EPISODE 495

[INTRODUCTION]

[0:00:00.3] JM: Kubernetes has become the standard system for deploying and managing clusters of containers, but the vision for the project goes beyond managing containers. The long-term goal is to democratize the ability to build distributed systems.

Brendan Burns is a cofounder of the Kubernetes project, and he recently announced an open-source project called Metaparticle, a standard library for cloud native development. Metaparticle builds on top of Kubernetes primitives to make a distributed synchronization easier. It supplies language independent modules for locking and leader election as easy-to-use abstractions in familiar programming languages.

After decades of distributed systems research and applications, patterns have emerged about how we want to build these distributed systems. We need a way to lock a variable so that two nodes will not be able to write to that variable in a nondeterministic fashion. We need a way to do master election so that if a master node dies, the other nodes can pick a new node to orchestrate the system.

We know that just about every distributed application needs locking and leader election. So how can we build these features directly into our programming tools rather than bolting them on or outsourcing these to another tool? With Kubernetes providing a standard operating system for distributed applications, we can start to build standard libraries that assume we have access to underlying Kubernetes primitives. Instead of calling out to external tools, like Zookeeper and etcd, a standard library like a Metaparticle will abstracts them away, and that's really what's so exciting about Kubernetes is that you have this standardized distributed operating system, and you could start to build stuff on that and expect that other people are going to be building on top of the same standard operating system.

An example is if I'm writing a system to do distributed map reduce, I would like to avoid thinking about node failures and race conditions, and Brendan's idea is to push those problems down into a standard library so the next developer who comes along with a new idea for a multinode application has an easier time.

Brendan Burns currently works as a distinguished engineer at Microsoft and he joins the show to discuss why it is still hard to build distributed systems and what can be done to make it easier. This is the second time we've had Brendan on the show. The first time he came on he discussed the history of Kubernetes and some of the design decisions of the system, and this episode was more about the future. It's really exciting to talk to Brendan because he has such ambitious futuristic ideas about where we can take our computing systems.

Full disclosure; Microsoft, where Brendan is employed, is a sponsor of Software Engineering Daily.

[SPONSOR MESSAGE]

[0:03:16.1] JM: Azure Container Service simplifies the deployment, management and operations of Kubernetes. Eliminate the complicated planning and deployment of fully orchestrated containerized applications with Kubernetes. You can quickly provision clusters to be up and running in no time while simplifying your monitoring and cluster management through auto upgrades and a built-in operations console. Avoid being locked into any one vendor or resource. You can continue to work with the tools that you already know, such as Helm and move applications to any Kubernetes deployment.

Integrate with your choice of container registry, including Azure container registry. Also, quickly and efficiently scale to maximize your resource utilization without having to take your applications off-line. Isolate your application from infrastructure failures and transparently scale the underlying infrastructure to meet growing demands, all while increasing the security, reliability and availability of critical business workloads with Azure.

Check out the Azure Container Service at aka.ms/acs. That's aka.ms/acs, and the link is in the show notes. Thank you to Azure Container Service for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[0:04:42.8] JM: Brendan Burns is a founder of the Kubernetes project and a distinguished engineer at Microsoft. Brendan, welcome back to Software Engineering Daily.

[0:04:49.9] BB: Thank you so much for having me. It's great to be back.

[0:04:52.5] JM: We are at the beginning of some crucial changes in the way software runs, and I'd like to map out some of the ways that Kubernetes has altered software, and then we'll talk about what changes are likely to occur in the future. So going a little bit back in history to 2013, Docker helped solve packaging and deployment in distribution and isolation. It also helped us think about the bounded context of the different components of our application, but we still needed an orchestrator. Why was that what? What were the operational problems that were unsolved where we still needed an orchestrator?

[0:05:30.9] BB: I think there's a couple of different pieces. One is that in sort of packaging things up and hopefully decoupling them from a specific operating system or a specific machine, you need something that is been going to go in place that container out on to a machine somewhere, just like your operating system doesn't prompt you for which core you want to run an application on. It just finds a core and runs it. We don't want to have to force people to think necessarily about where their application is running. We just want a system that knows how to run it for them.

Additionally, of course, as we've moved from applications that are running on individual machines, to applications that are running in the cloud that are supplying APIs that have always on availability, we need — Have a much greater need for things like redundancy and reliability and horizontal scale. So those are all problems that are difficult for a user to figure out how to solve, but they're all problems that an orchestrator can come along and solve for you.

So in effect, the container packaging is great for the single machine, but the truth is that the systems that we need to build, support APIs and all was on sites, required distributed systems, and so you need an orchestrator in order to build and manage the distributed system.

[0:06:48.8] JM: This year at Cube-Con, Kelsey Hightower came out on to the stage and he was proudly announcing that Kubernetes was becoming boring, and I think he meant boring in the same way that we think of Ubuntu being boring. You want your core infrastructure to be boring. What were some of the struggles of getting to that blissful state of boredom that we're getting to today?

[0:07:12.8] BB: I think there's a couple of different things. I think one is that in the early days, it was a lot of teaching people why they may want this, or I think people, for a long time, their applications were so tightly bound to the operating system. So tightly bound to shell scripts or some of the older ways of deploying software that it was almost unimaginable that these things would be separated, and the orchestrator is that abstraction that separates you from your operating system, that separates you from your hardware. For a lot of people, I think it was exciting not because it was inherently exciting, but it was controversial, because it was a different way of doing things. So I think in some ways we had to set a lot of time talking to people about how and why they might want to do stuff.

I think there's a lot of listening that had to happen, because if you look at things like replica sets where everything was homogenous and really had no identity, and over time we listen to the challenges that people had in deploying stateful applications, like Mongo, into these homogenous environments. Overtime we develops things like of the stateful set, which sort of provides a little bit higher level abstraction, a little bit more coordination between various replicas and it makes it dramatically easier to deploy the stateful application like MongoDB. That was a compromise position, and I think part of what had to happen overtime was enough people had to use it, raise up all of the pain that some decisions that were made sort of out of a sense of API purity or a sense of design purity had to be sort of relaxed and rolled back in order to be practical and support ease-of-use for the end-user, because at the end of the day if people don't like it and don't want to use it, then it's kind of pointless to build the system.

I think there was also a sense that this area in the software stack was going to be somehow important at the business level, and I think what we're seeing is as Kubernetes becomes available in every single public cloud, for free effectively. It's turning into a commodity, and so I think there is no longer a sense that people are trying to own the space for their business, but rather it's just something that needs to be supplied for the things the people build on top. Certainly, from a cloud, from an Azure perspective, we really view the Kubernetes API as the beginning of the user experience, not the end of the user experience.

[0:09:37.1] JM: I’d like to talk about some of the ways that people are running applications today, and perhaps some of the patterns that we’re centralizing on. I know that you have a desire to see more patterns and perhaps more standardization, because the sooner we get to some patterns and some standardization, the easier this will become to proliferate. Let's talk about state management, for example. So for a while we've had this notion of the 12 factor app and most of our state, we’ve wanted to write to external databases or we write it to an external Redis cluster, but we don't actually think about managing state within the container itself. We just think of the container as this thing that can die at any time. Is that changing? Are we going to a place where the state in the container, we don't need to consider it as ephemeral as we once did?

[0:10:32.3] BB: I think you certainly don't have to considered it ephemeral, right? I think we do believe generally and in disaggregated storage, meaning that computer and network big storage are bound at runtime. You’re not particularly bound to anyone machine. You have a network file share that you mount into your container and that network file share follows you around the cluster.

We are seeing in the extreme, some extreme use cases, where local storage becomes really important. I guess it's always a trade-off. It's like how closely do you want to manage your storage? If you manage ait very closely, you take on a lot of application level responsibility for application level sharding and data replication and things like that. That can be really important if you need the performance that you can get from a very finely tuned stack. But in many cases, it's way more work than somebody really ought to be pursuing. Then we’re seeing people say, “Okay. Well, if I just use network attached storage, well it's easier because it follows me around the cluster and we’re bust to machine failures. Performance might not be quite as good as if it's local, but it's good enough.

Then I think for a lot of people, using cloud providers, storage as a service, something like CosmosDB in Azure that provides a Mongo API or a bunch of others, the Cassandra API, where they carry the pager and they are the ones responsible for all of the DRI. That makes a ton of sense to me. I have always said to every single team I’ve ever had, like you shouldn't be in the storage business generally. It's complex. It's hard and usually it's not worth it. Now, sometimes it is, but most of the time it's not. It’s not a question I guess of what is possible. It's more a question of what is most efficient for you and your team and the use case and business that you're trying to build.

I think generally speaking, the less state you keep in the application, the easier it is for people to build reliable applications, and thus you can sort of have people who are less distributed systems experts building applications that are robust to a lot of failures. I think an early failure mode is the sort of like, “Oh! Look how easy it is to stand up Cassandra in Kubernetes. Let's go to production with that.” It's very hard to measure like the cost of operating something overtime.

We do a good job of measuring like, “Oh, it was hard to set up,” but we don't do a very good job of measuring, like, “How much is this going to cost me every single week, every single month to keep this thing up and running, keep it okay?” I would say, generally, I would recommend most everybody. Especially if you’re in the cloud, use a storage as a service. It's just going to be a happier path.

[0:13:10.7] JM: Continuing the conversation of data management, how durable does the data in etcd need to be? And do we need version etcd? Do we need to keep old records so that we could roll back to an earlier snapshot of our etcd configuration data?

[0:13:29.2] BB: I think there’s a couple of different answers to that question. I think that DRI is always a good idea, and we’re starting to see tools getting created in that space to do —

[0:13:39.5] JM: What is DRI?

[0:13:40.5] BB: Disaster recovery.

[0:13:41.6] JM: Oh, yes.

[0:13:42.7] BB: Disaster recovery is always a good idea. Like it’s sort of a you — You should always have multiple ways back. On the other hand, I would also say that you should be treating the data in etcd as a reflection of something that you have stored in source control. There is no good —

[0:14:01.0] JM: YAML files or other [inaudible 0:14:03.0] files.

[0:14:02.9] BB: Yeah. There's no good reason to treat the etcd database as the source of truth. It should be a reflection of what's in source control, because otherwise you've lost at some level the ability to do code review. You've lost the ability to have audit trail — Actually, we do have some degree of audit trail for changes into etcd, but you want the edits to flow effectively from people editing text files and code reviewing text files and probably running through some sort of CICD validation system so that you validate that the config is sane before that configuration is pushed out to the etcd cluster in the Kubernetes, in the Kubernetes cluster, because that's just the right practice.

I think you want disaster recovery. You want to treat the data as being relatively important, but I think that the processes that we should be using mean that you should also have that data in a bunch of different other environments as well, and that you shouldn't really believe that the etcd piece is the source of truth.

[SPONSOR MESSAGE]

[0:15:12.2] JM: The Casper mattress was designed by an in-house team of engineers that spent thousands of hours developing the mattress, and is a software engineer you know what kind of development and dedication it takes to build a great product. The result is an exceptional product when you put in the amount of work and effort that went into the Casper mattress. You get something that you'd use and recommend to your friends and you deserve an exceptional night's rest yourself so that you can continue building great software.

Casper combines supportive memory foams for a sleep surface that's got just the right sink and just the right balance. Plus its breathable design sleeps cool to help you regulate your temperature through the night. Stay cool, people. Stay cool.

Buying a Casper mattress is completely risk-free. Casper offers free delivery and free returns with a 100 night home trial. If you don't love it, they'll pick it up and give you a full refund. Like many of the software services that we have covered on software engineering daily, they are great with refunds. Casper understands the importance of truly sleeping on a mattress before you commit, especially considering that you're going to spend a third of your life on that mattress. Amazon and Google reviews consistently ranked Casper as a favorite mattress, so try it out. Get a good night's rest and up voted yourself today.

As a special offer to Software Engineering Daily listeners, get $50 towards select mattress purchases by visiting casper.com/sedaily and using the code sedaily at checkout. Get the select mattress purchases if you go to casper.com/sedaily and enter the code sedaily at checkout.

Thank you, Casper.

[INTERVIEW CONTINUED]

[0:17:18.0] JM: Are there standards around how many Kubernetes clusters of companies is going to deploy given different circumstances? Like if I'm a two-person company, probably I only have one Kubernetes cluster and all my different services are managed in that cluster, but if I've got 100 person company or thousand person company, it's probably many different Kubernetes clusters within the company. Is that right? What are the best practices you've seen around how many Kubernetes clusters a company will deploy?

[0:17:50.2] BB: Yeah, I think it's kind of a trade-off. I mean in some cases, especially in big companies, you talk to them and they're just not set up to do accounting for a shared resource, like a shared cluster. They don't know how to do chargeback. So if you turn up a large cluster on a thousand VM's, well who's thing for those VM's? Also, at the end of the day, what business unit is paying for those VM's and how do you do chargeback to the various people who deploy stuff on to that cluster? That's something that is not very well handled, and some companies are willing to do that, especially if you're a startup or you're willing to say like, "We just have infrastructure costs. We're going to pay the infrastructure costs and all of our teams use that infrastructure and that's the way it is," and that sort of leans towards one cluster. But if it's difficult and then in your paying when you run one of these things in the cloud, you're paying based more on physical. You're paying based on the machines that are part of the cluster. It can be difficult to do the right chargeback, and so people run multiple clusters, because they want to have a cluster for a team, because that's their unit of costing effectively.

I think the advantages of having a small number of clusters is you can get all of your best practices into those clusters. Monitoring and alerting is very consistent. If you have a thousand clusters, it's kind of like having a thousand VM's. They're all going to be snowflakes that kind of look a little bit different and have you may be different monitoring plug-ins installed or are running different versions of Kubernetes. The more you have of something, the less consistent they're going to be, unless you then build automation on top that sort of forces them into consistency. I think that that's sort of the trade-off.

Obviously, also if you have multitenant clusters, you need to start worrying about multitenant security, and Kubernetes has things like RBAC that helps somewhat network policy in RBAC, but containers are hostile, multitenant secure. So if you have your payments processing system sitting next to your web UI, maybe that's not comfortable for you and maybe you want to actually have a separate cluster for, really, private stuff are really important stuff versus the more utility stuff. So I guess I'm not sure that there's a single best practice. I think there's a bunch of axes that either lean towards one cluster or lean towards lots of clusters and you kind of have to figure out where your business and your experience fits on all of those to decide how you do it.

I would go into it intentionally, I guess, is the only other thing I would say. You should go into it with a plan. You shouldn't wake up one morning and suddenly be like, “Oh gosh! We have fee of 500 clusters around the company. How did that happen?” or vice versa. You shouldn't wake up one morning and be like, “Oh my goodness! There's like 25 teams on this cluster that I turned up for my one app and I have no idea how to manage all the resource usage and they are effectively getting a free ride on my budget,” and that sort of thing. I think it's definitely very important to go in eyes wide open and decide like, “Are we going to be a cluster per team kind of environment or are we going to be a one cluster for the company kind of environment?”

The great thing is, with cloud services like Azure container service, it's really easy to turn up clusters, and so we do see things like people will turn up a fresh cluster for CICD and then just tear it down at the end of the CICD run. Definitely having your cluster as API is a really useful primitive.

[0:21:06.8] JM: We had Joe Beda on the show recently and he was talking about how the future is going to be very plug-and-play, and there is some plug-and-play ability today with things like Helm and cube apps and it's becoming much easier to just one click install anything to your Kubernetes cluster. I know you've talked about making distributed systems as easy to build his mobile applications. How far is Kubernetes from that world that you would like to see today?

[0:21:38.3] BB: I don't think Kubernetes is ever going to be taken to that world. I think that we're going to build on top. We're going to build experiences on top of Kubernetes to produce that world. I don't think we ever — I think to layer that Kubernetes is at right now is the layer that we want to leave it at. It's in the extensions that we build on top and the systems that we build on top that sort of power will come from. I kind of always view Kubernetes as the assembly language of distributed systems. We're going to build higher-level languages that compile down to this lower-level assembly language, if you will.

I don't actually know if you had a chance to take a look at my Cube-Con talk, my Cube-Con keynote, but I introduced this new system called Metaparticle that I've been working on for a while. It's at metaparticle.io.

[0:22:25.2] JM: I didn't see your Cube-Con keynote, but I read your medium post about Metaparticle.

[0:22:30.5] BB: So this is even newer than the medium post. That's was a part of this. That was a part of the — But if you go to metaparticle.io, there's what I would call a cloud idiomatic environment, which is to say, I think that rather than thinking about deploying a distributed system being separate from writing a distributed application, we need to start thinking about how do we actually integrate cloud into our programming languages. I think that what we've seen over the course of history, if you look at something like locking, locking starts out not really existing. It's something that you do via a complicated series of assembly language instructions. There are no locks. It's just you use compare and swap and there's this value that you figure out and it locks. Then over time, it becomes a library, it becomes POSIX, it becomes a library that you can link into your application and use so that you don't actually have to understand the intricacies of the assembly language. Then eventually it becomes a keyword. Synchronized in Java as a keyword. It's part of the language.

C++ doesn't really understand about locking. It has libraries that you can use to implement locks, but it doesn't understand locks. Java understands locks at the programming language level, and I think the same thing will be true with cloud, and I think with Kubernetes we've sort of built out a bit of an assembly language and we're starting to see some libraries developed, but eventually this stuff is going to be the sort of thing that you express in a programming language. You might say something like, "This is a cloud thread, and if it's a cloud thread, I want it to not run on this machine. I want it to go find a location in the cloud and execute and I better not have any side effects, because state management in the cloud is hard.

I think that you're going to have to — This int is a cloud integer, which means please store it to a durable backing store. Please make sure that you do transactional updates. Throw exceptions if you can't assign a value. Whatever it happens to be. I think that's the only way we move forward, and I think we're going to become come to a world where programs don't just sort of run on the cloud, programs expect the cloud.

[0:24:36.2] JM: Another example of this type of change of how we would write our applications might be the intent-based APIs that I've seen you talk about, and this is basically the idea that you would describe your API requests declaratively. So instead of, for example, saying I want my application to make a request to a load testing service and have that load testing service barrage my application, you would have somewhere in your declarative configuration make this service be load tested as an aspect of the service.

Do you agree with that? Is that a good example of this?

[0:25:19.7] BB: Yeah, I think that's another example of being able to — I think in general, what we're trying to do is build the ability for people to assert things at a higher level. You can conceptually understand what a load test is without having to understand the details of how that load test might be implemented. For most people, that's going to help them. If you sit down and you say, "Gee! I should really load test my app today." If all you have to do is to find a YAML that defines the rough properties of what it means to load test your app and it just happens, well, we're all going to do that. But if you have to go research like, "Oh, what is the best load testing framework and how do I configure it and how do I deploy it?" Even if it's a webpage that you read through and you run 20 different shell commands, that's a barrier to entry and it's not a very smart barrier to entry, because like what value is it that you learn how to use this specific load testing framework to run a load test? At some level, you don't care. You just want to see the results.

I think that's definitely an important part, building abstractions that are at the right level, so that people can focus on the important aspects without getting lost in the details. In some ways, that's what programming languages have been doing since the beginning of time. So I think those developments are going to be crucial over the next set of years as we become more cloud native in how we build our applications.

[0:26:45.9] JM: So Metaparticle is a kind of another example of this like you said. Basically, as I understand, Metaparticle is — If you want to make a distributed application today and you want to have a in-memory storage system and you want to maintain the kind of 12 factor app things with external storage today, you might use something like Redis, but what you're advocating with the Metaparticle system is you don't actually want necessarily to make a call out to Redis, because that's kind of messy. You want to have a language level abstraction where you just like you have no atomic variables. You would have maybe like a distributed persistent variable. Is that correct?

[0:27:37.7] BB: Yeah. In general, I think that's an example of what I would say is the broader goal, which is that you want to write your applications in the cloud and have them look and feel like they are part of the programming language. Not like they’re an add-on or a library or a thing that you happen to link in that the programming language doesn't understand. You want it to look and feel like it is a first class citizen within the programming language, because I think that it's that important. I think that what — And I think you saw this with synchronization. Like if you look at all of the major or at least a large number of the major languages that have come out in the last 10 to 15 years, they all have synchronization as a primitive. There’s channels in Go. There is the synchronized primitives in languages like Java and languages before then, because most computers were single core computers and that sort of thing didn't really think about this very much. They didn't think about multiprocessing. They didn't think about multithreading. Then all of a sudden in the sort of middle, late 1990s, suddenly concurrency becomes way more important and languages start having a surfacing of a lot more primitives to help people deal with concurrency, or like JavaScript where node says like, “It's all event driven. You’re going to learn this event driven model. There's only one thread. You're going to pass callbacks all over the place, and then we’ll develop promises and things like a wait in C# and typescript.” All of these language level stuff to support synchronization.

I think that what you're going to see over the next 10, 15 years is language level support to support cloud, to support distributed systems, and because it has to, because it's just too hard. It is just way too hard to build a cloud application. It's too hard to do storage. It breaks the flow. You've taught someone about storing something in a variable and then suddenly you tell them like, “Oh no! Actually, variables no longer really exist in the cloud.” Variables are these kind of a local thing and assigning to a local thing. It doesn't really matter. You need storage system somewhere in order to actually store that variable. It just doesn't — Why introduce that barrier? We already have the equal sign? Why are we saying you have to say set instead of using equal sign? It doesn't make sense. It makes sense, because you understand the history and you understand where it came from and you understand that things have to go through this progression. They have to go through a progression from being a library to being a part of the language, but I guess what I'm trying to say is like we are moving from the, “Hey, it’s a library. You need to understand how to use,” to “How do we figure out how to put it into the language?” There are other examples of this. I mean I'm not the first person to talk about this stuff. You can look at Hadoop as the first example of one of these systems where Hadoop basically says, “Hey, if you implement these two Java classes, I'll go use a thousand computers on your behalf.” The person who implements those Java classes doesn't really understand what's going on. I mean like sort of do, but like not at a deep level. They just know, “Hey, if I implement this map thing and if I implement this reduce thing, somehow magic happens and my word count runs in a 100th of the time it took otherwise.

That's usually powerful. That's democratizing that distributed system. In fact I would argue the big data as an industry wouldn't exist without that kind of abstraction. You're allowing all these people who aren't distributed systems people to suddenly harness the power of all of these computers.

[0:31:12.0] JM: Yeah, absolutely, and people build tools on top of those and turn map produce into a single one-liner command that you just import and use as SaaS service somehow, and in order to get to that place we needed to have those nice little abstractions to build on at the bottom.

[0:31:30.0] BB: I think you've seen this happen as well. So there's this paper recently out of the amp lab, which is in Berkeley, which is actually where Mesos came out of as well. It was called this system called, the system was called [inaudible 0:31:40.3], and they basically took the map function in Python and they backed it up to functions, functions as a service. So suddenly anybody can write a Python map expression can distribute that across functions as a service and get massive throughput for the computation.

It's so easy to take teach someone like what that function does, and then if they can just automatically unlock the cloud to make it run faster, that's a huge win. Again, it's fitting it inside the contours of the language. You don't teach anybody about functions as a service. You don't teach anybody about cloud even. You just say like, "Hey, if you use this thing, that little piece of code that you run will run a hundred times faster."

[0:32:28.8] JM: And Metaparticle, did you kind of write this like just an example or do you think this is like an actual project that will have a route into production systems?

[0:32:40.7] BB: The Metaparticle project that I just launched at Cube-Con, which is bigger than just the storage piece. That actually includes some concurrency. It includes automatic packaging of the application as a container and deployment of the container to Kubernetes. I don't know if it will be the system that is the canonical example of this. I think we have to work together a lot as a community and discuss these ideas. I guess what I'm hoping is that it becomes a place to build and a place to experiment moving forward, and if in two years we look back at all the different pieces that we've built and we say, "Oh, you know what? Actually, we can understand the space a lot better now and having that understanding. We're going to go build from scratch a new system and maybe give it a different name." I'm not going to be sad about that.

On the hand, maybe it will slowly evolve into being the production system. I think these things take time to evolve, and I mean a lot of languages spend a lot of time with only a few people sort of playing around with them before they become like the mainstream thing. Java spent two or three years. It wasn't called Java when it first came out. Spent a bunch of time with people kind of noodling on it before it really became a mainstream thing. Maybe that will be the path.

I think what's important is that we have the discussion, and what's important is that we drive forward with these ideas, because — I guess the other thing I would say is I don't know that Metaparticle necessarily will be the system that everybody uses, but I am 100% confident that a system like this will be the way that people develop for the cloud in, say, 3 to 5 years. You just have to. There's just too many of these systems that need to be built. It's like imagine the number of programs that you would have built if everybody — If the only way to build applications with C++ and like old-school UIs.

[0:34:38.3] JM: Yeah, not so many.

[0:34:39.2] BB: Not so many, rights? Visual Basic, Java, C#, all of these languages that came in the sort of the mid-90s, when suddenly it was clear like, "Oh, yeah. We need thousands of applications or millions of applications. We don't need you know a handful of applications." These languages were developed to radically increase the number of people who could successfully build applications. I think what's happened now is it's no longer sufficient to build an application that runs on a desktop. We have to build reliable cloud applications, because people expect multiscreen experiences. People expect their data to follow them around. So we're back in the world where the number of people who know how to build those things is just too small for the number of systems that need to be built. As a result, we build bad systems that aren't reliable, that are hard to maintain, because people don't have the skills. Also, they're too expensive. So we see systems not get built, because they're too expensive. We have to make it easier, and I think that it's inevitable that happen.

[SPONSOR MESSAGE]

[0:35:44.7] JM: If you are building a product for software engineers or you are hiring software engineers, Software Engineering Daily is accepting sponsorships for 2018. Send me an email, jeff@softwareengineeringdaily.com if you're interested.

With 23,000 people listening Monday through Friday and the content being fairly selective for a technical listener, Software Engineering Daily is a great way to reach top engineers. I know that the listeners of Software Engineering Daily are great engineers because I talked to them all the time. I hear from CTOs, CEOs, directors of engineering who listen to the show regularly. I also hear about many newer hungry software engineers who are looking to level up quickly and prove themselves, and to find out more about sponsoring the show, you can send me an email or tell your marketing director to send me an email, jeff@softwareengineering.com.

If you're listening to the show, thank you so much for supporting it through your audienceship. That is quite enough, but if you're interested in taking your support of the show to the next level, then look at sponsoring the show through your company. So send me an email at jeff@softwareengineeringdaily.com. Thank you.

[INTERVIEW CONTINUED]

[0:37:12.3] JM: I want to put you on the spot here. The way that I think we've seen people build platforms on top of Kubernetes so far, it's fantastic, but I think we're going to see a lot more in the future. So I think what we're seeing today is mostly people that are building really nice visual façades on top of Kubernetes, and then we're also seeing things like what Cloud Foundry has done where they're kind of reworking their core orchestration layer to also expose Kubernetes, and all that's great.

You have a background in robotics. I'm sure you have thought a little bit about how Kubernetes might be used as a distributed operating system for robots or how you could build a platform on top of Kubernetes to work with robotics. Do you think that would be a use case where somebody might build something on top of Kubernetes?

[0:38:04.8] BB: I think that there is — I don't know about robotics necessarily, although I think we are seeing the increasing mixed mode stuff where devices are offloading a lot of their computer up to the cloud. Machine learning loads are running in the cloud when possible and things like that. But I do believe that like we need to — To your earlier question about sort of either it's a façade on top of Kubernetes or it's in a full-blown PaaS where Kubernetes is sort of there as a feature, but it's not really the main focus. I do view Metaparticle and other things like that as being a standard library. I mean I think that the important way to approach this is to think about the fact that you have a runtime like the CLR, the JVM, but without a standard library it's useless. It's not useless, but it's a lot harder right? If everybody just started with object and then had to build everything on top of object, we wouldn't get very far. We need this standard library of useful patterns and useful abstractions that in order to build our applications quickly and reliably, but it can't be a PaaS. It can't be a full-on, like, “This is the only way to do things. We’re going to be super opinionated.” It has to be more like a standard library that you'd expect, where I can take the bits that I need and in the places where I don't want it or it doesn't fit, I discard them and I build my own.

I think building up that layer where it's not — You have consumable abstractions, but that they're not all-encompassing. I think we have to do that, and that's a part of where I want to see the community go, is to figure out how do I build a set of reusable modules that I can consume in an à la carte way? The load test example is a great example. I just want to instantiate a load test. I don't want to deal with it. Ideally, I want to instantiate a load test in my code. I want to write a little Java program and run that Java program and have that Java program become the load test, because that's the way that it's going to become easier for me to — That's my natural sort of place to express these things. I don't want to have to write some YAML. I don't want to have to learn about some extra config, learn about some new tool. I just want to describe this thing in a language that's familiar to me.

I think if you look at like Brigade, which is this tool that we open-sourced recently, which is a workflow engine for running workflows in Kubernetes. You write your workflow as a JavaScript, because writing it as a programming language just feels very natural. It feels very familiar to people. You express the workflow as a simple JavaScript program, but when you run it, it goes and executes containers in parallel and synchronizes them on a Kubernetes cluster. I think that's a great example of, again, allowing people to use familiar tools to access the power of the cloud and the power of a cluster orchestrator.

[0:40:59.8] JM: Another element of the future and also of the present is the idea of cluster wide services. So we should have things like logging and monitoring and security standardized across the cluster. Of course, as you said earlier, it's problematic if your organization has 20 different Kubernetes clusters and you can't standardize between the clusters, but let's assume we're just trying to standardize within one cluster. Do you think all of these things are going to be facilitated by the service mesh abstraction, or do you think that the service mesh is just going to be one of a number of different cluster wide services?

[0:41:40.3] BB: I think it's one of a number for sure. Service mesh is never going to do logging for you. Not like the stuff that you print out. It's never going to be able to catch a stack trace or aggregate stack traces to tell you what your most common error is. It's not going to be able to understand like it's all based on RPCs and HTTP RPCs generally at that. So there's going to be all sorts of things that it doesn't understand. If you have a batch system that is processing jobs, like it's transcoding video say, service mesh isn't going to help you understand your video transcoding system and why it's running slow or what may be happening or alert if you were falling behind your workload and you need to scale up.

I think it's a great tool to have. But I think that there's going to be lots of these cluster-wide services. I mean intrusion detection. There's just all kinds of these daemons. Like service mesh is a very active thing. It's like in the middle of your application while your application is executing, but there's a lot of other sort of cluster level daemons that you would want to run that are more about introspection, or more about monitoring or certificate rotation. There's lots of these kind of like utility services that you want people to just be able to opt into that the service mesh isn't necessarily going to be able — Isn't intended to handle for you, and I think that's right. I mean we really subscribe. I really subscribe to the UNIX philosophy of lots of little tools with focused purpose. So I think that's the right way to design these things.

[0:43:10.8] JM: What the design pattern that you would like to see for these cluster daemons?

[0:43:16.5] BB: I think that the pattern that we're starting to see emerge is — It's twofold, I guess. There are customer level daemons that you just install as a daemon set as a cluster owner. Things like intrusion detection, or logging, where I just kind of want them to be like oxygen. Like I just want them to be there. Like I don't even want the user necessarily to know that it's there. I just want them to get it automatically whenever they deploy. Then there's more sort of like augmenting services, where I'm going to install something that does certificate rotation. Well, in order to actually do that, well I need the user to put an annotation on their ingress controller or to — That they want a certificate here and they want it to be rotated. So that's a little bit more like an example where the daemon expects an input from the user. It's abstract input. It's not like a very specific thing. It's a high level concept that the daemon then makes easy to use, but I think those are sort of the two main points of interaction. Either It's just sort of there and you don't even necessarily notice or you're creating a custom resource definition object that represents a higher level way of interacting with the daemon.

[0:44:25.5] JM: It's clear that people are developing the open-source implementations of many of the cloud services that we have seen over the last 10 years, for example, object storage. Replicated object storage seems like it's going to become a nonproprietary commodity, or it already has, and it's very easy to deploy an open-source version of an object storage system to your Kubernetes cluster. What's the future for managed cloud services? If we can build an open-source version of whatever on — We can build off of Kubernetes, where should the cloud service providers be focusing their time?

[0:45:06.4] BB: Well, it's a mix. I mean, I think that if for some things, Kubernetes is going to be able to give you exactly what the cloud gave you at a cheaper price point. So I think that like web services style PaaS, I think we're already starting to see that shakeout. We're already seeing the sort of like traditional PaaS vendors be kind of challenged by container orchestration.

On the other hand, like problems like storage, you got to be at a pretty big scale where it makes sense to roll your own SRE team. You carrying the pager is almost always better than me carrying the pager until I can hire five or six people to carry the pager. It's hard to hire a fractional person. You can't really have like a 10th of a person to carry pager, and you can't even really have one person to carry a pager, because they'll go insane. You need at least four or five people in order to have a healthy pager rotation, and otherwise people are just constantly on call, and that's bad.

There's always going to be a scale point at which the cloud version just makes more sense, and I think that's just always going to be true. Now, the specifics, like if you could turn up a four 9 storage service in Kubernetes that ran forever and upgraded itself and like just sort of worked all the time. Well then, yeah, cloud services are going to be under threat. We're not there yet, but I suspect we will be eventually, and then they'll provide some higher level or harder abstraction. I think that's just sort of the natural way things go. I do think that one of the things I would love to be able to see is I think that we've kind of — In the world of cloud, we've kind of destroyed the independent software market. Because people expect things to be delivered as a service and because it is so harden and potentially expensive to the workings of the service, there isn't the equivalent of like downloading an application off the Internet and running it and getting like a three of four 9 service out of it, because the sort of thing that you would get with access on your one workstation in 1990.

I think that we need to be able to create that market again, so that a software team of five people can effectively sell like the equivalent of software on a CD, where I sell it to you and you use it and it mostly works and you'd never really feel like you need to call me. If we can do that, then we can re-create sort of the independent software industry in a way that I think it's been sort of under threat in the world of cloud. I think it's a really important thing to do. I think it's a much healthier world for our industry if people can people can make money as software developers effectively.

I see Kubernetes as being a big part of that for sure, but I think we need to do more. Another way to think about this is like, it used to be that you paid attention to the version of the web browser you ran. Like you are like, "I'm running version 4.5, or 4.7 or whatever," and then web browsers just started to auto upgrade and people stopped noticing. They're just like, "No. I just have a web browser. I don't know what version." I guarantee you, you probably can't tell me what version of the web browser you're running.

[0:48:12.5] JM: That's right.

[0:48:13.5] BB: And that's a huge shift, right? I think people forget about kind of how big a shift that is. I suspect we with Kubernetes, we will get to a place like that with these cloud distributed applications where it's, "I can download and install this application from you as an independent software vendor," and it just kind of takes care of itself. It mostly just works all the time, and it's not perfect. It's not like the 6 or 7-9's you're going to get out of a cloud storage service, but it gives you 3-1/2, four 9's at a lower price points and you're happy with that.

[0:48:46.0] JM: Yeah, and you're not paying for subscriptions. You're not paying for support. You're paying for a proprietary binary, which is fine because that makes a more clean, kind of a clean contract that doesn't really exist in today's web applications. But if everybody's on Kubernetes then it creates a market for people to make these $99 Kubernetes binaries that, “Hey, I sell this to you. You take it, and that's it. That's the end of our transaction.”

[0:49:16.0] BB: Yeah, and I think that right now the reason people don't do that in the cloud in general is because like they know that they're going have to operate it for the rest of their lives and they know that most cost comes from the operations. But if via Kubernetes we can create a world where actually the cost of those operations go away and the binary itself knows enough about itself to like deploy itself and work with Kubernetes to scale itself and upgraded itself when need be, suddenly then people start maybe saying, “Oh, actually. You know what? I'll do that. $99 for two years’ worth of us key-value store that will just kind or keep itself up-to-date. Yeah, I’ll do that.”

That’s great, because that means that it creates a business model for a lot of even open-source startups that I see right now who are kind or struggling, who are kind or trying to figure out like we really don't want to be in the consulting business, but we’re not big enough to do software as a service. How do we find a place in the middle? We have to create that place in the middle, I think, in order for the industry to be healthy. It's something I'm definitely very excited about and passionate about.

[0:50:20.9] JM: That is so cool.

[0:50:22.1] BB: Honestly, I would say like I think Microsoft is an ideally place. Microsoft has a really strong reputation for partnering and partnering well and ensuring that our partners can make money on our platform and being committed to our partners making money on our platform. So that's one of things I'm really excited about being here, is building that platform where not only is Azure successful, but actually the partners we work with to provide services on top of Azure are also successful.

[0:50:48.2] JM: I know our time is short here. I want to ask you a little bit about building a managed Kubernetes Nettie's as a service offering. So you've been at Azure for a while working Azure a Kubernetes service. Whenever I talk to people at cloud providers, it so interesting getting introspection into what it's like to be at one of these giant cloud providers. So tell me, what is it like to build a Kubernetes as a service offering that is deployed on a massive cloud?

[0:51:19.4] BB: I think you have to remember that nothing can be bespoke effectively. If there's a thing that you can automate, you have to automate it, because you may think, “Oh, you know what? We only have a hundred of these clusters today. It's not that bad if I have to run a script across all of them,” but next month it’s going to be a thousand, and the month after that it’s going to be 10,000. You always have to be thinking about how you automate all aspects of what you do, as well as having the right alerting in place right. I mean, I think that you have to know before the customer does that there's a problem. That's a big thing. That's extremely important. So I think those are some of the big aspects of it. It comes with a certain amount of pressure and scale, but I think that's good. It means people are excited and interested in what you are doing. I should also say like this is real credit to the team. This is not I felt was sitting in my basement cranking out Azure container service. It’s a big team effort. I've got a team that's on call and making sure that it stays reliable, and I’m really grateful to them for doing that. I think that's the other — The other piece of it is it takes a village to deliver one of these services. It's not the sort of thing you can kind of do on your own.

[0:52:34.8] JM: What are the subjective decisions that somebody can make, because there's so many Kubernetes as a service things. What are some of the subjective decisions that you’ve had to make when designing Azure a Kubernetes service?

[0:52:46.8] BB: Yeah, that’s actually one of the hard things, is that you always go up to a customer and the customer will be like, “You know what? I totally love your managed service. Totally love it, but can I just have this one thing over here?” You have to decide like, “Is it worth it? Is that one feature that this customer wants, is it going to destabilize other users? Is it something I can operate and sustain?” If 80% of your customers are asking for it, then it's a no-brainer, but if it's one, it's like, “Well, do we think more people are going — Is this person a leading edge customer? Are they asking for a feature now that only one person is asking for, but in six months, everybody's going to ask for it? In which case it makes sense to do it now, get ahead of the ball, or are they just a snowflake and do they just have this weird —” Sometimes it's like institutional stuff, like they said, “Well, you have to be on this specific version of the specific operating system, or else you can't run.” That’s like an IT, central IT dictate. Because of that, actually we run not just Azure Kubernetes service, but we actually have an open-source service — Or not service, but an open-source product called ACS Engine, and ACS engine, it doesn't have an SLA. It's just a tool that helps you stand up Kubernetes clusters on Azure, but because it doesn't have an SLA and because it's a tool that you use as opposed to a service that we provide, we can be a ton more flexible. There about helping you craft a solution that fits your needs and still not have you have to fall all the way back to raw IaaS.

Oftentimes, what would happen before was someone would come in and say, “Oh, I love Azure container service, but I need this one little exception,” and we’d have to say like, “Well, here's the manual. Go figure out how to run Kubernetes on Azure.”

With ACS engine, we have sort of a place in the middle where we can say, “Hey, we have all these tooling. You still operate it yourself. It's still your thing. You might need to actually do some PR's to the ACS engine to get the things that you need in there.” But it's a middle ground. It's a place where we can collaborate, and as we improve running it on Azure, we upgrade Kubernetes from 1.8 to 1.9. People who are using that tool can inherit those upgrades, and it actually is the core of AKS also. It's just that we — It’s AKS as a subset of the functionality that's available at ACS engine. And so in many cases we start — A feature starts in ACS Engine and then we see more and more customer demand, and we migrated into AKS, oftentimes code that customers have provided. So we’re actually having customer provided code inside of ACS Engine that that eventually becomes part of AKS. That’s been a really great way to kind of be able to move across the spectrum from like totally pure IaaS where the customer — We’re not even involved. The customer is just doing it all themselves, to a place in the middle where we’re working collaboratively in in an open-source context around some tools, to the full managed service. I hope by doing that we can hit customers in all the different phases of the lifecycle there.

[0:55:39.8] JM: Brenda Burns, thank you for coming back on Software Engineering Daily. It's been great to have you.

[0:55:43.4] BB: Yeah, thank you so much for having me. It was great to chat.

[END OF INTERVIEW]

[0:55:49.1] JM: GoCD is an open source continuous delivery server built by ThoughtWorks. GoCD provides continuous delivery out of the box with its built-in pipelines, advanced traceability and value stream visualization. With GoCD you can easily model, orchestrate and visualize complex workflows from end-to-end. GoCD supports modern infrastructure with elastic, on-demand agents and cloud deployments. The plugin ecosystem ensures that GoCD will work well within your own unique environment.

To learn more about GoCD, visit gocd.org/sedaily. That's gocd.org/sedaily. It's free to use and there's professional support and enterprise add-ons that are available from ThoughtWorks. You can find it at gocd.org/sedaily.

If you want to hear more about GoCD and the other projects that ThoughtWorks is working on, listen back to our old episodes with the ThoughtWorks team who have built the product. You can search for ThoughtWorks on Software Engineering Daily.

Thanks to ThoughtWorks for continuing to sponsor Software Engineering Daily and for building GoCD.

[END]