# EPISODE 829

[INTRODUCTION]

**[00:00:00] JM**: Relational databases such as PostgreS are often used for mission-critical workloads such as user account data. To run a relational database service in the cloud requires a cloud provider to set up a highly durable, highly available system.

John Daniel is an infrastructure engineer at Heroku. John joins the show to describe the engineering and operations required to build a managed relational database service. Full disclosure, Heroku is a sponsor of Software Engineering Daily.

[SPONSOR MESSAGE]

**[00:00:40] JM**: The Techmeme Ride Home Podcast is a short news summary of things that have gone on in the previous day. Every day at 5 PM Eastern, Techmeme Ride Home releases a 15 to 20-minute summary of what happened in the previous day. This is pretty useful in a time where technology news are both moving so quickly, and the Techmeme Ride Home Podcast is not just headlines. It's detailed conversation over the course of 15 to 20 minutes.

The podcast collects all the context and the conversation around the news as well, the tweets, the blogs, the subreddits, the medium posts, the commentary. The folks at Techmeme are online all day and they read everything so you might not have to.

You're busy making technology, and building software, and creating cool ideas, so let the Techmeme Ride Home Podcast summarize the days goings-on for you. You can go to techmeme.com. You can search for the Techmeme Ride Home Podcast, or just search Ride Home in your podcast app and you'll find the Techmeme Ride Home Podcast.

[INTERVIEW]

**[00:02:04] JM**: John Daniel, you are an infrastructure engineer at Heroku. Welcome to Software Engineering Daily.

**[00:02:08] JD**: Thanks, Jeffrey. Thanks for having me here.

**[00:02:11] JM**: I'm excited to talk to you about running databases in the cloud, because that's a complicated infrastructure problem. I want to start off by just talking at a high-level about relational databases. Why do people use relational databases?

**[00:02:28] JD**: Yeah, this is a question that we get a lot, especially with things like CouchDB, Mongo or whatever the latest NoSQL solution is these days. People pretty much use relational databases because they are well-understood and their battle tested ways of storing, retrieving and processing data.

**[00:02:46] JM**: And what kinds of queries perform better in relational databases versus the other kinds of databases you mentioned, like document databases like MongoDB?

**[00:02:56] JD**: Yeah. So any query that kind of involves like an algebraic-like relationship, such as associating like a group of people with a group of addresses and you have two different data sources to work with. Also, these transactional natures can be – The transactional nature of these databases, they can be really useful for things like financial systems or customer relationship management tools.

**[00:03:16] JM**: So there are different relational databases that have different performance characteristics. We have PostgreS, we have MySQL, we have SQLite. We have the cornucopia of “NewSQL databases”. How are the different databases that we encounter in the wild, how are they making tradeoffs and what are those tradeoffs?

**[00:03:40] JD**: Right. So I used to work with MySQL pretty heavily before I made the jump to PostgreS about three years ago. So basically PostgreS and MySQL for a while were sort of neck and neck and what they can offer in feature set. PostgreS these days seems to offer more native, like built-in types that they have, like a standard, like JSON type. They have one for siders. They have loads and loads and loads of others. Whereas I feel it MySQL has kind of taken the let's take this project that we already have and keep improving the performance and the stability of it, but not really kind of expand beyond that.

But MySQL has the advantage of it kind of being like the first big open source SQL database to gain the sheer volume of traction that it did. So it's always going to have that built-in section of users. PostgreS on the other hand, I feel like it iterates far faster. It also made some design choices that I think are a little better and that it relies on UNIX process types instead of like handling threads and all of these other things. I find it easier to administer and easier to track down issues when they occur.

Then we have things like SQLite, which Rails including SQLite by default was I think a mistake, because running a web application on SQLite is just not what it was designed for. It was meant for embedded SQL. It was meant for running things on a device where you don't necessarily have network access. The fact that Apple included SQLite in the iOS for a lot of like the app backend management, I think that is a more appropriate use of it and something I wish more people would understand. But I think SQLite in general is an extremely, extremely powerful tool. Way better than all of us building our own half-baked SQL implementations for our local apps and such.

I'm not as familiar with some of the new SQL databases, but there are a few that I've seen that look interesting. I know that like React kind of came about a while ago. I'm not entirely sure what query language it uses, but a lot of people in the operations world seem to get very excited about it. At a previous job, we used Datomic a bit, which sits on top of either PostgreS or MySQL, and it allows you to do some really interesting things, like going back kind of in time to query differences and all of that stuff, which sounds extremely exciting.

**[00:06:06] JM**: So the subject of relational databases, we can go in a lot of different directions, but we've covered relational databases and their usage in a lot of previous episodes. For this episode I’d like to take the approach of how do you run a relational database in the cloud from the point of view of a cloud provider.

For example, if I'm spinning up a web application where I'm using a SQL database as the backing store, I want to be able to just one click instantiate that database and start to make requests to it, and that feels very seamless and simple from the developer's point of view today. But, of course, so much infrastructure work has gone on to allow that to be a seamless experience. Could you just give us an outline of what are some of the challenges, the fundamental challenges, from a cloud provider's point of view of offering a SQL database in the cloud?

**[00:07:11] JD**: Oh boy! Where do I even start? So offering a SQL database in the cloud, it feels like a very simple process coming from a kind of more traditional like IT systems administrator background. You grab an operating system like Ubuntu or CentOS, you throw it on a server. Yum or apt-get install PostgreS and you're ready to go.

But from a cloud provider standpoint, it's a much more involved process. First off, before Heroku even existed, whenever we had to make these decisions about providers and do we run the servers ourselves, do we use something like Amazon or DigitalOcean or Linode. First off, you have to make that provider decision. You have to look at pricing. You have to look at what they offer. You have to look at what kind of support is available, and especially automation. What APIs do they expose? So you have to make that provider choice fairly early on, which is a big decision.

Next up, you have to investigate what exactly do they offer from an instance perspective. In older days of AWS, people would mistakenly provision a lot of services using their ephemeral disks, not realizing that as soon as your EC2 is rebooted, all of your data is gone, and they had EBS, which some people found to be rather complicated. But you have to make sure that your data is persistent. So you have to have a server first. Then you have to provision disks. You have to figure out what kind of disks do you need. EBS has like their gp2 drives, which are general-purpose spinning disks. Then they have all their IO iterations that you can provision, what are called prIOPS, which are priority IOPS. So how fast is this disk? How often can you read and write to it?

So you have your server. You have your disk and you need to figure out how to connect it to your greater environment. The old days of Heroku, we published everything open on the internet, and it was behind an EC2 host name. That worked for the time, but these days we actually run everything inside of a VPC. So it's a virtual private cloud, which you can think of it as like almost like a VLAN of sorts, or a like mini-AWS instance or mini-AWS ecosystem that you have access to.

So then you have to figure access controls. You have to figure out how do we take these credentials from PostgreS itself and vend it to all of our applications, and that's before you even get into like installing and configuring PostgreS. You have to think about instance types. What are the different performance requirements that we need? What performance options do some of these instances have?

For example, AWS offers instances that are called T2s, which burst CPU. They offer things that burst disk. So, finally, you have to tune PostgreS based on those kinds of instance types, and that’s an extremely involved complicated lots of trial and error process. Once you get your PostgreS instance up, how do we handle failover? How do we handle database backups? What do we do if the instance goes away because the hypervisor crashed? The list goes on and on, and this could obviously take up an entire hour. So I'll plan to give you an idea of what goes on there, and what I've said so far just scratches the surface.

**[00:10:19] JM**: That’s a great summer, and I'm looking forward to diving into some of those topics. To take an even more naïve perspective, there are some people listening who have probably never operated their own high-availability database, because they've always offloaded that to the cloud. I think it's worth even discussing why are we doing this? Why did we get to this point where we’re running a database in the cloud? Why is that preferable to the days of the 90s when we had our own servers and we operated our own database and we served the traffic to that database?

**[00:10:56] JD**: So it's all about risk, really. If you have your database running in your own server and it's managed by your IT people, there is a lot more trust that you know exactly what's happening. You can see the server. You can inspect the data center. You could do a lot of things, but what happens if that server crashes? What happens if that server is damaged beyond all repair? The CPU gets fried and you have to replace it? So now you have to physically go out and buy another server.

There is a question a friend of mine always asks whenever he's touring a data center for like a site evaluation, and that question is, “If I walk in here with chainsaws for hands and start cutting every cable I see, how long until my website goes offline?” That’s something you have to consider when choosing a data center.

We've had issues where a tractor-trailer loses its brakes and drives right into the power management systems of the data center. Thankfully nobody was hurt the one time that happened, but it does happen. If it is any physical data center that requires human access to make changes, what happens when things go wrong?

Also, now there is an inherent cost to all of the underlying administration. We have to make sure that the server’s operating system is up-to-date. We have to make sure patches are being applied. We have to make sure we have intrusion detection systems, backend access in case of emergency. If we can’t SSH into the server anymore, do we have helping hands? What do we have? Putting it in the cloud kind of abstracts a lot of that away behind an API. So now you don't have to worry about a physical server sitting somewhere and you don’t have to worry about the BIOS getting out of date. You don’t have to worry about the server just getting old and dying. You have a defined instance type. You tell Amazon or Google or whoever, “Here’s how much memory I need. Here's how many CPUs I need. Here’s so much storage I need. I don't care about the rest.” So it's just offloading things that I worry about.

But on the other hand, putting it in the cloud gives you may be a little bit less trust, because you can't physically see the server or physically inspect the data center. But that's also what contracts are for. So you have to change your perception of a few things. Frankly, personally, after doing this for almost 11 years now, I trust Amazon's data center operations and their security practices a lot more than myself running a server in a basement data center of an office building.

**[00:13:24] JM**: Indeed, and we have been able to build on that trust further by using the abstractions that we have been offered by AWS and building other services that have been built on AWS, and people are developing trust in those services. You have core infrastructure that has been built on top of AWS and people trust those providers that have built on top of AWS.

So we have this chain of trust, and trust can be articulated to some extent by the term availability. Availability is an abstract term, but it basically means can you interface with the software that you're trying to interface with? From a database standpoint, the database that is highly available, you will be able to interface with that database from the point of view of writing to that database, reading from that database, deleting data from that database. I guess that's write, basically. But there are circumstances that can change over time, where maybe a database is highly available, but it's only highly available from the read point of view. You might have the write point of view be degraded in quality.

Could you talk about the term availability and explain how that applies to cloud databases?

**[00:14:55] JD**: Yeah. So availability, like you've said, has a lot of different terms. There're a lot of thought processes behind availability, and it depends on your workload. I don't want to name names, but previous job I work for, it was an e-commerce provider, and their workload was predominantly read- heavy. Any writes were to our payment providers mostly. You bought a thing and now we have to make sure we can take your credit card.

So we defined availability in if you go to our.com and can see all the things we have for sale, that's available, and that came down to, in this case, it was MySQL. Can MySQL return data? So we had a lot of various read replicas that our developers were able to use. If we had some backlog in writes or anything like that, we weren't happy about it, but we were okay with that. That didn't necessarily constitute like a P0 incident.

But a lot of customers, especially working with things like Salesforce CRM, which is what a number of Heroku's customers integrate with, at least at the higher level, write is extremely important, because they're trying to like sync data between their CRM, which is their source of record and their PostgreS database so that their Heroku apps can perform whatever work they need to do.

So in that case, write availability is far more important, and that puts us in kind of an interesting position, because if we can spend up all of these database – We call them followers instead of the old terminology, so leader and follower. If we can spin up all these followers to make reads highly available, but writes fail, that's not helping the customer at all. S

So we kind of take a very sort of conservative approach to how we consider database available in all cases. If we see a backlog of queries, or too many locks open, or even health checks failing because they're failing to get a process available even if there are other processes currently returning data. We do consider that database to be unavailable. We run some automated remediation, and if that fails, we page an operator.

I'd say a lot of times it comes down to bad queries or things that are indexed incorrectly, or things that are suboptimal. But there are a few cases that we've seen where EBS, for example, will give out. They have had issues in the past where writes get queued and they can't hit their sand fast enough. We do consider that be an unavailability situation, and if it's a highly available database, we will trigger a failover hopefully getting it on to an instance that has a healthy EBS disk, which considering that we use separate availability zones for leaders and followers, that almost never happens. It's almost always safe. But it really depends on your application. When you're a kind of general-purpose cloud provider like Heroku, you have to sort of assume that read and write are both just as important.

[SPONSOR MESSAGE]

**[00:18:10] JM**: On Software Engineering Daily, we've had several shows about the future of technology education. We believe that boot camps are an efficient, cost-effective way to become trained for the tech industry whether you want to be a programmer, a data scientist, or a designer.

Flatiron School can teach you the skills you need to build a career that you will love. Flatiron School has immersive programming courses on JavaScript and Ruby, everything you need to become a full stack developer. If you're interested in becoming a data scientist, Flatiron School has courses on Python, SQL and machine learning.

You can learn in-person or online and you can find everything you need to get started by going to flatironschool.com/sedaily. Flatiron School has options to save money on the program, such as gender diversity scholarships and income share agreements. Flatiron School also helps the students who graduate find a job. Every graduate is paired with a dedicated career coach so that they can find a job or their money back.

The complete details are at flatironschool.com/terms. Flatiron School is a cost-effective way to start working in a tech industry. Learn more at flatironschool.com/sedaily.

[INTERVIEW CONTINUED]

**[00:19:40] JM**: Another term we should discuss here is consistency. When you talk about databases, consistency often means if you have different replicas or different materialized views of the database, to what degree are those views of the database the same?

Could you just talk about why consistency matters and the different circumstances where inconsistency can develop and whether inconsistency is a problem? When it's a problem?

**[00:20:15] JD**: Yeah. I'd say the vast majority of cases, if there is inconsistency, that is a problem. If you were to go to amazon.com and go to buy a new videogame, and I think the last game I bought on Amazon was Spiderman for the PS4, and one minute it's saying it has 25 copies available. You reload the page. It goes to a different database follower, and now it says it has zero copies available. You're going to be a little unhappy. That's going to be a confusing experience.

So consistency especially in PostgreS, when you have like ACI-compliant databases, is incredibly important. At Heroku, we use just native standard PostgreS replication, the thing that's built into the database engine that anybody can use, and it works most of the time almost always, I would say. There are certainly instances where it can fall behind. If you do a lot of writes, like really, really intensive database writes on your leader, you will start to see inconsistency problems, because replication just cannot keep up.

I'm not going to name names, because this happens, but we did have a customer one time that was essentially row-by-row deleting every record in their database and recreating it every single day, and that caused their replication to fall behind so badly that we did the math, it would've taken 20 some thousand years for it to catch up from a PostgreS streaming replication standpoint.

So these are things you have to kind of take into consideration when it comes to consistency, that every time you run one of these queries, that same query is going to have to pass over a network that may or may not be working at that present time, because the internet's a Wild West, and it will have to replay on a different server. That different server could have different qualities to it. It might be the same instance type, but it might be on a more noisy region or a more noisy availability zone. It might have more disk load because something else is happening on.

We've had cases where customers will use different instance types entirely for their leaders and followers, and that's also not great. We have some last-ditch efforts that we do in case of serious inconsistency. For example, if streaming replication fails so badly, we actually revert to restoring a base backup on the follower to get it active again, and then we replay all of its write-ahead logs to try to get it closed enough for streaming replication takeover again. But you have to just be mindful of every time you write data, that data doesn't – And just to that disk, there is a whole power of things that happen behind-the-scenes.

**[00:22:57] JM**: What you're describing here is the fact that when we start using databases in our developer 101 experiences, whether we’re hacking on a laptop, taking some coding school, or we’re in computer science courses at the University. Generally, our experience with the database is this thing that's sitting on my computer and it's just a single node instance and I write to it and I read from it and everything feels pretty simple. But in a production deployment, what you actually want from your database is replicas. You want a replicated database. You want a situation where when I write to my database, I'm actually writing to multiple instances of the same database, because if a meteor impacts the data center, I want to know that my data has been replicated to another data center and I can continue humming along and my users can expect availability from my application, because the data center that was not hit by a meteor is still intact.

So maybe you could talk a little bit about this idea of replication and how replication fits in with the other two terms we’ve discussed; availability and inconsistency.

**[00:24:19] JD**: Yeah. So replication is an easy concept in theory and very difficult in practice. The rather naïve view of replication is I run an insert statement on database A. Something happens, and that insert statement is magically run on database B.

In reality, it's essentially the same, but the implementation is a little bit different. But it's the basic idea that data on one instance will always be replicated or reproduced on that other instance, but there a lot of asterisks by that.

So if you have to databases physically sitting next to each other with a 10 gigabit per second fiber-optic cable connecting the two, you're probably going to be fine. Nothing bad will most likely happen. But when you're talking about like in the context of AWS, you have availability zones, which in some cases AWS is a little hush-hush on the underlying implementation. But in some cases they're physically separate buildings on opposite sides of a neighborhood. There can be – Whether it'd be copper or fiber leased lines running between these buildings, and that is essentially entering a much more high-speed version of the internet. It's entering a MAN or a metropolitan area network.

So you are going to have all sorts of things that could potentially go wrong. That data that you write on database A that is in the same physical availability zone as your app servers might have a 50 to 100 millisecond lag before it gets to database B, and that can cause your replicants to fall out of sync. There's an SLA involved. It's usually going to be in the order of milliseconds, but as you write more and more and it takes longer and longer to reach the other database and this replication gets backed up, you're going to get more and more behind.

So there's no magical answer to any of this. It's all about monitoring. It's all about operations. You need to make sure that you have the appropriate alarms in place to know when databases are falling out of sync and your operations team, whether it'd be someone like a service provider like Heroku or GCP that you're paying, or if it's your internal operations team your company. They need to be aware of how do I graph that replication lag? How do I discover how far back it's falling? How can I tell when this is just a momentary blip and it will catch up, because we had a big heavy write operation? How do we tell if this is something more serious? Is there a degraded network performance between two physical data centers? Is it one of the instances is failing? There are so, so, so many questions, but it boils down to like most things in tech. It's not magic.

**[00:27:11] JM**: There's an abstraction within databases such as PostgreS that I'd like to discuss a little bit, and this abstraction is the write-ahead log. I believe in PostgreS it's called the write-ahead log.

**[00:27:23] JD**: Yup.

**[00:27:24] JM**: And this is an abstraction that we need to achieve reliable replication. It's basically the cure for the lack of this – You could just copy the insert statement from one database to another, and most people who are interacting with the database never have to think about a write-ahead log. They never have to interface with something like this. Could you explain what a write-ahead log is and explain what we are doing with the write-ahead log in order to have replication?

**[00:27:55] JD**: Right. So you can kind of to think of write-ahead log in its basic sense as a file that contains all of the almost block on the hard drive level, but it's a little higher level than that. But block level DIFs that happen between point A and point B. You can configure PostgreS to automatically put out a write-ahead log every so many seconds or whenever it reaches a certain size. So like one megabyte, or whatever. But these are like the physical disk DIFs of point A and point B.

So this is kind of more powerful than using just copying the “insert statement”, because in that case, what happens if one of the insert statements fail on database B, but they succeed on database A? There can be inconsistency there. But replaying the write-ahead log from database A to database B ensures that at a like file system level these two databases are exactly the same.

Your primary or your leader PostgreS database every so often will spit out one of these write-ahead logs and it pipes it out to what you defined as your archive command in the PostgreS configuration file. So in a lot of the like mastering PostgreS administration books that you'll see, they'll talk about, “Okay, you can use rsink, or you can use copy, or FTP, or whatever to get this elsewhere,” and what we’re doing at Heroku is essentially the same.

We use a tool called Wally, which will connect to S3, and every time a wall segment is pushed, it'll actually push that into S3 to make sure that if the worst were to happen to your instance, you are safe. The reason we use S3 over other options is S3 does guarantee high-availability. We don't run things on like reduced availability. We can have all kinds of access controls in place to make sure that nobody can read anybody else's wall, and it allows us to potentially give that wall to customers if they want to do some kind of review or inspection or anything. That's not something we do very often though. So it has a lot of advantages, and it's a lot better than pgdump rsink to another database.

**[00:30:10] JM**: So that model of just writing the wall to an S3 bucket and you get replication that way, does that mean that you actually only need one running PostgreS instance because you don't need to have actual replication in other database nodes?

**[00:30:32] JD**: This is how we’re getting back – Or this is something that goes back to availability. In theory, yes. In practice, no. Because restoring a PostgreS is not ever instant. If you are building a basically Rails or Django app on your laptop and you’re trying to learn new things, your databases is going to be a few megabytes and it will be really fast. But if we’re talking about insert large e-commerce provider here, their database is going to be terabytes in size, and performing a base backup restore and then Wally – Or then wall recreation from S3 on that database will never be fast.

So it's about time. If you're willing to wait the hours or days to get your database back, which I doubt any company is willing to do, then yes, you can rely purely on S3. But having that like separate database ready to go, it’s the difference between an almost hot and cold standby. The hot standby, the minute the leader has a problem and has to go offline, that hot standby is ready to go. It can just start taking requests immediately, and relying on S3 is more of a cold standby. Your data is safe. You didn't lose anything, but getting it back is going to take a little while.

**[00:31:44] JM**: I’d like to start getting into more of these discussions about what it's like to actually implement a database service as a cloud provider. So if I log on to Heroku, I've got service I want to deploy. It requires a PostgreS database. What goes on in the backend when I click provision, or deploy, or whatever I'm clicking to spin up a database. What's happening on your infrastructure?

**[00:32:16] JD**: A lot. So Heroku at one point made this decision to kind of break away what we call runtime, which are your actual application processes and what we consider add-ons, which an add-on is something outside of a process running your rails app, for example. One of those add-ons, for example, would be a database.

So if you run Heroku add-ons create standard zero, which is our lowest here and cheapest paid plan, what happens is that API request that's sent to platform API, which then sends a request to the provider add-on, which in this case is Heroku itself. When we receive that request, we look up the plan type. We look up your account. We make sure credentials are in order. We make sure that you have a credit card on file that we can charge. But at that point, we grab an EC2 instance. Often times we have them ready to go. We have them – What we call Slack instances. So we don't want to spin up an entirely new one. We grab that instance. We do some configuration on it. Pretty much, we configure PostgreS to be the plan type you asked for. We make sure that appropriate discs are mounted. We make sure appropriate IP addresses are configured. We configure PostgreS based on the version that you requested.

Actually, every supported version of PostgreS that Heroku offers is installed on every single one of our instances, and that's just to make the general provisioning faster and upgrading your database later a little easier.

So once we have network, once we have disk, everything like that, we drop configuration files. We create you a database user. We create that database. We install extensions that are necessary, and at that point our control plane checks to see when this database is healthy, when a consent select one, for example, to the database and get an appropriate response back. Once that's done, we take those credentials and we export them through the platform API and then it appears in your Heroku config listing, if you run the Heroku config, like show all command, then you'll see it. At that point it's available.

**[00:34:19] JM**: Tell me a little bit more about the operational side of things. So you've got people spinning up these databases. What's required on your end to make sure that those things stay up?

**[00:34:34] JD**: Yeah. So from that perspective, we have a control plane that kind of acts as a giant like clock or like scheduler, and we have these different process types. Some check services specifically or PostgreS services specifically. Some check what we call a server abstraction, which is all things that make up a server. Others check specifically the instance, specifically the disks, specifically the network, and we do a series of tests there to make sure like, “Okay, TCP connections work. We can SSH into the box. We can run select one,” or numerous other health check queries that you can see in your logs if you look closely enough. We kind of evaluate all that to determine if an instance is healthy. We also do some preventative checks to see when is this instance going to potentially stop being healthy.

One of them is disk size. We advertise plan limits. So a particular instance will have, say, one terabyte cap. So what happens if you go to 1.1 terabyte? You have 1 terabyte provisioned, but does that .1 that mean that you databases is dead? No. We look at the instances and we figure out how fast is this growing, and if it looks like it's going to surpass that, we do provision extra disk space, but let you know that this is going poorly.

Another thing we’ll do is we have a bit of a wall cache, because the network backs up sometimes, and we can't just hold up your entire PostgreS database because we can't write to a wall directory, or wall S3 bucket. So we have the wall cache on the disk, and if that disk starts getting too full, then we get alerted and we can take preventative action to prevent these kinds of things. So there's more to the health of a database than just being able to select star from people or insert into blah.

**[00:36:24] JM**: Has this stuff gotten easier to do in the last – I don’t know, three or four years? Because I've spoken to a lot of companies who have, for example, migrated to Kubernetes, or migrated to a managed container service, and their life has become a lot easier. They don't get paged as much, because the way that the infrastructure gets architected, if it's built on top of these container services, sometimes these managed services, can make things a lot easier. But I'm not sure if that applies to something like a managed database, because that's kind of a different beast than like an e-commerce company that's got some microservices and got a single database and it's much simpler architecture. What you're doing as a little bit more technically intense. So to what degree has running a database cloud service become easier in the past few years due to some advances in fundamental infrastructure?

**[00:37:25] JD**: AWS has been a huge advantage, and I don't mean that the fact that AWS exists, but more of AWS's commitment to enhance monitoring, enhanced operations, enhanced API management. I haven't looked in Heroku, like my team at Heroku's AWS console in probably a year, because I don't have to, because if you can do it in the console, you can do it via an API call, which is pretty amazing. A lot of other enhanced monitoring has really helped. The fact that they can let us know ahead of time if a hypervisor is going to have problems. That's when you get the like your database requires maintenance emails that everybody loves from Heroku. Those are often imposed on us by Amazon.

But some of things that have made things easier, honestly, is the improvement of container technology. So the fact that – We actually don't use Kubernetes or Docker in the data team at Heroku, but we use LXC directly. The nice thing about that is being able to have fairly standard, almost single tenant configuration set up on that instance, and the configuration changes themselves as to like what version of PostgreS or what plan type is really based on mounting. So we know that everything is going to act very similarly from one to the other and there's not a lot of one-off configuration happening in there.

Also, advancements around performance. Really, the fact that Amazon is able to offer us kind of like enhanced provision priority networking, provisioned input-output operations on disks, faster provisioning times. All these things have made working with these services easier and also much, much faster to recover.

Prior to the introduction of VPC, whenever you had EC2 Classic or what they call EC2 Classic, it was a total Wild West. If somebody was port scanning you from – If you had a noisy neighbor or someone who's port scanning even the same AZ, there wasn't much you could do aside from complain to Amazon. But giving us this like secure isolated mini-AWS or mini-AWS ecosystem has really help from a managing standpoint, because  we can have a lot more trust that the only things happening in our ecosystem are things that we know about.

[SPONSOR MESSAGE]

**[00:39:47] JM**: Today's episode is sponsored by DataDog, a cloud scale monitoring service that provides comprehensive visibility into cloud, hybrid and multi-cloud environments with over 250 integrations. Datadog unifies your metrics, your logs and your distributed request traces in one platform so that you can investigate and troubleshoot issues across every layer of your stack.

Use Datadog's rich, customizable dashboards and algorithmic alerts to ensure redundancy across multi-cloud deployments and monitor cloud migrations in real-time. Start a free trial today and Datadog will send you a t-shirt. You can visit softwareengineeringdaily.com/datadog for more details. That's softwareengineeringdaily.com/datadog, and you will get a free t-shirt for trying out Datadog.

Thanks to Datadog for being a sponsor of Software Engineering Daily .

[INTERVIEW CONTINUED]

**[00:40:52] JM**: We've had a number of shows recently about these second layer cloud providers, which are cloud providers built on top of AWS, or cloud, or other fundamental cloud infrastructure, and Heroku was one of the first to start to develop in that market. But there's all these interesting problems for the second layer cloud provider industry. One that I think is interesting is the cost management issue.

If you're a software architect or a software engineer who's familiar with cloud services, it can look like an all-you-can-eat buffet of things that you want to work with. But if you're building a cloud provider on top of another cloud provider, your fundamental economics are really, really, really important, because you need to be able to offer a somewhat commodity feeling service to the user, your customer base. But you want your cost structure to remain low enough for that kind of commodity service to be at a defensible price. At the same time you're looking at all these like new fancy Amazon services and these are the building blocks for your infrastructure. You don't want to overspend on them. To what extent is cost management and issue the you're thinking about?

**[00:42:17] JD**: Right. So one of the first things to think about when you're a kind of cloud provider on top of another cloud provider is there's an episode of Seinfeld I think where Kramer talks about nobody pays retail, and that's very much true once you get to a certain size in Amazon. You have reserved instance types. You have prepay. They'll do like volume discounts if you're big enough, and having just your sheer size and like the weight of your organization behind you can allow you to make really good deals with Amazon to get things like lower-priced instances or you pay upfront for them or even just better support so you spend less time mucking about, so to speak.

The other thing is you kind of have to take into account of almost the opportunity cost of what you're providing to customers. I've seen numerous blog posts about like how I ditched Heroku and moved to DigitalOcean. DigitalOcean, they offer some really great products. But the kinds of things that they're going to offer with their PostgreS databases may not be the same. We are essentially taking on the role of your operations team, we're doing a lot of preventative maintenance. We're doing the a lot of just things that you would never think about behind-the-scenes, and that's sort of what we bake into our costs.

But there is a fine line. We need to make sure that our costs are competitive enough that we won't really fall into, "Oh! It's just cheaper for me to hire an operations person to make that more appealing," but also not sell ourselves so short that we're not making a lot of money. It's always a fine balance, and more often than not, it's your bigger customers that sort of dictate the direction that you go with a lot of your like higher-tiered plans. Because most customers at a smaller stage, if you're like standard zero to standard four, you're really looking for a fairly general-purpose database, and those are more like we're building cheaper cars here as opposed to like building a Ferrari, and it's when you get into the larger customer scale that you're kind of building that like more customized solution. Your cost is becoming less of an issue and it's more about the features you can provide.

**[00:44:23] JM**: You've shown in this conversation is a willingness to talk about other cloud providers, which I appreciate. So I'd love to get kind of high-level perspective for this like layer 2 cloud provider, layer 1 cloud provider dynamic that has developed. As a software engineer, it makes me really excited. Then kind of as a business analyst, it makes me really intrigued, because who would've thought that there would've been this variety of second layer cloud solutions that are built on top of other cloud providers so that the second layer kind of cloud resellers or however you want to call them, they have a really like dynamic flexible business, because they don't have to invest in this core server infrastructure and their life is more about like developer experience. It's like a design problem.

What kind of developer niche can I cater to? I mean, I've used Heroku for quite a long time because I am more interested in kind of the product level discussions and decisions and maybe like the frontend decisions. I don't really want to have to think about like did my write-ahead log get backed up today. So like Heroku is kind of a good fit for that kind of thing. Other people really enjoy getting their cost structures really, really low and don't really mind managing their own cloud infrastructure. So it's like, "Okay. Maybe you want to go with AWS or DigitalOcean." I mean, even the layer one cloud providers have their own subjective decisions, design decisions that they're making. So I'm just so optimistic about this environment, but maybe you could just give me your feedback from being a developer for a while and obviously working at layer two cloud solution. To some extent, I think the Salesforce has its own servers to some extent and there are some Heroku involvement there. But just tell me what your perspective is on this emergent dynamic of the layer 1 and layer 2 clouds.

**[00:46:31] JD**: I actually came from a primarily software development background before I made the move to Heroku, which is a little more infrastructure-driven. I'm seeing a lot of similarities between the evolution of a lot of – From machine code, to programming languages, to these like application frameworks. So I kind of think of running your own data center as being a C developer. You're working in the weeds.

It is full of a lot of very well understood problems and there's a wealth of knowledge the you can kind of pull from and build on the back of, but it's hard and it's easy to shoot yourself in the foot and it's easy to make decisions that you feel are the best ones at the time, but are very, very difficult to undo later if you find a more optimal solution.

I kind of look at things like AWS as being like the abstraction that they built on top of their data centers, AWS itself, to be like the Ruby programming language. Ruby was written in C. It utilizes a lot of the functions that C can provide, but it gives you a happier kind of interface to that. That was like a whole – Like Ruby is a programming language to make developers happy, and Amazon is sort of an abstraction to make system operators happy.

Most people don't like being in data centers. They’re loud. They’re hot. They're just generally uncomfortable. So the thought of, “I never have to go to a data center again. I click a button and I get my server,” is really appealing. Then on top of that, Heroku is more like Rails.

So, yes, there’s still Ruby involved. There’s still the concept of a server and of CPUs and memory and these things you have to take into account, but I don't have to think about session management now. I don't have to think about, “Oh no! How am I allocating objects in the most effective way?” I don't have to think about how do I build an abstraction layer around my database. These things are available to you and its batteries included. So building a Rails app, I don't have to think about building a web framework. I can just focus on building an application that provides value to my business, and using Heroku is similar. You have to think about all of the things that come with spinning up a server. You just focus on building your apps.

**[00:48:35] JM**: Do you have any war stories from your experience of helping to build out this large-scale managed database service? Like any outages that come to mind, or maybe not outages, maybe just difficult technical problems that were really hard to sort out.

**[00:48:55] JD**: I have some. I wish I could go into detail about a few, but I can't unfortunately. I always assume everybody's doing the best job that they can, and that is a really, really difficult thing to keep thinking of when you have some of these problems that turn into war stories.

I was not at Heroku when the great U.S East One outage happened a while ago, but I was using a different cloud provider, Engine Yard at the time, and that felt like the world had ended. We had a customer issue a while ago that I can’t get into details on, but it involved their database being down for a nontrivial number of days, and there wasn't much we could do, because they had backed themselves into such a corner.

Those are always a struggle having to sit in front of a customer and tell them that this is not going to be fixed anytime soon and we're doing everything we can is always tricky. Also, it's important to remember that software is dynamic. We see a lot of people coming to us needing help, and the first thing they always say is we didn't change anything, and that is a certainly useful data point. But it's not really vindication either. If I don't put gas in my car for two weeks and I start driving to a different city, my car is still going to be broken down. I haven't changed anything, but something bad still happened, and it's just a data point. Sometimes customers kind of make this assumption that like, “I run my database on Heroku and I never have to think about it again, and I can just throw whatever I want at it and it'll just deal with it,” and that's really not true. These are just servers in the background.

So there's a little bit of back-and-forth that kind of has to happen, and it's important to remember, when these things happen, we as Heroku, we want you to be successful. Nobody's out to get you. Nobody's trying to be a jerk support engineer or jerk engineer, and sometimes when you don't hear a response back right away, it's because we’re just as confused as you are. We’re not like absolute database masters over here. There are certainly some people that know more than others, but we’re all just developers trying to do our best.

As far as large-scale incidents too, it happens. We've had full platform API outages before that sometimes redo the mistakes we've made sometimes due to higher-level issues, and you just have to do your best and you have to be calm. If you're a service provider like Heroku, we’re upset when this happens, but you just have to take a deep breath and deal with it.

If you’re a user, if you're running your apps on Heroku and your business is down, we feel your pain just as much as you do and we want it to be back so badly, but also be calm and know that we’re trying our best.

**[00:51:40] JM**: S3 goes down sometimes. S3 has gone down. S3, the entire service, that underlies so much of the internet at this point. Every time that kind of thing happens, it scares the crap out of me. It's terrifying, and nobody wants that to happen. As you have already spoken of, Amazon is the most – Basically, the most reliable cloud provider at least in terms of data that we have probabilistically speaking, and it still goes down sometimes, because these are really, really hard problems. They’re getting solved over time and there's a lot of cause for optimism.

But what you said about like sometimes the customer can shoot themselves in the foot and backed themselves into a corner, like if I do – I can totally do something with PostgreS. If I do something like I try every entry to the write-ahead log. I decide to wipe out my database and then reconstruct it from the write-ahead log. If I just have some trigger that's trying to do that, I'm probably going to overwhelm – You can do these infinite loops, infinite recursive, overwhelm the database very quickly loops.

**[00:52:53] JD**: You'll have a bad time.

**[00:52:55] JM**: You'll have a very bad time. For you, as Heroku, in general, you want to prevent people from shooting themselves in the foot, but you can't account for every kind of user shooting themselves in the foot behavior. To what extent do you want to design the database service to account for people who may be shooting themselves in the foot?

**[00:53:18] JD**: Yeah. We have some preventative monitoring in place and we will let customers know to the best of our ability when we detect bad things. Some customers may have seen these very handcrafted artisanal emails usually from a support engineer at three in the morning that are, "Hey, we noticed you're doing this really weird thing and you're write-ahead log disk is full. We can't push it to S3 fast enough." You're going to have a bad time if you don't stop doing whatever you're doing or work with us a little bit here.

So those are tricky. We have some improvements that are coming down the pipeline that are going to kind of give customers a little bit of a heads up on other data corruption or potential backlog of archiving takes place. Also, we do make tools available to customers. So if you've ever run the Heroku help PG or Heroku PGPSQL commands, there are actually a whole lot of assistive tools in there as well.

One of my favorite that I actually use all the time is PG diagnose, and that will actually run a series of basic test against your database and give you a really nice sort of ASCII printout of everything that it detects that could be a problem. So your most expensive queries, indexes that are rarely used but written too frequently. It'll even give you like index cache hit rate so you can kind of re-architecture queries a little bit better.

We would like to eventually make this a little more preventative and let you know when we detect these kinds of differences, but even if after a deploy, maybe 15 to 20 minutes after one, you just run PG diagnose on your instance. That'll give you a good bit of information about what could potentially be going wrong and what could be improved.

**[00:55:07] JM**: Now, you don't work on Kafka or Redis, but you have colleagues within Heroku who are working on these managed services. These are, to some extent, similar to PostgreS, because it's a manage storage service that you have to maintain high-availability for. You have to maintain a simplified surface area to allow the developer to operate their own storage service for all the difficulties of operating your own replicated PostgreS cluster. I think Kafka probably puts it to shame with how difficult it is to operate. Do you have thoughts for what it's like to operate those other services to manage a distributed system of Kafka or Redis in contrast to something like PostgreS?

**[00:55:59] JD**: Yeah, they have their ups and their downs. I've worked on both of them a little bit. Redis hasn't really been getting a lot of attention lately unfortunately, but a lot of those operations are very similar to PostgreS with the biggest advantage being that restoring a Redis backup tends to be a lot faster, because it doesn't have all of the bells and whistles in place of PostgreS, because Redis is really more of a cache or a temporary place to put your data.

So basic day-to-day operations on Redis are almost exactly the same. We even have a bit of a joke that the tool that we use for PostgreS real-time database backups is called Wally, and the tool that we use for Redis is called EV. So it's from the Disney movie.

Kafka, that's an interesting beast. So people who haven't ever run Kafka themselves may be surprised to learn that Kafka itself isn't actually just a single service. It's actually two. There's Kafka on the frontend, which is your message broker, and then there's Zookeeper on the backend, which handles a lot of the distributed locks and the orchestration and the kind of raw storage of your data.

So when you spin up a Kafka instance in PostgreS, you don't have to think about how many Zookeepers do I need or things like that. We handle that for you, but that really means that spinning up a Kafka instance means we're managing two different services, and each of those services are running on a non-one number of servers.

So a lot of our smaller Kafka plans run on three Kafka brokers and then five Zookeeper instances, I think, which is part of the reason why Kafka plans tend to be more expensive. Then you get all the way up to like eight or nine Kafka brokers and five zookeeper instances. It can be pretty intense. One advantage to Kafka though is it's more distributed nature. If you have a PostgreS primary that goes down, you're in trouble. Even having failover kickoff, it's not instant. It takes 30 to 45 seconds, give or take. But if you have a Kafka broker go down, you can fix that pretty quickly. Request just stop going to that broker and the other ones pick it up. If the broker that went down is a controller, then there is a leader election to pick a new controller, which is the kind of leader Kafka instance. The general disaster recovery operations of Kafka are much easier, but the overall orchestration and moment-to-moment management of Kafka is a bit trickier. So it's all about tradeoffs.

**[00:58:33] JM**: Well, it’s been a great conversation. Let's just wrap up. Are there any elements of working on large scale cloud provider infrastructure that have surprised you that are memorable?

**[00:58:46] JD**: Yeah. I was a Heroku user for years before I joined the company, and back in 2011 when I deployed my first Rails app to Heroku, it seemed like magic to me. As I got to peer behind the curtain of Heroku a bit, it's extremely impressive that it all works and it still feels like magic sometimes. But these are all just servers and networking and ultimately people that are no dumber or smarter than you. Many of whom come from the same background as you who are just doing their best, and that knowledge that we have just as many rough edges we make just as many mistakes as you. Sometimes we are just as confused as you and bad things happen. That was extremely surprising to me, especially coming from someone who works in like smaller northeastern rustbelt tech companies where we think of like the Silicon Valley as this like shining hill and that's where Heroku is based out of. But just seeing the general human element of everything was really surprising and very delighting to me.

**[00:59:54] JM**: Okay. Well, John it’s been great talking. Thanks for coming on the show, and I remain a big fan of Heroku. I continue to use it to run many of my application. So, nice work.

**[01:00:03] JD**: Thank you. Thanks for having me.

[END OF INTERVIEW]

**[01:00:08] JM**: GoCD is a continuous delivery tool created by ThoughtWorks. It's open source and free to use, and GoCD has all the features you need for continuous delivery. Model your deployment pipelines without installing any plug-ins. Use the value stream map to visualize your end-to-end workflow, and if you use Kubernetes, GoCD is a natural fit to add continuous delivery to your project.

With GoCD running on Kubernetes, you define your build workflow and let GoCD provision and scale your infrastructure on-the-fly. GoCD agents use Kubernetes to scale as needed. Check out gocd.org/sedaily and learn about how you can get started. GoCD was built with the learnings of the ThoughtWorks engineering team who have talked about building the product in previous episodes of Software Engineering Daily, and it's great to see the continued progress on GoCD with the new Kubernetes integrations. You can check it out for yourself at gocd.org/sedaily.

Thank you so much to ThoughtWorks for being a longtime sponsor of Software Engineering Daily. We are proud to have ThoughtWorks and GoCD as sponsors of the show.

[END]