, I think the first thing that happens is we find ourselves in a place where our database infrastructure isn't keeping up with the philosophy of changes that are happening in the rest of the organization and we're finding that we're making great progress in certain areas, and then anytime we have to interact with our data stores, we either have to have special exceptions, or it has to be manual, or painful and people are starting to ask why we can't do the same things that we do with the rest of our environment, the rest of our architecture with these data stores. They're also at the same time terrified of that because of the risk involved.

[0:05:36.6] JM: We have cloud providers these days. We can outsource much of our infrastructure uptime to cloud providers, but not all of it. Similarly, we can outsource some of our database administration to our cloud provider. What aspects of database management do we still need to manage ourselves?

[0:05:55.9] LC: So it is actually a great thing to be able to start to use these databases as a service to essentially get rid of some of the toil and a lot of the things that have traditionally consumed a significant amount of time and effort from our database engineers. But it definitely does not make it so we don't need those database engineers. Instead, it really is pushing those engineers up to stack, right? So at this point they should have more time, and with that time we hope that they will be focusing on data access patterns, data modeling itself, making sure that we are doing data governance. Essentially, all of the aspects that are much more closely tied to how this application works and then how that data is consumed and how that data is stored rather than replication backups, failover, all of those things.

[0:06:44.7] JM: Role of the database administrator has been around for a while. How does the role of database administrator differ from the idea of the database reliability engineer, which is something that you've written about in this book; *Database Reliability Engineering*?

[0:07:03.1] LC: So database administration has been around for a while. I myself have been doing it since the late 90s, and this concept of database reliability engineering is, it's kind of

nuance paradigm shift, right? I think it's important that we don't – Even as we look at these new paradigms, even as we look at new ways to do our jobs, that we don't forget about the importance of operations and reliability and things that we do here alike can sort of just be abstracted away, but in reality are still core to what we do.

The database engineers' world than what we're looking at here is people who really just like traditional reliability engineering. If you can call something so young traditional that we need people who do approach the concept of managing data from a software development perspective, but who can combine that with operations. People who are willing to develop a depth of expertise around state of modeling, around the risks of data loss or data corruption, being able to take widely distributed systems, micro service systems and provide the same rigor that in the past they might've provided in terms of replication and ETL, et cetera and creating what is an architecture wide focus on the data flow and helping to take that kind of knowledge, helping to take that kind of expertise and push it into the software development organizations. Because I think that is a lot.

Really, the key here is we know that we're never going to have enough database administrators, just like we're never going to have enough security people. So this database reliability engineer needs to approach their job as the work instead of being gatekeepers to the database, to the data itself, is how to enable self-service, how to enable scale from software engineers, how to get them in the databases, working with the data in a way that still manages risk and where recoverability guardrails are crucial, but no longer is it a case of, "Well, I need to wait for this database administrator to approve and gate keep all of these," but instead this database administrator has – Or in this case, this database engineer use their expertise, use their depth to create these self-service patterns that I know will work that I can apply.

Oh, here's one that based on the heuristics given to me from past where I've worked with, these specialists. I know that it is time to go work with the data engineer on the really crucial items that just can't be handled without someone put deep domain expertise, because you still do need that deep domain expertise.

[0:09:45.1] JM: As you said, the SRE role is fairly new to the public. It's a role that has been in the domain of Google for a long time, this site reliability engineering role within Google, which

we've done a couple of shows on. When Google started talking about it more publicly, they came out with that SRE book. Many companies looked at that and said, “There is something there that is appealing to me. The SRE idea takes DevOps and does something to it that either makes it more concrete or puts it in some different light that allows me to understand what I should be doing in my operations, my software operations side of things more acutely.” What is it? What did the SRE role that came out of Google do to the DevOps, the broader DevOps community? What realizations did it make people have about software engineering?

[0:10:49.2] LC: That is an excellent question and one that I'm sure no matter how I answer I’ll get in trouble for. The semantics arguments are very fun. What I personally view is that DevOps got this idea particularly for young, rapidly iterating organizations that we didn't need to silo away the concept of operations from our developers. In fact, developers who had access to the operating characteristics of their systems often wrote better services, wrote better applications.

I think that that was crucial and created a lot of value. Then what I think we started to see after that was a lot of people believed in the concepts of DevOps and they saw it work in certain places, but as enterprises and larger organizations, older over organizations went to do it, there were certain impedance mismatches. I think SRE is really the next step from this. It's this concept of we recognize that our developers should be brought as close as possible to the operating characteristics of the systems that they build, but at the same point there always will be this need for a core team of experts who can bridge the gap between operations and our developers even if they are practicing DevOps principles.

This idea of a team that is focused on scaling by teaching, by providing reuse really takes things the next step, particularly for environments that just simply can't get the traction with DevOps alone. It is sort of an expansion on the model and a mature model for a larger company where you’re never going to be able to give full access to your developers to do all the operations, nor do a lot of your developers want to, but you still want to give, embed in their teams these operators, these database engineers who have deep domain expertise and have shown to teach that to partner and pair that and bring that expertise into the decision-making of the day-to-day service developers.

[0:12:59.2] JM: At many of the organizations I've talked to, I'd get the sense that they are not full-time database engineer. So if you think about a company like Uber, I did a show recently about Uber's data platform in which they have a Hadoop cluster, and an Elasticsearch cluster, and Presto, and MySQL, and MemSQL and all these different databases and data things, but there's no particular engineer for any one of them. There are data platform engineers who help with the uptime of the data platform. There are people who are writing applications against front-facing APIs that are exposed out of that data platform, but there's not an actual person who is working on any one particular database. In an organization like that, who is practicing database reliability engineering? Is it all of us, or are you advocating for a specific role of a database reliability engineering?

[0:14:07.6] LC: It's dependent on the company itself. I think, particularly, if you look at older companies that have a long tradition particularly in Oracle, even mainframe data stores, SQL server, they have experts in these systems, and the idea that these are the people that only work on the data stores is one that I definitely don't think needs to be there anymore. Even at Fastly, we don't have anyone whose job is purely to work on the data stores themselves, but at the same point we have people who started their careers and built their careers being database engineers, database architects, who are now becoming platform engineers who are taking that expertise and not only focusing on MySQL, but starting to work on this general principle of how do I build a platform as a service in Google Cloud.

Since I think the type of skillset, the type of brain that makes a good database engineer often translates quite well into those platform architects that tends to be where we're going. But if you don't see those teams with people who I consider and, again, I don't necessarily think a company needs a database reliability engineering team. I think it can work. I think it's one model, particularly for environments that have significant data store needs. At the same point, I think even if you have a more generalist team of platform engineers, I would imagine if you look at Uber or anyone else, you've got database reliability engineers without the title doing that work, and I think that's the more important thing here is do we have people who not only deeply understand the storage mechanisms, the MVCC locking and concurrency characteristics, the optimization, all of those things, but are they also good teachers? Are they also people who can build patterns that can be applied elsewhere? That's what you end up getting out of those experts rather than trying to build this silo team.

[0:16:11.9] JM: And the SRE world has a number of subjects that are always worth discussing, and there are things like incident response and run books that we could talk about and I'm sure we will talk about. If we're trying to articulate what database reliability engineering is, is it those ideas, those abstract ideas, like incident response and run books and the other aspects of site reliability engineering and drawing a line from those to the database reliability, the database world, or is it something different than that?

[0:16:55.7] LC: I think that is part of it. Obviously, if you need a database, someone who has deep knowledge on the date level – Sorry, I'm losing my power of speech there for a moment. Who can then apply those principles and say, "Okay. These are the things that anyone who is deploying a data store should automatically get these monitoring templates. They should automatically get these failover, the tooling and run books associated with it." These are the areas we need to abstract out so a generalist can perform some typical tasks and these are the areas that we can start applying to the CICD platform as well around how to apply change to these systems. So you do become – I think the DVRE should and can become almost linchpin in a reliability organization as a whole who can take those principles, take them down to the level of depth necessary to apply them to the data stores and also who can learn from other reliability experts on risk management, on service level objectives, incident response, all of these areas and build a richer, deeper team. That's one side of it.

But I think the other side is this concept of the DVRE should be focusing on eliminating the gatekeeping, and that means building guardrails, providing education, collaboration, teaching people how to work in these environments and making sure, whether it's through shadowing, and embedding, or it's through self-service, that we're building systems that are less risky to allow any software engineer to work within, because obviously the level of risk involved with working within the data stores is much higher than within your web server or even your caching tear or anything else.

[0:18:50.8] JM: Eliminating the gatekeepers. When you say gatekeeper, are you referring to the barrier to getting a question answered for – If you have some data related question, the fact that you have to go to somebody on the data science team or the data engineering team and have

them figure out how to write your query against the Hadoop cluster. Is that the kind of gatekeeping you're talking about?

[0:19:15.5] LC: No. I think that's more of a symptom of just centralization and siloing. I'm talking more along the lines of we don't allow people to push DDL, make table changes into the data store, or we don't allow people to necessarily write SQL or do data modeling without X-X-X reviewing and approving, or the CI platform works great and we're even doing continues deployment or some level of continuous delivery. But, hey, when it comes time to apply the changes to the database, we now need a human from this specific team to come in and apply them. That's more of the kind of gatekeeping I'm talking about.

[0:19:55.7] JM: Why is that important? Why is it important for anybody in the organization to feel empowered to write queries against the database?

[0:20:03.4] LC: Well, I mean, the data is the life blood of the system of the business, and the more that people can query or interact with the data, the more experimentation you're going to have, the more knowledge and database decision-making is going to be driven there. Similarly, I think it's a matter of velocity, right? All companies right now are competing based on how quickly they can get features out, how quickly they can ideate on them and pivot and change, and almost all of that at some level will require changing what data is stored or changing how you access that data. If you are dependent on a core team of people for that, then they will inevitably slow you down and then you will use in the competition for feature deployment and ideation.

[SPONSOR MESSAGE]

[0:21:02.8] JM: Test Collab is a modern test management solution which integrates with your current issue manager out-of-the-box. For all development teams, it's necessary to keep software quality in check, but testing is complex. There are various features to test. There're various configurations and browsers and platforms. So how do you solve that problem? That's where Test Collab comes in. Test Collab enables collaboration between software testers and developers. It offers wonderful features like one click bug reporting while the tester is running the tests, collaboration on test cases, test executions, test automation integration and time tracking. It also lets you do a detailed test planning so that you can configure platforms and

browsers and configurations or any variables in your application and divide these tasks effortlessly among your team for quality assurance.

All of the manual tests run by your team are recorded and saved for analysis automatically. So you'll see how many test cases passed and failed with plenty of QA metrics and intelligent reports which will help your application's quality. It's very flexible to use and it fits your development cycle. Check it out on testcollab.com/sedaily. That's T-E-S-T-C-O-L-L-A-B.com/sedaily. Testcollab.com/sedaily.

[INTERVIEW CONTINUED]

[0:22:41.5] JM: I think I'm understanding what you're saying here. You're saying that if you can get to a culture that has well-defined incident response, for example, you won't be as afraid of changing the database or having the intern write a new query against the database, because even if the intern brings down the database, you have a well-defined way of responding to an incident so you don't need to be afraid of that in turn query.

[0:23:13.4] LC: Exactly. The guardrails are in place to manage that and they're at different levels, right? Like you said, incident response as a whole detecting when there's a problem. Obviously, architecturally, you can create environments where access to data can be done on a data store that is not the same as the production data store. But similarly if you've created an environment of continuous recovery, this idea that your backups are not only running, but they're always being tested, they're used every day in standard deployment practices, standard test and integration practices and there is not a single backup that exists that hasn't had some level of integration or testing against it, you're going to be more comfortable with this idea of letting people iterate on the data models because you know recovery works. It works well and it works fast.

Similarly, I think a lot of people are comfortable now with this idea of any number of read replicas available that you can distribute work across queries, across everything else, and that model works well. We even get to the point where someone might be willing to say, "Okay. I'm willing to apply a little bit of chaos engineering and shoot a replica in the head, because I know I have a load balancer. I know I have extra capacity," but then it's taking it to the next step and

saying, “We actually have tested for failovers of the data persistence, the writing to the database well enough that that primary data node could even get shot in the head,” and we know exactly what will happen. We have back pressure mechanisms or secondary persistence mechanisms in place to make sure that we don't lose data and we test it regularly. At that point, you get to the point where no one is even concerned about “What happens if I have to you'll fail over the primary node?” and that is a significant paradigm shift and how people think about the data stores, is the comfort with the risk that comes with letting people create failure situations on the primary notes that are taking writes in versus failure situations in the more horizontal read notes of a data store.

[0:25:16.8] JM: There are some challenges that people can encounter when they're trying to make these cultural changes, and I think database reliability engineering, as I'm hearing it from you, is really about a mindset shift. It's about a cultural shift as supposed to being about any specific role change or any specific point organizational change. It's more about a shift in mindset to being able to move faster and move more comfortably.

I went to the DevOps Enterprise Summit. I think it was like a year and half ago, and it was interesting because there were a lot of organizations that I would –I don't talk to them on Software Engineering Daily, but they are companies that are doing software engineering and it's some company in the Midwest that has lots of software engineers and they have lots of software engineering problems and lots of legacy issues and they come to the DevOps Enterprise Summit because they're trying to learn to move faster and you see them struggling and eventually succeeding in figuring out how to adapt their organization. But the point is that I want to avoid Software Engineering Daily becoming this place where I just like talk about these things in theory and don't witness them where the rubber meets the road.

So I’m very curious, because you work at Fastly, and that's a high throughput, intense engineering project building a CDN. So when you think about applying these kinds of mentalities that data database reliability engineering mentalities to a CDN, like Fastly. I mean, a CDN is in some ways just a big database that has a lot of operational challenges around it. What are the challenges that come to mind in trying to implement database reliability engineering practices?

[0:27:08.0] LC: It is interesting that you talk about Fastly and this concept of our edge network being giant global key value data store, because that is what it is, and I laughed when I first joined the company and saying, “Someone who spent their entire career working in databases comes to a company that doesn't really have a lot of data of significant size.” In fact, we don't store logs for customers, right? We make sure that those laws go directly to their customers, but we don't store them. Our entire edge storage is essentially volatile and doesn't need to be persistent. So it was pretty funny that I was coming to this organization, but our CEO, Artur Bergman, consistently said, “No. You are just working on one of the largest data stores ever.”

In many ways, they've already applied a lot of those concepts to manage an edge, to manage a data store that is so distributed. We've already had to solve for velocity. We’ve already had to solve for reliability in terms of making sure we can recover. In this case, we don't have to worry that much about recovering our data, but we do have to make sure that we’re not running in split brains or places where the data isn't there.

So the edge side, I think a lot of customers have already solved those kinds of problems for that, but then when you look at the control plane side of our system, where the systems that provide the interfaces for customers to Fastly and how it works, or looking to get data out of our systems about how their end-users are experiencing Fastly and what's going on there. That is a place where we are undergoing an active cultural change in what we do, and because our data store is in those environments aren’t huge, but they have to be reliable, right? Our customer can't lose the ability to configure how their edge works. At that point, they’re in deep trouble.

So we have a case where we said, “How do we optimize for reliability without impacting how frequently we can iterate and change and grow for those control plane environments?” There are particularly cultural changes involved, because for one thing, I think people who are used to operating a fleet or operating a very horizontally distributed system, when you ask them to start looking at potentially becoming on-call and become regular operators for the data store, they put their hands up. They’re like, “Nope. I don’t want to touch that. That is something terrifying,” and culturally you have to, first off, make sure that you can't just hand someone who is a generalist, a data store and say, “Manage this without training, without runbooks, without ideally some level of abstraction on failover and backup studies very easy, and that you can teach people to do and show them, “No. It actually works very well,” and that's what you end up having to culturally

do, is to build the self-service, build the guard rails and start to teach these people and show them, “No. Look. You can do this look. Look. We’ve now invested in a proxy layer that allows you to do a failover in the middle of the day without anyone noticing. Now, let's go have each of you do this. Now, let’s shadow an operator from – Or even a developer who does database engineering with a failover and show them how it works. Show them how our recovery works.”

It takes time and it is a matter – It takes time not only for those engineers who are terrified of working in those environments, don't necessarily want to. But then on the other side, for the managers, for the risk folks, the compliance folks, everything else, to show them that you can create an environment where you’ve de-risked this, you’ve created this ability and to what happens in your data stores so that they themselves feel like this isn’t Wild West. This isn’t, “Oh! Everyone's in there doing anything now,” because, honestly, some holders companies are just getting to the point where they're starting to give people a little more system access outside of their core area. So then to tell them, “Well, now, everyone can work in the data store,” and that isn’t necessarily true anywhere. Not just anyone can work in the data store. There is still risk. This is still one of the most important – Actually, I would say the most important area of our architecture.

So it's not like we’re just saying, “Okay. Day one, there you go.” There is rigor involved in teaching people how to work and how to evaluate risk and how to learn from it, and on the other side, it is teaching those folks who do the risk management, the compliance manager, the incident management and showing them the auditability aspects of it, the controls that are in place to mitigate any impacts the guardrails that have been done, and that comes to a lot of more core areas of SRE and even DevOps around what are the metrics we’re tracking? What are the controls in place? What are the processes that we’re doing to make sure that we are optimizing for recoverability of a system rather than trying to build this sort of robustness that's actually incredibly brittle and fragile, because it's robust, so no one's ever had to actually deal with a failure for the last two years. So when it comes, everyone's hair is on fire.

[0:32:09.4] JM: One aspect of database reliability engineering that I have seen you talk about is capacity planning, and capacity planning is really important because you need to know in advance if your database is going to grow in size to something that's beyond what your current provision capacity can tolerate, because you're going to need to plan to buy more database

nodes. So I imagine this is an issue at Fastly among other companies, but describe the capacity planning process for a database reliability engineer either through the eyes of Fastly or more generally.

[0:32:54.2] LC: It is a very multidimensional approach, right? So first of, it's understanding of the scaling pattern that you've employed and understanding the constraints therein. Are you still in the phase where you are vertically scaling your database nodes or have you gotten to a point where you have decided what a good unit of work is for horizontal node and apply your models to that?

So it's understanding that part first, and there are certain areas and constraints within there that you have to look at. There is no single one. I think a lot of people, when they're thinking about database capacity, they might be thinking about storage, obviously, disk space. Some so people are thinking of memory and working sets, but then there's other constraints around concurrency within the system, which varies tremendously based on what database you're using, if you're using a relational store or you're using Redis or something similar. Their connection then concurrency models are quite different.

So first off, you have to build the model in place of what are the constraints that could potentially max out, bottom out? From there, with load testing, ideally you identify where those two max out. At that point then you start deciding, "Okay. Based on the load testing that we've done, based on the scaling model we've done, we know that when this metric hits here, this metric hits here, that we have to start doing this." Then once you've sort of said, "Okay. That's how we get through the current model," we know then the next step is going to be either functionally partitioning our data store, or sharding it, or anything else. Let's start to look at when that constraint is going to come in place and start deciding the amount of effort needed to shift to that, because you don't want to optimize too early, and start to look for those key factors as well.

That's great if your loads are linear and you can sort of just know, "Okay. Well, this happens, this happens, this happens," but the reality is this is data storage. So within that, if you are a young company or a rapidly iterating company, you're constantly changing what you do, and that means you're going to have different indexes, different tables. So even – It's not just a matter then of saying, "Okay. Well, X customers means X amount of storage, X amount of queries."

Instead, then you have to keep up with all the changes that are coming into your system. That is where ideally you're teaching your developers to not only understand what they are doing in terms of data access and data storage from a functional perspective, but you're making sure they understand the underlying characteristics of that data store and how that applies to change.

Okay. Well, we know that this data store it's a binary tree index. You always, on a new table, get a primary key, which adds an index immediately and then we know that based on the characteristics, it's going to also need this. We know with these data types, that that happens. So at that point if you taught your software engineering team about the data store, and that's one of the key areas where it gets scared when people try to abstract the engineers away from the data store. You then don't have advocates for capacity. For people who will understand the impacts of their changes not just from the perspective of performance and functionality, but also the underlying sort of longer-term shifts and the workload that will completely invalidate your capacity models. That's where I think a reliability engineer for databases with deep domain expertise is invaluable, because they can teach people the impacts of a data type change, a table change and index change.

[0:36:39.7] JM: The job of the SRE – And a lot of that Google SRE book is about automating yourself out of any manual work if possible. If you can write a script to do something that you're doing manually, you should write that script. Are there any tasks that the database reliability engineer should keep as manual tasks just to force themselves to do the process in order to have more scrutiny over that process? Because I think it's like a mental trick that sometimes people play on themselves where they say, "You know what? I want to leave this process as manual, because I'm kind of afraid of automating it."

[0:37:19.2] LC: Yeah. So I think automation itself is an interesting topic and I think this idea that we can automate almost all of our manual work. You probably could, but would it be the right amount of time and effort to automate everything? So I really ask people when they look at a process and trying to decide what to automate and what not to sort of value stream out the various components of the overall process. What is either takes a lot of time and effort? What is risky and having a human do it over and over again creates risk is a repeatability of software,

obviously, that precludes no bugs or anything else. It makes it easier for something to happen without a human error coming into it or being attributed to a problem.

So when I ask people to look at what's automated, it doesn't matter of saying, "Okay. So what de-risks the day-to-day operation of the system, and then what potentially you adds risk?" Just because something – There's nothing that should be automated that a human shouldn't be able to still do. So, for instance, failover, right? If you get to the point where you have primary server failover or a secondary server, the read replica failover is automated, there are still times when you should regularly test those, because what if the automation is down. What if it's a bug in the automation software itself and a cascading failure therein?

So you absolutely do need to teach people how to do the day-to-day traffic shifts and recoveries that an automated system can do. The same for a data recovery, right? Yes, you should be using this as part of your everyday work. You should be automating the testing and as much of the recovery as possible, but you still should be asking your engineers to some reasonable level of frequency to test or restore themselves, to test a failover without that automation. Otherwise they will get in the same – Just like I said before, where you have a reliable system that's fragile because no one's tested the failovers. Even an environment that's incredibly resilient to this through automation, if you haven't tested those processes themselves, then you're going to come into the same thing. So I think anything that is a matter of recovering a service to functionality for users can, and if you can appropriately de-risk, should be automated. But, nonetheless, you need processes in place for regular game date and testing and just to make sure that people can do it if and when those systems fail.

[0:39:51.3] JM: We've had some differing opinions on the show of the idea of using a write-ahead log is a tool for recovering from an incident. For people who don't know, the write-ahead log is this append only log that a lot of transactional databases have, that it's not the actual database itself. It's a history of the changes across a he database. So if you have the write-ahead log up into timestamp X and then you have the database of timestamp X, the database of timestamp X is great because that's a transactional data store, but the write-ahead log is useful because you can actually understand the entire history of how the database has looked from what you can reconstruct from the write-ahead log.

Now, that said, the write-ahead log is much more verbose than the database. So there are these differing approaches. So, Tammy Butow from Gremlin now, she does chaos engineering. I asked her about this question, like, “Should you use the write-ahead log as a means of recovery from some kind of incident where you can reconstruct up into timestamp X?” Her approach was, “Yeah. I mean, if you need to look at the write-ahead log, but you should have recovery mechanisms in place such that the write- ahead log is not your source of restore. You would much rather just have database snapshots as your recovery vector. Do you have an opinion on one of these two recovery mechanisms?

[0:41:26.4] LC: Like any good database engineer, it depends. So I think one of the things I talk about in the book in the recovery section is this concept of recovery in depth and having a multi-tiered approach, because you can't always guarantee exactly how you're going to have to recover a data store. Day-to-day, what you use to get a system back in service or to provision new instances, add capacity, that absolutely should be snapshots wherever possible. Something that is on fast storage that is easily put in place and allows for quick transport of data. Those are ideal for that solution.

Then you have this concept of – And it used to be the idea of a logical versus a physical backup and you might – If you’re in the MySQL world or MySQL.parade, that is similar in many ways to what you're talking about, because to restore, you have to basically apply and insert for everything.

In this case, you’re capturing the changes, but those changes, first off, they can corrupt, they can get corrupted. So you can't depend on them completely. They’re very time-consuming to recover from, because you have to apply every change. But they do have places. Some of those places are when you are reconstructing data after a very complex corruption, and that corruption could come from something in the distributed system environment itself or it could come from the corruption based on an application and some really interesting logic there. You, of course, are going to need to reconstruct that.

So in those cases, it's a great tool. One tool you didn't even mention, which is also very interesting, is the idea of versioning objects in the database. Just like if you look like an object store and the idea that you can version what's in that bucket and then reapply it. There's no

reason that you can't store previous versions of an object as you mutate it. If you mutate a row on a database, if you always store previous version of it and you had a large bug, you can actually programmatically let the application fix things in a just-in-time approach. So when a customer comes in, either engaging them and saying, “Hey, we have previous two versions. What’s the most recent one?” That’s obviously a very blunt example, but making it so that you have multiple ways to restore your data, particularly as your datasets grow and you get multi-petabyte and even multi-terabyte, this idea of ever recovering your database directly from a backup of record is less and less likely.

So at that point you have to look at, “How can I incrementally fix this in a way that it doesn't destroy my system for days at a time?” for weeks at a time in some cases. On the other side of it, “How can I make my datasets as portable as possible to allow for creating test environments on the fly, creating – Adding to the past and anything else.

[SPONSOR MESSAGE]

[0:44:28.4] JM: Citus Data can scale your PostgreS database horizontally. For many of you, your PostgreS database is the heart of your application. You chose PostgreS because you trust it. After all, PostgreS is battle tested, trustworthy database software, but are you spending more and more time dealing with scalability issues? Citus distributes your data and your queries across multiple nodes. Are your queries getting slow? Citus can parallelize your SQL queries across multiple nodes dramatically speeding them up and giving you much lower latency.

Are you worried about hitting the limits of single node PostgreS and not being able to grow your app or having to spend your time on database infrastructure instead of creating new features for you application? Available as open source as a database as a service and as enterprise software, Citus makes it simple to shard PostgreS. Go to citusdata.com/sedaily to learn more about how Citus transforms PostgreS into a distributed database. That's citusdata.com/sedaily, citusdata.com/sedaily.

Get back the time that you're spending on database operations. Companies like Algolia, Prosperworks and Cisco are all using Citus so they no longer have to worry about scaling their

database. Try it yourself at citusdata.com/sedaily. That's citusdata.com/sedaily. Thank you to Citus Data for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUE]

[0:46:13.5] JM: Let's talk about the concept of failover more generally. How much failover do you need, or how do you calibrate the level of failover that you need for a given database application? I mean, you can have your failover to different nodes in the same data center. You could have multi-geolocation, multi-availability zone distribution for different failover recoveries. What's the model for determining the level of failover that you need?

[0:46:44.3] LC: Yeah. I think it is a – I mean, it's a general risk management exercise in terms of looking at the failure scenarios involved. You mentioned a lot of them just now. What're the most typical failures? Are they node failures? Are they nodes that then have to be recreated? What's the likelihood of a zone failing? What's the likelihood of a region failing? Then deciding based on the application itself, what uptime that needs based on service level objectives. Then understanding, of course, databases are often multitenant. So you have many applications on them. So the more centralized your database is, more services on it. The recognition that, well, all services within that database will then have to be set up at a level of redundancy and replication factor necessary for the most critical, and that's a case where you can start looking at functionally partitioning your workload so that services that are less sensitive to downtime can be put on systems that are less available.

So it's, first off, understanding that those multitenant workloads and what you need to optimize for there. Then it's a matter to your service level objectives and risk tolerance and deciding from there how long does it take you to restore service and is that acceptable to the business and what's the likelihood?" It's all just a calculation at that point of what's the cost of my downtime, which can be hard to quantify, but you need to come up with some level of model that puts that in place. What are the mitigations? Whether from a feature flags, putting your application in read-only mode, anything else? What are the mitigations that give you more time? Taking all that into account, it's just a simple calculation of cost versus cost of being down versus the cost of maintaining those services.

Obviously, if you're doing the reliability engineering model right, the more databases you add doesn't add a linear cost to operations, but there still is a cost to that complexity. I think that's a nuance that has to be very carefully considered, is when you start going multi-data center, multi-region, your distributed system starts having much higher latency across connections, which adds its own risk, whether that's a risk of collisions, weird failovers that create split brain scenarios.

So at that point, you have to start considering, “Okay. If I have come to the conclusion that I need this, this and this, am I taking into account the complexity of those distributed systems and the potential failures that come from having –” Just because you set up replication across regions into databases doesn't mean that it's going to work well. It doesn't mean that you're not creating more risk than if you had just optimized for the ability to easily redeploy a service in a different zone than always having a hot spare available with one second latency in the replication string.

[0:49:44.9] JM: As you explained earlier, much of your positioning on this database reliability engineering topic is about spreading the knowledge within the organization, having more egalitarian access to database knowledge. Do you have any favored practices for evangelizing information within an organization about databases?

[0:50:14.0] LC: Let’s see. So I'm a big fan of pairing, and I think pairing between teams. So pairing a database engineer directly into a team of software engineers can be incredibly valuable. So, for example, I was engaged for a little while with [inaudible 0:50:32.1] on their infrastructure as a service system, and I am not a software engineer. I’ve written sort of procedures. I’ve written shell scripts, but I am not a computer scientist and I would not consider myself a programmer of any worth. But as you know, pivotal is very intense on their pairing, so we did it anyway and I learned a huge amount and those software developers who had access to a database expert, while they were building the services, they got tremendous value. Because it was pairing, it was active problem-solving based on what they needed to know right then versus just a piece of content that may or may not be applicable to what they're doing. So I think the more you can build these matrixed or just even temporarily paired teams that cross the database function with the other functions that you ideally are giving information to, the more

likely that that information is going to stick and those people are going to get more used to working together.

Similarly, I think the shadowing with on-call, even if you don't want to make software engineers on-call for the database services themselves, pairing them up for shifts so that they can just watch and understand or pairing them up for the changes, right? It's like, "Well, we'd like to add these three columns history indexes to this table that's a multi-terabyte." Making sure that they actually understand how that works and what are the processes involved at the persistence tier to apply a change of that size without impacting customers is a great way for them to have respect for the data store, but also start to understand it. So that respect becomes an informed respect rather than the sort of mysterious, "I don't ever want to touch the side of the system."

Another thing I like, particularly for software developers looking to make changes to the data stores or queries, is a library of patterns and even anti-patterns. So the this idea that, "Okay. There are a finite number of mutations you might want to make to a table or a data model," and so as we do those changes and as a database person embeds with a team and teaches them, let's then add that to the pattern library so that in the future when a developer says, "I want to do this," they know immediately where to go to look at how to do that without having to consult a database person.

[0:52:52.1] JM: So in many businesses, the customers are going to be price-sensitive to whatever service you're offering them. So if you have a person who is familiar with the database infrastructure, they can find ways to cost save on that infrastructure. Would you consider the idea of finding cost savings within database infrastructure to be in the purview of a database reliability engineer?

[0:53:22.7] LC: Absolutely. I mean, cost savings is essentially just a trade-off, and at that point someone who understands the implications of the trade-off can partner with finance or with anyone who is responsible for the OpEx of that environment assuming a cloud data store and come to those conclusions.

So, I mean, anywhere your cost savings is or they're going to come from reducing throughput, producing storage, reducing availability and replication factors. Each of those trade-offs comes

with a potential risk implication. The database engineer or reliability engineer can then take that, their knowledge of the system, and it can be anyone who's familiar enough with that system to say, "Okay. If we are going to make this cost savings trade-off, then this will impact our ability to hit our service level objectives," whether they're an availability objective, or a latency objective, or a durability objective, because that's all it really is then, is an exercise of understanding the trade-off, the potential risks and deciding if that service, that new service with the cheaper cost factor is still a service that a customer would be comfortable working with.

[0:54:32.3] JM: I touched on the Tammy Butow conversation I had a while ago and I was talking to her mostly about chaos engineering. What do you think about the idea of instituting random failures in the database? Is the database too sensitive? Too exposed? Too chaos engineering type practices?

[0:54:52.7] LC: I think that you can absolutely start to explore chaos engineering with a data store that's designed for that purpose. I will be up front, just like you said, where does the rubber hit the road on theory versus practice?

A lot of this database reliability work is still being tested and practiced and some of it is conceptual. Most of it I've seen somewhere or I've done, but where you start getting into the world of, "Hey, we'd like to have a chaos agent that can shoot a write master or a write primary in the head and we're comfortable with that being a non-planned exercise," there aren't too many companies yet that are actually willing to do that. But I think of that – When I work with our engineers at Fastly and look at the data source, that is the aspirational goal, the 20% of the OKR that probably won't get met that I asked them to architect and design for, is this idea of, "If I came in on Sunday morning and shut down that data store," which I can't do because I'm an executive and I don't have access to anything in our systems. But if I could or if we wanted a system to do it, would we be able to sustain it?"

Right now I think it's just having those thought exercises and doing a tabletop walk-through of what would happen and why and then starting to do a control failure and see if what hypothesis was actually happened is a great start to that. Then when designing, starting to think about that as well. Are we to a place where we could easily do this and where most people would feel

comfortable? No. But I think it is a worthy thought exercise and it's a worthy design factor to add into your architecture.

[0:56:36.3] JM: What should the average software engineer take away from this conversation about database reliability?

[0:56:42.6] LC: Well, I think they should – Ideally I think what they should take out of this is that, one, they know that what they feel is a security factor of having a team of experts that they can rely on. Being there for them is one that they probably can't depend on, just because those folks don't scale. They should be looking for areas where they can gain knowledge about the data stores they're working with, particularly the ones they are working with rather than relying on abstractions away. They should be looking for opportunities where if they have the ability to get knowledge from database experts, that they can start sharing it within their teams.

Similarly, I hope that they take away from this the idea that they should be able to work within the data stores. Obviously, that requires guardrails, but if they start thinking about how they can do their work every day and without necessarily being blocked by subject matter experts who aren't in their teams. When they start to think about that and start to think about the guardrails that they can put in place so that if a data server fails, that their replication still work even if partially or an integrated state, the more opportunity they will create for them to have that kind of velocity by working within the data stores.

[0:57:57.9] JM: You've written a book with Charity Majors, and she was on the show a while ago talking about – Well, we mostly talked about her past experiences with the acquisition of Parsed by Facebook. That was actually a really good set of stories, but I like Charity a lot. She's very entertaining and has a whole lot of information to impart that's quite useful for people who are in this, I think, probably the space of operation/logging/monitoring/infrastructure conversations. What have you learned from Charity Majors?

[0:58:36.2] LC: I've learned how to drink from Charity Majors, how to drink properly. I grew up in New Orleans. I already knew how to drink. But outside of that, one of the most wonderful things about what Charity's knowledge is the fact that it is a pretty much nonstop informed by real life and real life decisions and trade-offs.

Charity came into the database world, as she calls it, the accidental DBA, and I came into this world as the crazy person who decided the first day of their world in computers that they wanted to be a database engineer, which probably, it's a psychological dysfunction that should be in the DSM. But Charity's outlook on, "No. I didn't ever plan on being a database engineer and yet I was forced to," is a very different view than mine. So that paradigm and her approach to looking at this is very different from mine, right. She also comes from the startup world and I came from the world of Travelocity. I joined Travelocity when it was still a startup, but it was owned by Sabre and EDS, and these are very traditional mainframe and server system. So I learned a lot from those environments. I even did ITIL and various other things that can be considered archaic now.

So combining this iterative just-in-time approach that Charity takes to building an environment, because you don't even know if your business will be in business the next day, versus the careful planning and meticulous design that I bring in to an environment that's already built, scaled, at that point needs to be made faster and more agile is it's a great mix. It's a great complement or two skillsets.

[1:00:22.5] JM: Laine Campbell, thank you for coming on Software Engineering Daily. It's been great talking to you about database reliability engineering.

[1:00:27.2] LC: Oh, it is my pleasure. Thank you so much for having me.

[END OF INTERVIEW]

[1:00:33.6] JM: At Software Engineering Daily, we have user data coming in from so many sources; mobile apps, podcast players, our website, and it's all to provide you, our listener, with the best possible experience. To do that we need to answer key questions, like what content our listeners enjoy? What causes listeners to log out, or unsubscribe, or to share a podcast episode with their friends if they liked it? To answer these questions, we want to be able to use a variety of analytics tools, such as mixed panel, Google Analytics and Optimizely. If you have ever built a software product that has gone for any length of time, eventually you have to start answering questions around analytics and you start to realize there are a lot of analytics tools.

Segment allows us to gather customer data from anywhere and send that data to any analytics tool. It's the ultimate in analytics middleware. Segment is the customer data infrastructure that has saved us from writing duplicate code across all of the different platforms that we want to analyze.

Software Engineering Daily listeners can try Segment free for 90 days by entering SEDAILY into the how did you hear about us box at signup. If you don't have much customer data to analyze, Segment also has a free developer edition, but if you're looking to fully track and utilize all the customer data across your properties to make important customer-first decisions, definitely take advantage of this 90-day free trial exclusively for Software Engineering Daily listeners. If you're using cloud apps such as MailChimp, Marketo, Intercom, AppNexus, Zendesk, you can integrate with all of these different tools and centralize your customer data in one place with Segment.

To get that free 90-day trial, sign up for Segment at segment.com and enter SEDAILY in the how did you hear about us box during signup. Thanks again to Segment for sponsoring Software Engineering Daily and for producing a product that we needed.

[END]