EPISODE 1029

[INTRODUCTION]

[00:00:00] JM: Infrastructure as code allows developers to use programming languages to define the architecture of their software deployments including servers, load balancers and databases. There have been several generations of infrastructure as code tools, system such as Chef, Puppet, Salt and Ansible provided a domain-specific imperative scripting language. It became popular along with the early growth of Amazon Web Services. HashiCorp's Terraform Project created an open source declarative model for infrastructure, and Kubernetes YAML definitions are also a declarative system for infrastructure as code.

Pulumi is a company that offers a newer system for infrastructure as code combining declarative and imperative syntax. Pulumi programs can be written in Typescript, Python, Go or .NET. Joe Duffy is the CEO of Pulumi and he joins the show to talk about his work on the Pulumi Project as well as his vision for the company. Joe also discusses his 12 years at Microsoft and how his work in programming language tooling helped shape how he thinks about building infrastructure as code with Pulumi.

We're looking for show ideas. If you have an interesting story about infrastructure, or a blog post, or a conference talk that you saw recently, or just something that you'd like to hear about, please send me an email, jeff@softwareengineeringdaily.com. We're looking for new ideas and fresh projects to cover in the world of software.

[SPONSOR MESSAGE]

[00:01:42] JM:

[INTERVIEW CONTINUED]

[00:01:42] JM: I've recently started working with X-Team. X-Team is a company that can help you scale your team with new engineers. X-Team has been helping me out with softwaredaily.com and they have thousands of proven developers in over 50 countries ready to

join your team and they can provide an immediate positive impact and lets you get back to focusing on what's most important, which is moving your team forward.

X-Team is able to support a wide range of needs. If you need DevOps, or mobile engineers, or backend architecture, or ecommerce, or frontend development, X-Team can help you with what you need. They've got a full-range of technologists who can help with AWS, and Go lang, and Shopify, and JavaScript, and Java. Whatever your engineering team needs to get to the points of scale that you want to get to, X-Team can help you grow your team. They offer flexible options if you're looking to grow your team efficiently, and their model allows for seamless integration with companies and teams of all sizes. Whether you're a gigantic company like Riot Games, or Coinbase, or Google, or if you're a tiny company like Software Daily. You can get help with the technologies that you need. If you're interested, you can go to x-team.com/sedaily. That's x-team.com/sedaily to learn about getting some help with your engineering projects from X-Team.

Thank you to X-Team for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[00:03:27] JM: Joe Duffy, welcome to Software Engineering Daily.

[00:03:29] JD: Hey, it's great to be here. Thanks.

[00:03:31] JM: You run Pulumi, which is a company building tools around infrastructure as code, and when I think about the evolution of infrastructure as code, I think of several generations. You've got the bash scripting generation, and then the Chef, Puppet, Salt stack generation of tools, then maybe cloud formation in Terraform, and then the current state of affairs where it's container orchestration, a lot of Kubernetes, and obviously these generations overlap with each other. They're not exactly mutually exclusive. But I'd like to start by getting your historical reflections on the different generations of infrastructure as code.

[00:04:10] JD: Yeah. It's a great question. I think a lot of the changes kind of mirror the changes in the cloud, honestly. I think you look back to the first generation of infrastructure tools, you

look at bash scripting and just sort of manual processes. A lot of teams previously that developers would file tickets to have their IT organization stand up servers, and then that turned into scripting, and then Chef and Puppet really were the first time where, “Hey, let's use code to solve some of these challenges.” But those were really born in a world of virtual machines and configuration where you’re patching servers. You’re not provisioning new servers. They go over the years and then the cloud vendors themselves introduced their own infrastructure as code tools, with cloud formation, arm templates, Google deployment manager.

Terraform was born around the time of cloud formation. I think Terraform was the first that really said, “Hey, let's take a consistent approach to this provisioning challenge where we’re provisioning new infrastructure,” and that's a DSL, which is definitely a step function better than using YAML or JSON. I think these days, you see Pulumi, the approach that we took is embracing general-purpose languages like Python, JavaScript, typescript, Go, C#, and AWS has the CDK Project, which is kind of taking a similar approach, but for the more AWS only style of project. I think that's really trying to recognize that, “Hey, developers these days really do have to think about cloud infrastructure in addition to the infrastructure teams themselves.”

[00:05:52] JM: Give a brief overview for how Pulumi approaches infrastructure as code.

[00:05:57] JD: Yes, my background, I was at Microsoft for a while and I actually ran the languages groups before leaving. So I came from a very developer-oriented background, where what got me up out of bed in the morning was thinking about developer productivity and really giving great tools to developers.

When I came to the infrastructure as code space, it kind of struck me that we just sort of applied different standards to the productivity, the tooling, the ability to test your infrastructure. We really wanted to take everything we knew and love about application development and apply that to infrastructure development.

It's great for infrastructure engineers who want to be more productive. They want be able to share and reuse patterns rather than copy and pasting, but also acts as a bridge. So now, “Hey, the development team can be a little bit more self-served.” In small teams, you really don't want to have that divide between the two, the infrastructure team and the developers. You just want

to say, “Hey, engineering team, go build something great and use the cloud to do it.” But in larger teams, you actually see the infrastructure team wants to let their developers go and spin up new microservices or serverless functions or queue in databases. But the technologies today sort of kind of force the silo, because they’re fundamentally different languages, different tools. We took an approach of saying, “Hey, let's unify those two sides of the engineering house and really help people work better together using familiar tools and languages.”

[00:07:27] JM: Describe the difference between imperative and declarative programming. When it comes to infrastructure as code, what are the merits and costs of each of those programming paradigms?

[00:07:42] JD: Yes, very good question, because a key piece of the magic with infrastructure as code especially for provisioning is the idea that it's based on declaring your desired state. Most infrastructure as code tools, the program, if you will, actually declares a desired state. It says, “Hey, I want these security roles. I want a Kubernetes cluster. I want a database. I want a load balancer. I want all these things.” Then the infrastructure as code tool, it's job is to make that happen and to do it in a reliable way where it works not only for the first time you spin up the infrastructure, but then when you go and make a change, like maybe you add a third server. The infrastructure as code tool, you’re, again, declaring your goal state. The infrastructure as code tool can say, “Hey, you’ve already got two. You’re asking for a third, so I’m just going to go create that third.” It can do this like incremental updating.

The approach that we took was actually kind of marrying the idea of general-purpose languages, which as you mentioned can be imperative. We actually support functional as well with F#, but marrying that model with the declarative model. So you're actually just using an imperative language, so you can use for loops or classes or functions or functional language if you prefer to declare that goal statement. It’s still declarative at its core. That's actually a key element that we don't want to lose, because today every cloud vendor has their own SDK. If you want to go write some Python to go spin up some servers, you can do that, but that imperative model without that declarative goal state core, there are a lot of reliability challenges. If it fails, what do you do? How do you recover from a partial state? You don't have a full history of kind of who changed what, when. You can't preview the changes and make a plan before you apply it as two independent steps, which are all key things that infrastructure as code gives you.

[00:09:37] JM: I can understand the merits of introducing imperative logic. Like you said, you have if statements and for loops and you don't just have a totally declarative structural representation of what kind of infrastructure you want. But I thought there were some kinds of risks that were associated with imperative logic for describing infrastructure. Are there some kind of risks or downsides or costs to having an imperative model co-mingled with that declarative syntax?

[00:10:12] JD: I'd say there are really two things. One, the tool has to get it right, and that's actually tricky. It took us about a year to really figure out how to do this, because you do want to be able to see all the changes before you execute them, right? In a typical imperative model, you don't know what the code is going to do until you run it.

Normally, let's say you're going to deploy a change in its going to lead to downtime. Maybe you're changing a property on a server that can't be updated in place or you're rolling out a new Kubernetes master version and you don't have an HA configuration, and so that change is going to require some downtime.

In many imperative models, until you run the code, you don't know if there's going to be – You don't know what the impacts are going to be, and so you can't plan for that. You can't plan for the downtime. Whereas, we've kind of figured out the hard problem of how do you marry that with the declarative model. The implementation of that is quite difficult and gnarly. It's got a lot of promises and asynchronous dataflow dependencies in its. It was tough to figure out. It took us probably three or four iterations before we really got it right. That's the one area of complexity I would say, or risk, that we don't have because we've figured out how to do that.

The second area of risk that definitely I hear from folks is, "Hey, once you've got a full language, you can build entire castles of complexity," and certainly that's a risk. I think that's a risk with application development too, however. That not one that really resonates too much with me and we don't see that much in practice. It turns out abstraction is kind of a good thing if you want to function here or a class there, you're free to do that. But certainly some people can take it too far.

If you're having – Like some folks are really religious about their sort of object-oriented patterns, and so if you're seeing factories of factories in your infrastructure as code, maybe that's an anti-pattern. Certainly, you've got the full power of a language and you can do great things with it, but you can do awful things also.

[00:12:17] JM: Have you ever seen that? That sounds horrific.

[00:12:21] JD: Not in infrastructure as code. I used to do a lot of Java programming and I definitely saw it.

[00:12:25] JM: Same here. I was going to say I've seen that Java code fly.

[00:12:29] JD: No. I do see your folks creating factories for servers, and also it's really cool some of the patterns that do emerge that are powerful. Some folks are using abstraction to help tame the complexity of multi-cloud. Being able to run a service on Azure Kubernetes and AWS Kubernetes and have some abstraction that helps maintain that. but you can definitely take it too far for sure.

[00:12:55] JM: Now, we've touched on some of the contours of what Pulumi is. It's obviously infrastructure as code. It touches on kind of the foundations of what makes different programming languages useful and how to make developers productive. I think there's a lot of elements to the company that you're trying to build and it kind of resists any analogy to any particular other company. To understand what you're trying to do with Pulumi, I'd like to get a little bit a historical context because you spent 12 years at Microsoft and many of those years were working on language tooling. Of course, this was in the context of Microsoft evolving into a company that was focused on cloud. So I'm sure you were in some really interesting conversations around what's the nature of programming languages. What's the future of programming languages in the context of cloud? I've seen some other projects that are kind of approaching the programming language question with acknowledging the cloud. There is the Brendan Burns M particle, or what was it? The Metaparticle idea.

[00:14:08] JD: Metaparticle. Yeah.

[00:14:10] JM: There's that thing. There is the Ballerina language, Dark lang, these kinds of like cloud- first programming language things, but I'd really like to get your perspective on how your work on programming languages has informed how you see infrastructure as code.

[00:14:26] JD: Yeah. That's a great, great question, because I did not expect to end up building an infrastructure as code platform. It really happened because of that long journey. I'd say, my early days at Microsoft, I was on the C# design team very early on. I worked on some really interesting projects. I think sort of around 2002, 2003 was sort of when multicourse started becoming a thing, and pretty high- level folks at Microsoft started to get a little concerned because if chips were getting faster, then – There's this is old saying, "Andy giveth, and Bill taketh away which," which was Andy Grove at Intel would ship faster processors, and before they would hit the market, Microsoft would figure out new software capabilities that would use up all of that computing power. That really was like the Wintel era, right? There is this nice synergy between the two, and that was threatened. It stalled out around that timeframe.

I sort of led some efforts to try to bring parallel computing into mainstream languages, and through that I kind of realized that there are all these specialized languages that we can learn from and there's – Also, you can read papers from the 1960s and 70s on parallel computing and how it's going to be the next big thing, and a lot of them talk about distributed and computing as well.

Back then, I really brought some of those key ideas into existing languages. Instead of creating a new programming language, like this one, NESL. I can't remember what it stands for, NESL. But they had data fork/join parallelism, data parallelism kind of baked into the language. I use this as inspiration, and I learned from that that while at times libraries are actually kind of an extension of the language itself, you don't always need a new language to express a core concept.

Honestly, a lot of the stuff we did there ended up turning into the async await stuff that started in C# and is now in JavaScript. It's in Python. RUST is now adapting it. A lot of the work we did back then really foreshadowed some of the modern asynchronous computing challenges.

I did work on a distributed operating system through that a little bit later around the cloud era where it's more of a distributed operating system with a lot of message passing and, honestly, foreshadow a lot of the microservices challenges that were happening now with service discovery and how things connecting and fault tolerance.

We did create a new language in that process, but ultimately folded a bunch of the ideas back into C# and C++ in fact. I think my biggest takeaway when I came to this new space was although I saw a lot of reasons why a new language might make sense, I saw many more reasons why we didn't need a
new language. I saw, "Hey, we could actually just takes some concepts and put them in the language." At Pulumi, we needed the idea of expressing this goal state with resources and cloud resources, but using promises and existing concepts actually was enough to really get something that worked and worked well. You didn't have to learn a new language. You can use your favorite language still.

That said, languages like Dark, I mean, I love them. They're really pushing the boundaries on the ideas. I think in a lot of these cases you end up pushing the boundaries and then you take the lessons learned and figure out how to apply them back to the existing technologies.

In some cases like with RUST, you really have a breakout success and that thing just becomes – It hits the tipping point to become a thing that exists over the long run. But in either case, I think goodness comes out of it.

[00:18:04] JM: What was it like being at Microsoft as the company shifted its core focus to Azure?

[00:18:11] JD: It's interesting actually, especially on the other side now where basically we're kind of the interface for a lot of our customers into AWS Azure or Google Cloud. When we started talking about Azure, I mean, it started with Dave Cutler who he's the main architect for Windows NT. He's a very operating system-oriented guy. We used to talk about Red Dog. We used to talk about as a cloud operating system, which is kind of the way I approach a lot of the things with Pulumi. It's like you write your code.

Today, imagine you want to write just a normal client-side program, you write a bunch of code and behind-the-scenes there're these operating system kernel objects that get created, file handles and mutexes and sockets and things like this. The operating system manages them. Well, imagine the cloud is just the new operating system and all these resources we talk about are actually just cloud-managed kernel objects in a sense. Well, that's kind of the way that we approached Azure in the early days.

The interesting thing is AWS approached it from a very different angle, and it's actually quite apparent now. I think Azure is much more platforms as you end-to-end sort of platform-y services. AWS on the other hand gives you these 180 different building blocks that you then stitched together. You can see those differences in philosophy really shining through, but it was an exciting time. It was also interesting, because at the time you had Bing and Azure in they were kind of two separate things. I was involved, in some cases, reconciling those differences, but definitely super exciting. A lot of my colleagues and friends are now kind of like in the Azure group and Azure continues to just continue growing and all the developer tools are over there. So it's come a long way.

[00:20:01] JM: When did you start thinking about building a company?

[00:20:06] JD: Well, I almost started a company before joining Microsoft. In fact, my first job was a consulting business that I kind of had in high school trying to help people get their businesses on to the Internet. So I kind of had a little bit of a taste of what it was like to run a business back then. But I want to say, I went to Microsoft. It's a great opportunity, great technology, great people. I just kept learning more things and meeting new people and having fun. Every year I actually ask myself, “Is now the time to leave and start a company?” and the answer was always no. I kept on learning new things and I really wanted to get a taste of managing technological innovation at scale, which Microsoft very few companies on the planet allow you to see at that level of scale. So I found that super valuable.

I just wrapped up the project to open source .NET and take it cross-platform. I was involved in that effort, and it got to a good place where I felt like now I’ve learned so much. I can now go and actually seize this opportunity, and that was the other element. I saw this huge opportunity with the cloud to really make it easier to program and bring it more into the developer’s inner

loop. So the timing is perfect. It just so happen I had great cofounder who was ready to go at the same time. It was kind of serendipitous. So we said, “Hey, now's the time. We’re going to make the leap,” and we did. It’s definitely not easy. It took a while to figure out kind of exactly what we’re doing, but I'm so glad that we did.

[SPONSOR MESSAGE]

[00:21:51] JM: DigitalOcean makes infrastructure simple. I continue to use DigitalOcean because of the low friction and attention to user experience. DigitalOcean has kept the experience simple and I can spin up a server in less than a minute and get high quality performance for a low price. For an application that needs to scale, DigitalOcean has CPU optimized droplets, memory optimized droplets, managed databases, managed Kubernetes and many more products. DigitalOcean has the flexibility to choose the right instance for the right workload and he could mix-and-match different configurations of CPU and RAM.

If you get stuck, DigitalOcean has thousands of high-quality tutorials, responsive Q&A forums and a customer team who treats customers respectfully. DigitalOcean lets developers focus on what they are building. Visit do.co/sedaily and receive $100 in credit over 60 days. That $100 can be put towards hosting or infrastructure and that includes managed databases, a managed Kubernetes service and more.

If you want to get started with Kubernetes, DigitalOcean is a great place to go. You can use your $100 to start building your distributed system and you can get that $100 in credit for free at do.co/sedaily.

Thank you to DigitalOcean for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:23:26] JM: Well, it’s 2020 and you’re building an infrastructure as code company. What are the shortcomings of the available solutions for infrastructure as code? Why did you need to build an entirely new company around infrastructure as code?

[00:23:42] JD: Yes. So the first – How it started was I was seeing a lot of tooling around Azure and I saw Kubernetes and Docker Swarm and those days. It was a lot easier than Kubernetes, but then you had to manage your own etcd cluster if you want it to go in production. But I fell in love with that Docker inner loop and I saw AWS Lambda was out there and it was just – I saw these amazing capabilities and I just wanted to use them in my application. I wasn't coming from an infrastructure perspective. I was coming from I see these capabilities and I see what they can potentially do to my ability to build more powerful software. I want to use them.

I think I was just a lot more naïve than I should have been. So that led me to think, "Hey, imagine five years, every developer is going to be using the cloud in one way or another," and the cloud is kind of a superpower for people, right? Whether it's hosted managed databases or compute services so you can do event-driven computing using serverless or hosted AI/ML services that allow you to do speech recognition as part of your application, the cloud has all these superpowers and it's just so hard for developers to get access to it today. Sounds kind of the starting point, and I guess I eventually realized that until the infrastructure as code problem is solved in a way that integrates more seamlessly with the application development stack that we could never get to that level.

Really, I saw power in the way that AWS sort of has built their platform where it really is all these building blocks. The idea was infrastructure as code is just kind of a way to stitch together all these building blocks to build bigger things out of smaller things. That was the enabling capability that I thought we needed to make easier for developers and to really make accessible to developers.

I looked at the tools today and I was like, "Well, I don't want to learn a new language, and if I'm going to go through the trouble of learning a new language, it'd better be a great one," and that wasn't the case. There were a lot of DSLs. There was so much YAML. I mean, I just wanted to do a lambda, and the next thing I know I've got three times the amount of YAML as I have application code. That joy wasn't there, right?

I think one of the defining characteristics of what we have an application development is joy. You got great editors. You get interactive documentation. If you mistype something, it tells you. It's a

great thing that we've got going there and I just wanted to bring that over to the infrastructure space, which frankly it seem like our standards were just a little bit lower than they should be.

[00:26:24] JM: There was a talk I heard where you mentioned Docker Swarm, and this comes to mind now. You are a fan of Docker Swarm, but as you know, Kubernetes won out. I wonder if you have any reflections on why Kubernetes won the container orchestration wars and just reflections on that time around what people want out of an infrastructure management tool. What did you learn from the container orchestration wars?

[00:26:52] JD: Yeah, it's a great question. Even though I'm a huge fan of Docker Swarm, I was always long on Kubernetes for a few reasons, and I can tell you why, but it's funny because people ask me a lot about Kubernetes and I think like Kubernetes is sort of should fade into the distance. I look at a lot of the – I mentioned, I work in parallel computing. I think back then you had to manage your own threads, and you didn't want to manage your own threads. You just want it to run tests or you want it to do data parallelism and it's like, "Well, I need some threads to run the compute. So now I have to manage them, and that's kind of painful because what if I have lots of competing demands? Well, there needs to be a scheduler, or there needs to be something doing resource management." So we came up with this thing called a thread pool, which is something that would like pool threads and schedule across them and make sure you didn't have too many, but you had just enough to make the most out of the resources you have.

I kind of view Kubernetes as filling that need, just instead of threads, you've got containers. I think this analogy, again, which known for taking analogies too far, but if you think of this cloud operating system, well, the containers are kind of the threads and you just want the cloud to manage the scheduling of them and you want to just declare, "Hey, I've got these 10 things that could run. Can you just make sure they do?"

Docker, the thing that hooked me was it's the first time I ever felt like, "Wow! I can just take a piece of code, an application, or a service and I can just package it up and run it in the cloud," and it's going to run in the cloud pretty much the same way that it ran on my desktop. Of course, you get into trickiness around networking and storage, but the problem is one thing is far less interesting than N things, and you almost certainly want a lot of them and they need to connect with each other.

Swarm to me, Docker Compose, I think it was called Fig before it became Docker Compose, I thought was great because it was easier. I've got a database. I've got an API server and a frontend and they connect in this way and you just run Docker Compose up and magic happens and it's great. The problem is running that in the cloud was then a huge drop off. You had to then manage your own etcd cluster.

The thing that was pretty clear to me early on is Kubernetes, although way more complicated, had already thought of the entire end-to-end set of challenges for how do you do that and how do you do that at scale informed by how Google did that. I always felt like Docker Compose was great for the simple case, but they were just learning as they were going all of those nasty issues that were going to come eventually where Kubernetes had already not solved all them, but at least thought about the hardest ones.

Honestly, I'm always a fan of publishing papers and research. With Omega papers and a lot of the precursors to Kubernetes, you could actually go and you could read all this deep thought and learnings and experiences. For me, that was why I thought in the early days Kubernetes, from a technology standpoint, is going to win out. It's now always the greatest technology or the best technology that wins, but in this case I think the need was so great.

Also, there's one other component to it, which is a lot of the customers we work with, they're using Kubernetes to basically get an approximation of a public cloud in their on-prem hybrid environments. So it's a way to modernize without needing to go bite off the entirety of the public cloud transformation so they can kind of tiptoe towards it. I think that's a huge reason why it's seeing a lot of success also.

[00:30:37] JM: Okay. Great. Now that we've taken the oblique approach, let's go more directly into Pulumi. I want to understand the programming model for Pulumi. Let's say have got a simple web application. Maybe it's literally just a server that serves requests to helloworld.info. When I go to helloworld.info, it just loads a page, it says, "Hello world," and I want to have this application be described in Pulumi infrastructure as code. What am I doing? What kinds of files am I writing? What does that even look like?

[00:31:19] JD: Yeah. I guess the first, we have these nice little walk-throughs for getting started, and basically you have to make to top-level choices, one which language are you going to use, and two, which cloud are you going to host your web servers or web server on? We have Python, Typescript, JavaScript, Go, .NET, so that includes C#, F#, VB. That kind of dictates that style of the code.

I think one of the key things we've tried to do is make sure that every language feels idiomatic. Everything in Python is going to be snake case. In JavaScript, you're going to have things that might be promise-based. C#, you've got async await. That's just one thing to know. The project structure as a result is identical to the project structure in whatever that language is.

If you're using node, we use npm. NuGet for .NET, and pip packages for Python. Everything should feel like you're just at home right away. The same thing is true if you're an editor. If you want to use JetBrains, or VSCode, or Vim, or Emacs, everything, all the syntax highlighting, statement completion, everything, just works. But the key is you basically are going to declare the main function of your program. Essentially serves a slightly different purpose than it typically does, because what it does now is it declares the infrastructure state.

In your case, maybe you're using AWS EC2 VM's, for example. There's native that gets package. Whatever language you chose, you import that package and then you're going to say, "Hey, new aws.ec2.webserver and then every property available on EC2 web servers is exposed to. We try not to get in your way. Even though we're multi-cloud, we give you all the properties, all the resources available in each of the clouds. If you want to use CloudWatch for diagnostics or Route53 for DNS in AWS, you got all of that at your fingertips.

So we've got a lot of sample code and templates that you can start from so you don't – If you don't know everything about AWS, you don't have to like start from scratch. I will say one of the other things to know is because it's a programming language, we give you abstractions. For common things, like say you want a static website. Maybe want to do effectively what Netlify gives you but just hosted in AWS in S3. Well, we've got simple components to make that easy, and that's again the magic of programming languages. It turtles all the way down. You've got classes that can compose other classes. We can encapsulate some of the complexities so you don't have to go super deep on every cloud.

[00:34:07] JM: Now, does this mean that Pulumi is essentially an SDK and the abstractions in the SDK are if I'm on the AWS Cloud, literally, the AWS abstractions, like I can say S3 bucket foo equals new S3 bucket, which is going to spin me up an S3 bucket. Meaning, you plug into the AWS SDKs. Is that an accurate description of what you do?

[00:34:37] JD: Yes. Yeah. We have packages for everyone of cloud providers that effectively just plugs into the SDKs, as you point out. The providers, it's pluggable. Actually, I have some customers who've implemented their own for different resources, but each one of those is projected into every language we support. AWS is available in all the languages. Azure is available in all the languages, but ultimately it's tying into those SDKs.

[00:35:02] JM: This is what to me sounds so hard, is it's by an N by N by N problem, because you've got three cloud providers with N services, an N programming languages that you support. Isn't that really hard?

[00:35:18] JD: It is incredibly difficult. I will say we just keep ourselves sane. We've come up with clever ways to manage the complexity. We actually have available, something like 36 different resource providers. It's not just AWS, Azure, Google. It's also Kubernetes. It's also New Relic, MailChimp, GitHub. Long list of providers there. I think we've taken a few key approaches to using existing provider implementations, like for example, you can actually take any Terraform provider and plug it in and that will just project it into all the different languages. So if you're using a Terraform provider you love, you can actually just plug that into the system and it's really easy.

Also, we do a lot of code generation. The provider model actually has sort of a schema at its core, and from the schema we can actually code generate all the different language SDK projections. Even though you've got AWS in Python, JavaScript, C# and Go, it's all generated from that same central shared logic. So we can test that and make sure the code generation is right and then we don't have to go and manually implement the SDK for every single language we support.

Furthermore, one other thing we do actually, for Kubernetes, we actually code generate the schema itself based on the open API specification in Kubernetes itself, which means when Kubernetes ships a new feature, we have same-day support for it without us having to go do a bunch of manual work. You add all these things together and we've managed to tame that crazy explosion of complexity.

[00:37:07] JM: That's a powerful idea. If I understand correctly – By the way, when I said it was an N by N by N problem, it's actually N by N by N by N, because I forgot to say N services per cloud. N by M by P by Q or whatever. Sorry. But what you said is actually really interesting. If I understood correctly, Kubernetes for example, you can have code generation tools that can – Essentially, if Kubernetes comes out with some new feature, you can have an automated way of saying, "Okay. We acknowledge this new feature exists in Kubernetes. Now we have to make this – We have to generate code that makes this feature available in the Pulumi SDK," and you could do the same thing theoretically if MailChimp updates their API. You could say, "Okay. Now they've added an additional call to MailChimp and we can just run our code generation tools and have then magically ingested into the Pulumi SDK and we could theoretically do the same thing for AWS and Google and so on. That's what you do or am I understanding you correctly?

[00:38:14] JD: It is. Not universally. We do it wherever we can. We're actually working with a lot of different folks to move more in that direction. Azure, for example, actually does publish all these open API specs and they themselves generate their SDK out of them. So that's an area where we're working with them to sort of move that forward.

Google has a project called Magic Modules, which effectively does this as well. I think the unfortunate bit is there's kind of a lack of standardization. It really does it slightly differently. Even though Kubernetes, for example, has open API specs, there is a little bit of glue that we have to implement. For the most part, it's general that works for any new service, but it's things like – If you think about the provider has to do, it needs to know how to create read, update and delete resources.

Usually, the open API spec describes the resources, describes all the properties and describes the APIs to perform those crud operations. But how do you wait for an operation to complete? What if an operation requires two calls to something instead of just one? Is there a way to

cancel operations if I control C my deployment or something? Those are the sorts of things that tend to be kind of bespoke for each of the different open API specifications.

But the nice thing is people are generally moving in that direction because people realize it leads to consistency. It leads to better tooling and better automation and it just makes everybody's lives easier. I kind of wish there was a CNCF standard for this is the resource model specification that everybody's kind of moving towards. Who knows? Maybe we eventually get there.

[SPONSOR MESSAGE]

[00:40:10] JM: When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

Go to softwareengineeringdaily.com/g2i to learn more about what G2i has to offer. We've also done several shows with the people who run G2i, Gabe Greenberg, and the rest of his team. These are engineers who know about the React ecosystem, about the mobile ecosystem, about GraphQL, React Native. They know their stuff and they run a great organization.

In my personal experience, G2i has linked me up with experienced engineers that can fit my budget, and the G2i staff are friendly and easy to work with. They know how product development works. They can help you find the perfect engineer for your stack, and you can go to softwareengineeringdaily.com/g2i to learn more about G2i.

Thank you to G2i for being a great supporter of Software Engineering Daily both as listeners and also as people who have contributed code that have helped me out in my projects. So if you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[INTERVIEW CONTINUED]

[00:41:58] JM: My understanding is it what's hard about doing that kind of thing even through all the mighty power of the CNCF is that AWS just has too much gravity and you can't really have the CNCF telling AWS what to do, because AWS could just say, “No. We don't want to standardize. We want to be AWS. We just want to do whatever we want to do.” I remember having some of this conversation with some of the people that were trying to do like an open event specification to do like the functions as a service, to make the functions as a service world more open and more heterogenous, and AWS was just kind of like, “I don’t think we really want to get involved with that. I think we just kind of want to stick with lambda being the concrete that cements you to AWS.” Is AWS playing nice in terms of making their specifications open and – I don’t know, open enough to allow this kind of code gen that you want to do?

[00:43:02] JD: Yeah. Actually, I completely agree. If AWS weren’t to participate in that sort of effort, it just wouldn't be worthwhile, especially the whole goal is to standardize on cloud resource provisioning. AWS is a sizable percentage of that, that market. But they actually are. If you look at a lot of the cloud formation, documentation and schemas are actually open source. I think the key thing is it actually makes it easier for the cloud providers to build an ecosystem around their tools. For example, cloud formation –

[00:43:34] JM: It’s like regulatory capture.

[00:43:37] JD: Well, actually it makes their lives easier too. It's much easier to – Instead of implementing every single one of these things by hand, it’s a lot easier if you actually have a model. Now, whether you open source that model and share it or not is the separate question. But I definitely see a lot of folks who are interested in that direction. I think it's probably a little bit pie-in-the-sky to say that we’ll ever have one standard, because in AWS’s defense, they did kind of come first. So they’ve learned a lot about their APIs and not all of their API shapes can conform to one standard at this stage. Like EC2 and S3, if you look at their APIs, they’re definitely a lot less standardized than, let's say, EKS and some of the more recent ones. I think some folks like Azure, Google Cloud have had the advantage of starting a little bit later so they could have introduced some of that standardization sooner.

I think that's why Azure really is pretty uniform. They have this open API spec, which kudos to them, they knew early on that, “Hey, we really want to standardize on this.” So they were able to do that, but it's hard to retrofit that.

[00:44:46] JM: But in any case, a lot of the stuff you can probably – Even if there were some kind of AWS APIs on the long-tail that were hard to get your Pulumi SDK properly updated with, the basics, you want an S3 bucket storage. You want maybe an RDS database. These things are probably not going to change for the most part. If you standardize your SDK around them, you can deliver to the developer the kind of experience that you want. If I understand it correctly, you want to give to the developer a programmable SDK that really simplifies this cloud experience instead of being this nightmare of having to go into the console and have to do these things or to have a bunch of different SDKs that are built by different AWS teams. You have this unified experience in Pulumi that really simplifies things.

[00:45:52] JD: Yeah, absolutely. A lot of our customers have to go to different clouds. So the fact that even though the APIs are different, you’re still using the same tools, you're still using the same languages, you’re still using the same method of deployment. It’s really important to them. I think another interesting thing for us is we let you kind of spin up new environments very easily.

Let's say, in your example, the RDS database. What if we want to test an environment? Well, a lot of our folks that are using Pulumi are spinning up ephemeral environments in their pull requests and just doing testing and validation and then tearing them down. That workflow is important also. This whole how do you do CICD in addition to just how do you write the code, but how do you actually orchestrate the deployments and how do you manage lots and lots of environments at scale.

[00:46:47] JM: If I'm only in AWS, what advantage do I get from using Pulumi over the AWS SDKs and infrastructure as code tools?

[00:47:01] JD: Definitely, we support cloud native technology. If you're using Kubernetes, Helm, any of those technologies, we’d let you have one standard way of going about it. I think

especially compared to cloud formation, which is YAML-based, the ability to use different languages that you're already familiar with is huge. We have a bunch of librarians as well.

Let's say you want to run a load balance service in ECS Fargate. Maybe you're not using Kubernetes. Frankly, ECS Fargate is quite a bit simpler to use if you're on the golden path than, say, Kubernetes. We have a way in 20 lines of code where you spin up an entire Fargate cluster. You build and publish a Docker file in your microservice to a private ECR registry. You spin an ALB load balancer and all the Fargate definitions that use your Docker image.

Honestly, that would be probably 1 to 2,000 lines of cloud formation YAML and you can write in 20 lines of your favorite language. You run pulling me up and sort of magic happens and out the other end pops a Fargate cluster with a load balance service running your Docker container. I think you just get way more done. Honestly, a lot of folks who think they're single cloud today, you'll be surprised down the road whether you're selling your product to customers that are running in AWS and you want to be able to reach Azure customers too, or some folks get acquired, or the economic situation changes. It's like actually standardizing on one workflow and tools that you know in the future can go to any cloud. It's actually kind of a nice insurance policy that happens well.

[00:48:50] JM: If I heard you correctly there, the idea is that because of how Pulumi is architected and because of how the SDK works and because of the integrations with programming languages rather than raw YAML, you just are writing fewer lines of code.

[00:49:08] JD: Quite a bit fewer. In fact, I think the first time we knew we were really on to something, we actually helped somebody go from 20,000 lines of YAML in their cloud formation templates down to 500 lines of JavaScript. Just straight NodeJS code, and the developers loved it because they had no idea what was in those 20,000 lines of YAML. They're afraid to touch. There's so much copy and paste going on. They've really kind of hit a wall and then they moved to this model and the developers are running full speed ahead.

This is a smaller startup where they just didn't want to even have like a separated infrastructure team. They had like an SRE who is really good at networking infrastructure and clustering, but

they really wanted to let their developers just kind of run full speed ahead and just let them do that. That's definitely a pattern that's played out time and time again.

[00:49:56] JM: The easiest comparison to draw between Pulumi and something else I think is to Terraform. How does Pulumi compare to Terraform?

[00:50:04] JD: Yeah. I think Terraform is definitely is the household name. It's the gold standard for current infrastructure as code. I would say the DSL-based infrastructure as code approach. I think a number of things. One, Terraform 0.12 actually introduced some sort of for loop capabilities and quasi programming language constructs, and it's really an admission that my experience with programming language design has always been DSL's work great in the early days until you start pushing around the edges and you realize, "Oh! I actually wanted a full full-blown language," and then you're forced to do these things with DSL's. You introduce kind of quasi programming language capabilities, but they're never quite as good as the real language. Then you start getting pushed in weird directions. Like what if you want functions? What if you want classes? What if you want package management? Well, at that point, you just want a language, right?

So we see that a lot of folks, whether it's because you're copying and pasting too much, or the other side of things is developers oftentimes don't want to learn a new DSL especially if it's a huge step backwards for them. They lose access to the IDE's that they love. They lose access to their test frameworks. Really, I think that's kind of where things start breaking down is kind of like a huge step up for the infrastructure team where they're not more productive. They can share and raise best practices. Then for the developers, they're kind of not afraid to touch it anymore.

Terraform, definitely, they've ceded that provider ecosystem, I think, which is great. That's why we added these sort of adapters so you can tap into those. But, definitely, I think Terraform users also kind of understand why Pulumi is great, because it solves a bunch of problems that they currently have.

[00:51:53] JM: The adaption process of Pulumi for a detailed complex enterprise application, what does that look like? If there's somebody out there who is working at a big enterprise and

they're curious about Pulumi, all these things that you're saying resonate with them. They want an ability to write infrastructure as code in their favorite language, whether it's JavaScript, or Python, or Go, or whatever. They want to combine declarative and imperative code. How do they get started? What is the kind of the entry point for a complex application?

[00:52:31] JD: I'd say Pulumi is open source. So in terms of just how do you get started? It's pretty easy to get started and to try something else. Most people will start by just trying something out, and it's usually like what you are asking earlier. It's, "Hey, let's see if I can spin up a web server. Let's see if I can spin up a new microservice or a serverless function or Kubernetes cluster. I see a ton of people doing it for clusters.

That's usually the first step, is like familiarize yourself with the tool and see what you do about it, and then usually it kind of becomes more apparent where the opportunity to leverage it might be. What that tends to be is let's say you're adding new infrastructure to an existing set of infrastructure. Again, going with your, "It's a big enterprise, a big project," maybe you've been tasked with, "Hey, we need to adapt EKS," Amazon's hosted community service.

Well, it turns out, we have a bunch of packages for that, a bunch of playbooks, in fact, because it's actually not that easy to do in terms of hooking it up to CloudWatch and Route 53 for DNS and ALB for ingress. Usually, it's important to find what is a scoped proof of concept that I can start with and prove that, "Hey, this is going to work for me."

We have a bunch of integrations into existing systems. Oftentimes, maybe you're using GitLab pipelines for deployments. We just plug straight into that, and Spinnaker, and Travis, and Jenkins and all these other systems you might have already be using. We're trying to make it really easy to integrate into your existing systems.

Another area where we make it easy is if you are coming from Terraform, we have ways to coexist. Either you can reference existing Terraform state files. Maybe your VPC is in a Terraform state file. You don't want to go rewrite that yet. But you want to spin up this EKS cluster that uses the VPC. Well, that's easy to do. Then we actually have tools to convert from Terraform source code translators that will actually translate from HCL into JavaScript or Python. We offer a bunch of other tools too, like adapting existing infrastructure so that when you decide

– Like we've had people, like major companies that you would recognize that have said, “Hey, we’re going to just go all-in on Pulumi and we’re going to convert all of our existing live environments on-the-fly to Pulumi.” That sort of thing is possible and we’ve created a bunch of tooling and playbooks to make that easier.

Usually, how do you get started? Have a really focused POC where you can have a clear success that you feel good about that you can then go show your colleagues, you can show your manager and start building up support for a broader adaption.

[00:55:13] JM: I guess just to wrap up. I know we’re up against time, but I’d love to get a little bit of understanding of where the business is going and what the revenue opportunities are, where you're actually going to be making money with Pulumi, the company.

[00:55:29] JD: Yeah. We really focus first and foremost on building an open source ecosystem, that to-date, that remains the most important thing to us, because we think happy, healthy ecosystem, that's the rising tide to lift all boats. That's the thing that we’re really focused on. Frankly, we’re seeing phenomenal success there.

People really do fall in love with the tool. I mean, some of the stuff I see on Twitter just makes me so happy and smile every time, like, “Pulumi gives me superpowers. I love this thing.” So we want to keep growing and nurturing that. I think we made a smart decision early on, which was we’re not going to hold anything back. It's all open source. We’re not going to do core. Instead, we’re going to offer SaaS. SaaS is kind of the business model.”

When you download the open source, it offers, “Hey, do you want to use the free SaaS to store your state and do kind of deployment management and things like that? The analogy I draw is like you can use git without using GitHub or GitLab, but it's so much easier to use Git with GitHub and GitLab and they offer free tiers, and if you want to use it in your team, it's really hard to use git by itself without something like a GitHub in a team.

That's sort of how the Pulumi CLI and SDK works with our SaaS product. We’ve got really cheap option for kind of teams just getting started. $50 month for a number of people and projects and then all the way up to the enterprise edition that has some advanced features

around policy and compliance and identity, SAML, SSO, things like that. We're seeing good adaption. We've got pretty major customers, large enterprises. Mercedes-Benz has been a great partner. We're actually expanding internationally with them

I think the long-term challenge for us even beyond that is, "Okay, we're doing infrastructure as code. How do we still attain that vision of every developer on the planet can use the cloud as a core part of how they're building software?" We're just scratching the surface of that. I think we've got a long way to go, but it's a promising foundation to build on top of.

[00:57:42] JM: Joe, it's been a pleasure talking to you.

[00:57:44] JD: Likewise. Thanks for having me.

[END OF INTERVIEW]

[00:57:55] JM: Today's show is sponsored by Datadog, a modern, full-stack monitoring platform for cloud infrastructure, applications, logs and metrics all in one place. Use Datadog's rich, customizable dashboards to monitor, correlate, visualize and alert on data from disparate devices and cloud backends to have full visibility into performance.

Datadog breaks down the silos within an organization's teams and removes blind spots that could cause potential downtime. With more than 250 integrations, Datadog makes it easy to collaborate together and monitor every layer of their stack within a single platform.

Try monitoring with Datadog today using a free 14-day trial at softwareengineeringdaily.com/datadog, and they'll send you a free t-shirt. That's softwareengineeringdaily.com/datadog. You sign up and you get a free t-shirt. Check it out at softwareengineeringdaily.com/datadog.

[END]