EPISODE 972

[INTRODUCTION]

[00:00:00] JM: Cruise is a company that is building a fully automated, self-driving car service. The infrastructure of a self-driving car platform presents a large number of new engineering problems. Self-driving cars collect vast quantities of as they are driving around the city, and this data needs to be transferred from the cars on to cloud servers. The data needs to be used for training machine learning models. These models must be tested in a simulated environment, which provides more data to be integrated back into the self-driving car system, which is deployed to the cars.

As the cars drive around the city, they can communicate with custom cloud services to get information about traffic, navigation and weather. Cloud services are also used for internal tooling. They can help with automotive diagnostics, configuration changes, deployments, and security policy management.

The software platform used to manage infrastructure at Cruise is a combination of cloud products, open source tools and custom-built infrastructure that is mostly deployed to Kubernetes.

Karl Isenberg is an engineer at Cruise and he joins the show to talk about the engineering requirements of building a self-driving car service as well as Cruise's approach to platform engineering. Carl has a great deal of experience in building platforms. He has worked on Mesos and he's also worked on Cloud Foundry, so he brings a ton of experience to the conversation.

We are hiring a writer. If you are interested in writing about software engineering or computer science, we would love to talk to you. This is a part-time role, but you'll be working closely with myself and Erica. We're also hiring an operations lead. If you want to figure out how Software Engineering Daily works and help us improve, then you can apply for this part-time role as well. Both of these roles, if you're interested, just send me an email, jeff@softwareengineering.daily.com. I'd love to hear from you – And let's get on with the show.

[INTERVIEW]

[00:02:10] JM: As businesses become more integrated with their software than ever before, it has become possible to understand the business more clearly through monitoring, logging and advanced data visibility. Sumo Logic is a continuous intelligence platform that builds tools for operations, security and cloud native infrastructure. The company has studied thousands of businesses to get an understanding of modern continuous intelligence and then compile that information into the continuous intelligence report, which is available at softwareengineeringdaily.com/sumologic.

The Sumo Logic continuous intelligence report contains statistics about the modern world of infrastructure. Here are some statistics I found particularly useful; 64% of the businesses in the survey were entirely on Amazon Web Services, which was vastly more than any other cloud provider, or multi-cloud, or on-prem deployment. That's a lot of infrastructure on AWS.

Another factoid I found was that a typical enterprise uses 15 AWS services, and one in three enterprises uses AWS Lambda. It appears serverless is catching on. There are lots of other fascinating statistics in the continuous intelligence report, including information on database adaption, Kubernetes and web server popularity.

Go to softwareengineeringdaily.com/sumologic and download the continuous intelligence report today. Thank you to Sumo Logic for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[00:04:00] JM: Karl Isenberg, welcome to Software Engineering Daily.

[00:04:03] KI: Yeah. Thanks for inviting me on to the show.

[00:04:05] JM: Cruise is a self-driving car company. That's where you work. Describe how Kubernetes fits into the architecture of Cruise.

[00:04:12] KI: Sure. So Cruise has cars driving around San Francisco, mostly autonomously, and the backend, the ride hailing service works on and other miscellaneous pieces that Cruise works on runs in the cloud on Kubernetes. So we have a lot of different things, a wide variety of workloads and any of them run on Kubernetes, but not necessarily all of them. So a just sampling of workloads, we have some job processing, and some machine learning, and batch jobs, and microservices, and sort of caches, and semi-stateful services, and databases, and all sorts of stuff.

So, basically, Cruise needs a lot of processing power, and one of the way we can enhance developer velocity and productivity is to give the engineers a platform that they can build on top of to sort of accelerate their deployment and getting to production readiness.

[00:05:06] JM: Does it ever make sense to deploy a Kubernetes cluster to a car?

[00:05:11] KI: We don't do it. I'm not sure that I would trust Kubernetes to do that at this point, but hypothetically, in the future maybe. It's not something we're currently investigating. Largely, we use it for large distributed systems. While there are people using Kubernetes for IoT systems distributed, there's a high requirement for quick processing in the cars more so than some other areas. So we use a real-time operating system for our cars running mostly like C++ code.

[00:05:40] JM: The architecture of Kubernetes though, can you imagine a world where some of those architectural features are useful for the car operating system?

[00:05:52] KI: So, in general – Now, this isn't quite my area in AB Engineering. But in general, cars have multiple computers on them these days, and so they need to have some sort of coordination for software and installation and management and stuff. We don't do over the air updates, because we're running the cars in sort of a fleet that they can come back and get software updates in the garages. So we don't have a sort of live update requirement for that.

So we don't necessarily need to have distributed systems that we can roll out rolling updates to. We just load the car software on there, and that's sort of a safer, a quicker way to do that. But if you wanted to do it live like that, you might have a system that you needed to have super high-

availability. So we have backup car, computers and backup GPUs and there's a whole bunch of testing that goes into that. But we don't use Kubernetes for that functionality.

The other sort of option is either you're in like a cluster on the car or a cluster that has like nodes in the car, and Kubernetes wasn't really designed for the use case where the nodes are sort of intermittently connectible. The scheduler model that Kubernetes uses and the distribution is not really designed for that use case.

[00:07:04] JM: Before you worked at Cruise, you worked at Pivotal and Mesosphere. Both of those companies spent lots of time developing platform as a service products, or arguably just open source platform products. You saw a lot about the design requirements that enterprises need for platforms.

Since joining Cruise and seeing what I presume was an entirely new domain of company that had its own requirements for platform infrastructure, what has surprised you? What has been fresh or novel as you are getting acquainted with building platform infrastructure for a car company?

[00:07:49] KI: So I've been doing container platforms for about 6 years at those companies, and then even before that I was doing sort of software platforms for ecommerce stuff. So there's a general requirements almost everywhere for platform services internal to the company or external, depending on the size of the company.

So a lot of the compute requirements are effectively universal, but one of the things that is sort of special about what Cruise is doing is the wide variety of workloads that we run, which is unusual from my experience with other customers I worked at at other vendors. So, for example, we do a bunch of AV processing, probably a significant amount of our time and compute process is taking the data off the car and processing it, chunking it in different units, and then feeding that into different systems, whether that's machine learning, algorithms or whether that's data infrastructure giving mapping data for our applications, or for developers to then go use to replace scenarios so that they can optimize their algorithms or train their systems.

So there's a huge amount of investment behind that sort of idea itself, but also we have the standard microservices that everybody has, applications that everybody has. They need to be highly scalable to handle internet traffic in the future when we have customers in our cars and requesting rides. Then we have a whole bunch of sort of behind-the-scenes processing and tooling internal to engineering that a lot of companies have.

So like CICD and testing infrastructure. I think one of the biggest challenge we have isn't just getting the car to drive safely and quickly, but also to be able to test that. To be able to simulate faster than we can drive in the road and having simulation platforms so you can do that kind of thing at scale using thousands of machines or even tens of thousands of machines to get as much computer power as we need to do it at a reasonable timeframe.

[00:09:53] JM: The idea of platform engineering, this has been done for a long time at Google. It's obviously been on Facebook for a longtime. Netflix has a great reputation as a platform engineering company. These companies that have large amounts of resources, great revenue streams, great engineering teams, companies like Cruise. It makes sense to have a platform engineering team.

Now, we're here at KubeCon. There are a lot of companies that have different requirements, different sets of constraints. Companies like big insurance companies, banks, oil and gas companies, where they do have compute-intensive requirements. They do have complex software platforms. I wonder if you have any general perspective on who should have a platform engineering team.

If I'm a bank, if I'm an insurance company, should I have a platform engineering team or should I sort of say, "Look, our platform engineering team is AWS, or our platform engineering team is GKE. We don't have the volume of engineering resources that Netflix has. We should not try to do platform engineering." When should a company have a platform engineering team?

[00:11:13] KI: So this is actually a question I had to ask for myself when I was in between jobs trying to figure out where I wanted to go. I wanted to work at a place that had an investment in a platform team, because that's what I've been specializing in. So from what I've seen, the companies that have a platform team, whether they should or not, tend to be sort of medium to

large size, and you don't usually see small startups sub-200 with platform teams, because they just don't have enough people to specialize that much.

As companies grow, they get more specialized in what their individual people and teams do. So there seems to be a turning point to me somewhere between like 200 and a thousand where first you start with like either SREs or an infrastructure group that has to like manage giving hardware. Then there's sort of another layer where you focus on engineering productivity, or CICD, that kind of thing. Then sort of the next layer after that is a platform group where you're building tools that are custom for your use cases.

But all of these groups are doing effectively the same thing. They're not necessarily building platforms from scratch. We're sure not. It's more like a lot of integration and then bring your own components and integrate them into the whole.

I usually talk about our Cruise PaaS as a constellation of applications and components and services that all sort of provide value to our engineers rather than something that completely abstracts and hides all the stuff underneath it. Because more often than not, the engineers need to treat them sort of like onion layers where they peel back to the layer to get the functionality that they need.

So some people will start at the top where they want like an application PaaS or a function as a service solution, because it's the least amount of work to configure. But more complicated solutions will need to peel back those layers and go directly to containers, and some systems will need to go directly to the VMs.

So we tried to build from the bottom-up there, and our general policy at Cruise is to only build if we can't buy for backend components specifically. That doesn't work for the car, because that's a brand-new tech. But in platform space, there are a lot of tools off-the-shelf that we can buy. Me personally, having experience working at vendors that did platforms in the past, I have a very good idea of what's available and what its abilities and capabilities are.

I did a talk a few years ago on the container orchestration wars, and basically did an audit on all of the different platforms available while I was at Mesosphere. Sort of comparing Mesosphere's

platform against them. Obviously, at that point, Kubernetes was one of the ones in the mix. So I'm very familiar with sort of the edges of where those tools end, but it's not always obvious to people. So you have to kind of look at the tools and figure out how you're going to integrate them and who's going to do that integration work.

If you don't have a platform team, then you have people doing it themselves. If you have a lot of them doing it themselves, then they're doing different things, and it's a little chaotic and inconsistent. So really a platform team is to sort of bring the chaos together and have some rising tide that lifts all boats.

[00:14:34] JM: This is not directly related to Cruise, and you don't have to answer if you don't want to. But do you think Kubernetes deserve to win the container orchestration wars? Was there something architectural that it did wonderfully or was it a combination of funding and marketing and open source tactical savvy that caused Kubernetes to triumph over Mesos and Nomad and all the other ones?

[00:15:01] KI: I think technology that becomes popular is not just because of its technical merits. I think there's always some other aspect to it. Whether that's marketing, or some popular vendor, or someone who popularizes it on the Internet with a fancy video or something, there's always a little bit more to it than just technical value. I see that as coming back to – The root cause of that is that people don't have time to go research all of the available options and make the best choice. That almost never happens.

Instead, people do a little bit of investment in what they know about and they do a little R&D and research on the Internet and they might proof of concept like one or two solutions and then make a decision because I need to move forward. So sometimes that biases on the things that do one thing and sometimes it biases on the things that do many things, because you needed all those things covered and you get one tool and you invest in that and it's faster than getting something else.

So I think Kubernetes does more than Masos, and I don't think anyone would disagree with me on the sort of scope of what it's capable of. There're more components. There're more features. There are more APIs. So just from that comparison alone, people would take Kubernetes

because it had more capability and you wouldn't have to invest as much. But if you compared it to something like a data center operating system that Mesosphere offered, Mesosphere's solution had a lot of components. Possibly more than Kubernetes does internally at least at the time and tried to satisfy more of that. So there's kind of step value functioning and like either you're getting the kernel of your operating system, or you're getting like a full operating system, or you're getting some proto mix in the middle.

So if you provide more of the stack in a single solution, it's attractive to people that don't have a solution yet. But it's less attractive to people that already have something, because it's harder to integrate, because it's more opinionated. So when you talk about, say, Cloud Foundry or DCOS, these are opinionated stacks. Whereas Kubernetes is sort of less opinionated and less complete than those full stacks, but also offers different value. So in terms of like whether it deserved to win or not, I don't know. But I don't think it super matters. It's what one in terms of popularity, and so is what people are using.

[00:17:24] JM: Right. Well, one thing that was interesting about – If I remember, DCOS had a really good story around we're going to help you get Kafka deployed. We're going to help you get Spark deployed. We're going to help you get these awesome open source frameworks deployed. It kind of turned out that people wanted cloud providers. People wanted – If you look at Cruise's infrastructure, you're all-in on Google Cloud. You kind of want Google Cloud SQL. You want the whole Google Cloud stack, and a lot of people want the full AWS stack. That may or may not have a design opportunity for Spark, or Kafka, or whatever open source thing you want to install. You kind of want it just be all-in on a cloud provider in many cases.

[00:18:16] KI: Yeah. I think when it comes to compute availability, there're a lot of add-ons that cloud providers give you that you can opt in later where you can initially opt in to some low-hanging fruit like an infrastructure as a service solution that has compute and storage and networking. Then beyond that, then you could pull other things off-the-shelf that are valued to you. Getting on a cloud provider that is a platform itself gives you the ability to pull from their service catalog, and that provides value.

I don't know that that's like a fault of Mesosphere's comparing. I think Mesosphere's solution was targeting more of the data infrastructure requirements and people that had on-premise

requirements, and then there was then later a cloud and hybrid focus as well, and not everybody has all of those requirements.

So our general approach is like if we can get it for less work, then that's what we want to do as long as it satisfies our requirements. So we'll bias towards a SaaS product if it satisfies our requirements. Then when it doesn't, we'll go on-premise. We don't have the same sort of data locality requirements that some people have in like the EU. So we don't necessarily need to do all our data processing on-premise. Because of that, we aren't trying to do hybrid Kubernetes. We have a hybrid network that spans multiple clouds and data centers, but we don't have hybrid clusters. The clusters are in individual computer areas.

[00:19:51] JM: At Cruise you vetted the different cloud providers. You tried different managed Kubernetes instances. You tried JKE and I believe some of the other ones. You wound up going all in on Google Cloud. Specifically – Well, I mean, based off of JKE and a lot of Google Cloud related services. What was that vetting process like? The cloud provider vetting process?

[00:20:16] KI: So when I came into Cruise two years ago, some Kubernetes was already in use. So I was put in charge of making sure that became a platform that could be used and not just a bunch of different snowflake clusters. However, they were also using or we were also using Rancher v1 because it was easy to deploy for a small team at the time. Cruise has been in hyper growth for the last like four or five years or something. So the team was much smaller when those choices were made, and it seemed like a good idea at the time, because it was smaller and we could get more bang for our buck.

As we acquired more cloud specialists and people who would be on a platform team, they came with opinions and investments and ideas about where to take the platform. So we did some investigation of cloud platforms, but we didn't necessarily try to audit the whole system. I mean, I had done some of that at previous jobs. So I took that knowledge with me and some of the other people had done similar things.

So it's not like we went and did a proof of concept of all of them. We did investigate some – EKS, for example, was not really mature enough to use at the time two years ago. It was barely out of infancy. It didn't have any upgrades. They're a little further along now, but they're sort of

behind EKS and JKE, and we were using a bunch of AWS at the time and we still have some AWS investments. So I wouldn't say we're like all-in on GKE. Maybe default to JKE – Sorry. All-in to GCP. We default to GCP, but we'll use other cloud resources that are valuable to us because they're better for some reason. We have the hybrid network to support that and make that possible.

I like that approach, and I think that – The only other company I think I've talked to about that kind of platform strategy where you center around one cloud, be you keep yourself open to being able to use the other clouds. Actually, I've talked to a number of companies that had this kind of infrastructure, but Thumbtack comes to mind. I talked to Thumbtack about, because they started on AWS kind of before Google Cloud was really mature. A little bit earlier in the timeline than Cruise. They built a lot of their business logic on AWS, and then they used a lot of data platform services on Google.

So you're talking about like cross-cloud networking infrastructure, cross-cloud whatever tooling. What does that look like? how much infrastructure do you have to put into having cross-cloud functionality?

[00:22:45] KI: A lot. So I wrote a blog post recently with – Collaborated on with one of the network guys at Cruise, and it turns out that there's a lot that goes into creating a hybrid cloud. In ours specifically, we had a high bandwidth requirement for getting data off the cars into the cloud. So we ended up having to invest in some private FIBER 2 and some Internet exchange point of presence.

So that whole set up is pretty expensive and not everybody is doing it, but if you talked to the bigger enterprises, a lot of them are doing something similar just to have like an on-premise and a cloud data. Similarly, if you're using sort of IoT or edge cloud instantiations, you'd have some way to get data between clouds quickly.

So there's a lot of complexity that go into setting up the interconnects, and routers, and the cloud router connections and managing IP propagation between clouds and then handing out IPV4 space, managing those, that kind of thing. So the platform team at Cruise doesn't do that part specifically. We do that for our Kubernetes clusters, but we have a whole other like core

infrastructure team that handles a lot of that, and then a networking team adjacent to that that handle a lot of the sort of making it possible for clouds to work together.

So there is definitely some concerns around having to be close to your compute power. So your storage in your compute, ideally, you want to be in the same place if they're the same application. But if you have some other application that doesn't talk to the main data lake or whatever, it can run somewhere else. You might have some on-premise things. So our like big use case I was mentioning was getting data off the car. So we have on-premise installation and clusters for that, and then they upload into the cloud, and then the cloud takes over most of the big processing for compute requirements.

[SPONSOR MESSAGE]

[00:24:47] JM: Apache Cassandra is an open source distributed database that was first created to meet the scalability and availability needs of Facebook, Amazon and Google. In previous episodes of Software Engineering Daily we have covered Cassandra's architecture and its benefits, and we're happy to have DataStax, , the largest contributor to the Cassandra project since day one as a sponsor of Software Engineering Daily.

DataStax provides DataStax's enterprise, a powerful distribution of Cassandra created by the team that has contributed the most to Cassandra. DataStax's enterprise enables teams to develop faster, scale further, achieve operational simplicity, ensure enterprise security and run mixed workloads that work with the latest graph, search and analytics technology all running across hybrid and multi-cloud infrastructure.

More than 400 companies, including Cisco, Capital One, and eBay, run DataStax to modernize their database infrastructure, improve scalability, and security, and deliver on projects such as customer analytics, IoT and e-commerce.

To learn more about Apache Cassandra and DataStax's enterprise, go to datastax.com/sedaily. That's DataStax, with an X, D-A-T-A-S-T-A-X, @datastax.com/sedaily.

Thank you to DataStax for being a sponsor of Software Engineering Daily. It's a great honor to have DataStax as a sponsor, and you can go to datastax.com/sedaily to learn more.

[INTERVIEW CONTINUED]

[00:26:27] JM: In the blog posts, you talk about some of the internal tooling that you've built. You already alluded to a little bit of the strategy around how you make build versus buy decisions. Tell me some of just maybe one or two examples of build decisions. Times where you had to build some particularly specific internal tool where the cloud infrastructure, the third-party vendors, for whatever reason, were not supplying a product that met your needs.

[00:27:04] KI: Sure. So we have a number of open source projects that we've released in the last year or two. I think our first one out the gate was called RBAC Sync. Generally, it's really just an integration tool tying together two different pieces of software that weren't talking to each other. In this case, it was Kubernetes and needing to use Google groups to manage permissions. So we would have the platform team managing the role binding, and then we would delegate group management to like an engineering manager or a tech lead to make sure that their people were in their groups so that they all had permission. So that added a little bit of self-service, and just tying those pieces together required a little piece of software that we put together.

[00:27:45] JM: Google groups, is that like an email list tool?

[00:27:49] KI: Yes. So Google groups is part of G Suite, and we use – Because we're integrated with JKE and GCP, we use G Suite for our identity provider. Now we use Octa for single sign-on and Duo Security for two-factor authentication, but that's like user identities. So because G Suite has Google groups, you can put people in Google groups and then you can be role bindings.

GCP does this natively. You can do role bindings against Google groups and then put people in those groups so it adds another layer of abstraction.

[00:28:22] JM: That's awesome.

[00:28:23] KI: We do that and build that into Kubernetes.

[00:28:27] JM: That to me, that kind of integration, the integration with the human identity email stack and your infrastructure seems like one of the most powerful opportunities for Google Cloud relative to AWS.

[00:28:40] KI: Yeah, somewhat. AWS has a pretty good system for system authentication of their own tools. But in terms of single sign-on, you usually had to go with someone else. You can get all sorts of identity providers for user authentication. I think the frontier for pushing forward is actually for service accounts. There's a lot of sort of missing tooling around identity providers and policy authorization for service accounts and then integrating them together.

One example is in our RBAC Sync application. We allowed putting service accounts into Google groups, which Google groups doesn't natively support.

[00:29:15] JM: What is a service account?

[00:29:16] KI: So a service account is what services use to talk to each other basically. So they have different means of authentication usually. A user account, you'll use two-factor auth or a user and password or something, and then a service account will usually use a signed certificate from a trusted authority or a temporary issued token or JWT, JSON web token. So that allows the identities to – Sorry, the services to know who they're talking to and know who's talking to them.

[00:29:49] JM: How does a service account and service identity relate to select – I've done a couple shows about the SPIFFE and SPIRE projects, which are for I think assigning application identity or – Well, there's a spec. SPIFFE is the spec for assigning application identity, and SPIRE is an implementation of it. I believe that these are useful because if you give applications an identity, then you can assign certificates, security certificates, TLS certificates that correlate with those identities. That's my understanding of why that identity system is useful. Could you tell me just more about those projects or how that pertains to the things that you've built around service identity?

[00:30:34] KI: Sure. So I worked a little with Joe Beda back a few years when I was at Mesosphere, and he had this idea that there wasn't a solution for service authentication authorization that was open source. Everybody had their own solution and there was nothing that you could provide to like integrate with Kubernetes natively, for example. If you have an open source ecosystem, like the Cloud Native Foundations tools, there wasn't a solution to pull off the shelf for that.

So he spent a bunch of time defining the SPIFFE specification to try to solve that problem, and I think when we sort of started our platform team at Cruise two years ago, that was in its infancy. It was like 0.4 version or something. Sorry. I'm confusing the two. The SPIRE implementation of the specification was in early stages. So it wasn't quite ready to be pulled off-the-shelf and used, and because the specification itself didn't have more than one implementation, and the one implementation it had wasn't mature yet. There wasn't a lot of vetting on it.

So I think there's been a lot of progress on that in the last two years, but it's not necessarily as integrated as our stack as if it was available two years ago. So we have some ways to do workload identity primarily with Vault, for example. So Vault can do certificate signing and basically hand out identities and also then put in subject alternative names into those certificates for validating. That's one of the ways that sort of SPIFFE connects things, is with certificate SANs.

We have some compatibility with that in our tooling, but it's not a primary function of how we operate. It’s definitely a place that needs improvement in the ecosystem in general and we’re trying to move that forward a little bit. But we haven't yet quite been participating in the sort of SPIFFE, SPIRE crew kind of waiting for that to mature a little bit.

[00:32:28] JM: Tell me about the release process at Cruise. So there're different types of releases obviously. So people could deploy cloud services. They need a release process for that. People also need a release process for deploying software to the car. So there’s kind of rollout process there. Can you just tell me about CICD broadly at Cruise?

[00:32:51] KI: Sure. So platform team doesn't own all of that ourselves. We have other teams that focus on some of those pieces. But just in general, I think we have two different flows for most software, and you can think of them as either the deploying to the car workflow or the deploying to the cloud workflow.

So we have a couple of different CIs we use, but primarily like CircleCI and Buildkite, and those usually do the testing for you of the software and the building most of the time. We have other tooling that does more complicated build pipelines as well that's kind of homegrown, and we have a homegrown image builder that triggers off of GitHub webhooks.

So we can basically build the image and then test it in CI, and then we connect through to a continuous deployment tooling. So we used Spinnaker for most of our deployments now, and we wrote a Spinnaker operator around it for managing the sort of multi-tenancy of that. Because one of the complexities with running Kubernetes multitenant is that a lot of the sort of constellation of tooling and components around it are not necessarily as multitenant as you'd like.

Spinnaker has roles and role bindings. So what we do is we create accounts with our operator for each namespace and then in each cluster. So when you go into Spinnaker, you're in this group, and this group has permission to deploy this namespace. This group also has permission to use this account in Spinnaker, and that account has permission to deploy to the namespace too. So you can see what you see in Kubernetes. You can also see and deploy to that with continuous deployment in Spinnaker. Then Spinnaker usually gets triggered with the service account from CI or some automated basis depending on your workflow. So then Spinnaker has a pipeline, and that eventually deploys into cloud usually.

[00:34:48] JM: So we're talking here uniformly about cloud services. Can you tell anything about the deployment to the car process?

[00:34:55] KI: So that's not my area of expertise, but my understanding is that we build our own custom build pipeline and run testing, and there is a large simulation platform that runs what we call the matrix to simulate San Francisco and do a bunch of runs of the car before we ever put the software on the car itself.

Then I think the deployment strategy for getting this off around the car is proprietary. So I can't really talk about that, but it's effectively similar and that it's a CICD flow because of the heavy testing requirements. It's effectively end-to-end tests. We have to run a lot of them, and there're so many of them. We can't run them all every time we test, every time we build, because it just takes too long. It's too expensive. So we had to like run samplings of them to make sure that it's valid and do a statistical measurement to make sure that the car is safe and then do a live test as well.

[00:35:48] JM: Cool. The cloud services – So are there cloud services that the car is talking to on a regular basis or is the car mostly – You think of it as an air gapped module?

[00:36:01] KI: The car is not air gapped. However, it can function without a connection. We generally don't like to operate without a connection, because it removes some capabilities, but it's definitely safe enough to like pullover or continue going down the block if you lose connectivity. We also have a very redundant system. So we're across many zones and many different ways to get information into the car.

So we have a lot of cloud systems. You can think of like – Even like Uber and Lyft have similar scenarios where the app the person is using is constantly talking to the cloud for mapping data, or traffic redirection, or just real-world updates about what's going on. Then if the car gets stuck or something, you might have to phone home and figure out what's going on. So there're all these sort of systems that it needs but doesn't need 100% of the time.

[00:36:55] JM: Right. That's actually a really good analogy, because I have been in plenty situations where the drivers phone doesn't have complete connectivity. I know there're been situations where my phone has conductivity. The driver's phone does not have connectivity. I'm sitting in the backseat. I can see their phone, and their map looks different than mine. I'm like, "Why are you going straight here? You should be taking a right." He's like, "Well, I guess my phone is broken or whatever." But there's a graceful degradation and it's not catastrophic. Life goes on.

Multi-tenancy, that was your middle name until yesterday. When you're deploying this large volume of services, you want to think about how to slice up your infrastructure to use it in an ideal fashion. You got plenty of data intensive workloads. You've got plenty of workloads where you need lots of redundancy. So thinking about how to deploy the different containers, the different services in a multitenant fashion where the containers are not suffering from a noisy neighbor problem that's going to starve them from resources. It's important to think through this.

Tell me about workload isolation at Cruise.

[00:38:19] KI: Sure. So that was a large focus of my talk earlier at KubeCon, all the different layers of isolation that's kind of required and optional for a multitenant platform. I tend to think of multi-tenancy as sort of a spectrum of requirements that you can opt in to depending on what your specific workloads need. So there's sort of a base level of isolation around authentication and authorization, and there's also some more isolation around resource isolation using CPU and disk and stuff.

Then there's more isolation around the integrations that are required for the whole system to operate. Then you can get sort of all the way to system isolation, which is effectively a multi-instance or single tenancy, and that would be like full system isolation. So different workloads have different requirements for them, and different workloads have different requirements for availability. For example, job workloads and batch workloads might be more interruptible than web applications or something that has low latency requirements. So you might want to group your applications in a way where the ones that are similar to each other are isolated from the ones that are different from them so that they don't cause disruption.

On the other hand, if you have ones that are dissimilar from each other, that usually means you can get higher utilization of your compute nodes. So if you're packing in jobs and services on to the same nodes in the same node pool and the same cluster, then you can get higher utilization out of your VMs that you're paying for in the cloud. So there's a tradeoff either way you go and there's a lot of concerns around what you want to isolate and what doesn't matter as much. So we tried to pick off the low-hanging fruit and isolate those and try to get towards logical isolation where the tenants can't see each other, which is difficult to do in a Kubernetes environment.

But, thankfully, this isn't sort of hard multi-tenancy scenario where we don't trust the people who are using the system. If something is broken, we can figure out what it is broken with metrics and go find the team or person responsible and get them to fix it, which would be sort of less viable in a full month multi-tenancy scenario where you have, say, GCP having us as a tenant. If we do something poorly, they don't usually go and rag on their customers and tell them to stop doing stupid things.

So there's a little bit more influence availability to make sure that we're getting to cloud native, and that's also part of the reason for doing multi-tenancy, is to find out where those edges are in our case before we have customers and be able to solve and remediate and add isolation and add security layers in order to make a better product and more production ready faster and sooner.

So we're sort of optimizing for that, but as we get closer to sort of production of customers in our cars and using the backend software in anger, we might value availability more than experimentation and getting our requirements in a row. So as we have reached scale peaks where Kubernetes will cap-out in certain scale aspects, we have found that we needed to pull out domains of workloads and put them on different clusters, or different name spaces, or even just different nodes depending on those workloads.

So those like spawn clusters or those secondary clusters are not necessarily single tenant. They're still multitenant. It's like a group of teams that work closely together that all needed to talk to their – Their systems need to talk to each other and collaborate, but they might have five or six name spaces on that cluster. So it's just a smaller multitenant cluster.

[00:42:12] JM: Kubernetes cluster management, a couple things I don't know the answer to. One; how many Kubernetes clusters do we want? If I am a big company and there's a bunch of people throughout the organization who are doing Kubernetes proof of concepts, do we eventually want to merge those Kubernetes clusters to have all the containers in one big cluster? Do we want to keep these Kubernetes clusters separate? Do we want to have some like God Kubernetes cluster that federal rates into those other clusters? There's that question. There's also a question of the pets versus cattle analogy. Does that extend to Kubernetes

clusters? Do we want disposable Kubernetes clusters or can they be pets? Those are two questions. You can take them as unrelated or related.

[00:43:10] KI: I think it largely depends on your use cases. So I don't think there's a one-size-fits-all answer to that question. In our case, what we've chosen to do is do the hard thing first upfront so that we know what the challenges are for scaling, and we can test that sort of scale capacity better with multiple workloads rather than trying to run synthetic workloads. So a lot of the scale tests that you've seen people give numbers out for like 3,000 or 5,000 nodes or whatever tend to be synthetic workloads and a real collection of workloads will fall over because of something else that wasn't in the synthetic workload.

So we specifically bias towards that to find the rough edges and fix them, and that strategy means that we are defaulting to the multitenant, and then if there's some reason why that isn't working, then we separate that out into another cluster domain. Whether that's single tenant or a smaller multitenant with less neighbors. So that strategy is both from like a cost savings perspective, because we optimize for the thing that saves us the most money until we can't do that anymore, and then we spend more money to do another cluster.

But also, just realistically speaking from an operational sense, when we started two years ago, the platform team was like five or six people, and five or six people can't operate a hundred clusters realistically without an extensive amount of tooling.

For the most part, those tools don't really exist off-the-shelf. You can get things like Terraform to do some of the tooling, but you end up writing a lot of the Terraform. Like kubeadm is not a full solution. It's only part of the solution. Then once you get to the like 10X, to 100X, to 1000X scale in clusters, you have to write some pretty beefy automation to do that. There's a couple of companies trying to do that, but I don't think there is a well-baked solution for that and there definitely wasn't two years ago.

I think we're getting to the point where we need more clusters and we will invest in cluster automation, and we would like to get to clusters as cattle. But I don't think we have a requirement for our production clusters to be super disposable. They need to be disposable for

a couple of reasons. Primarily, like if you make bad choices when setting it up or things change over time since when you set it up.

So like our prime example is like if you allocated too many or too few IP's to the cluster, and that's like a hard thing to change after you've created the cluster. So you might need to create a second cluster and migrate your applications and then delete your first cluster and then replace it. Ideally, you'd have some sort of multi-cluster ingress, which people are still investing in now in order to like do load shifting to another cluster so that you can replace clusters.

But it's not like you're discarding them when they fail the same way you're discarding pods, because it's a self-healing system. It's highly available. It's pretty large and complex. For the most part, the cluster will heal itself. So I don't need to delete it and replace it. So it's really only when you're like root configuration change and you can't do a live upgrade or a rolling upgrade on that.

There is another use case where you want disposable clusters, and that's usually for ephemeral development environments. If, for example, us on the platform team, if we want to spin up a new feature in our PaaS, sometimes we have to deploy an ephemeral cluster and we experiment on it and then we'd throw it away. Now we deploy into the dev cluster that our other engineers are using for development to test it against some real workloads that don't have as much availability requirements.

So there is a developer velocity argument to having dev clusters like that. But we also, like I said, have a have a dev cluster to provide a space for engineers that aren't on the PaaS team to have a target dev environment that they can create a namespace in and go do. So we're getting to the point where we can do self-service namespace creation for developers and we're going to make those disposable in the near future. But we haven't quite got to the disposable clusters yet.

[00:47:17] JM: It kind of sounds like maybe I misunderstood what you said, but it kind of sounds like the fact that we cannot just throw all of our production workloads into a single Kubernetes cluster or throw all of our dev workloads into a single dev Kubernetes cluster. Maybe I misunderstood what you said, but it kind of sounds like a failure of Kubernetes. The fact that

Kubernetes cannot itself figure out the resource constraints of these workloads to the extent that it could spin up new machines or put these things on new machines or whatever.

In an ideal world, it seems like you would not have to manage a hundred Kubernetes clusters with – I realized you're not doing that anymore, but like you wouldn't have to even that'd be something on the horizon, like Kubernetes should give you this one big cluster that you can operate from the default Kubernetes control panel and nobody should have to write this amount of tooling. This should all just be at the Kubernetes level aspirationally?

[00:48:25] KI: I'm not sure I agree with that. Having worked in different container platforms for six years, I've seen a lot of different customer use cases for these things, and there is marketing that says, "You just have one cluster," and it solves all your problems. But in reality, that never works out.

So everybody is using multiple clusters. Nobody is running just one Kubernetes cluster. They might have one production Kubernetes cluster if they're small or they're multitenant. But even us with our like gigantic multitenant cluster, we have multiple production clusters, because we have like them in separate regions, because spreading your nodes across regions means you have cross-regional traffic to your APIs server and if that's really slow and expensive and if you have to do like data exfiltration across boundaries, that gets expensive too.

So there is always going to be a scenario where people need multiple clusters even just for a same environment, whether that's development or production. So spinning up clusters is a basic requirement of using Kubernetes. But that doesn't necessarily mean that the ecosystem has gotten to the containers, those cattle sort of solution yet. I think there's still some investment to be made there.

[SPONSOR MESSAGE]

[00:49:50] JM: LogDNA allows you to collect logs from your entire Kubernetes cluster in a minute with two kubectl commands. Whether you're running 100 or 100,000 containers, you can effortlessly aggregate, and parse, and search, and monitor your logs across all nodes and pods in a centralized log management tool.

Each log is tagged with the pod name, and a container name, and a container ID, and a namespace, and a node. LogDNA is logging that helps with your Kubernetes clusters. There are dozens of other integrations with major language libraries, and AWS, and Heroku, and Fluentd and more.

Logging on Kubernetes can be difficult, but LogDNA simplifies the logging process of Kubernetes clusters. Give it a try today with a 14-day trial. There's no commitment. There's no credit card required. You can go to softwareengineeringdaily.com/logdna to give it a shot and get a free t-shirt. That's softwareengineeringdaily.com/logdna.

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

[INTERVIEW CONTINUED]

[00:51:09] JM: Cruise uses HashiCorp Vault for secret management. Is there anything particularly hard about managing secrets at a self-driving car company?

[00:51:21] KI: So Cruise has a strong bias and priority for safety and security, because they're tightly intertwined when people's lives are at risk in cars. So our cars can't be compromised. Our backend can't be compromised. So we spend a lot of investment in security and validation and testing, and that means that we need to do things in ways that are robust and will get us towards production readiness.

I think that requirement alone, while maybe not the same as someone who has a web application might be more along the lines of people that are processing credit cards. They have similar requirements. So we have an infrastructure security team and actually a whole sort of security group organization in Cruise, and they manage our vault clusters for us. But it's effectively a highly available system, and I HashiCorp has invested in in making that a little bit easier recently, but it used to be a little more complicated. I think now you can group them with Console and get them to be a highly available cluster, but it still doesn't necessarily scale horizontally for load.

There's a lot of complexity in managing the permissions themselves. For example, Vault has a CLI for allowing changing things in an API that you can do that with, but there hasn't been as much investment in the permission management using that API. So like we have a GitOps tool for having the configuration defined in GitHub. Then we can push it to Vault to update networks for maybe 75% of the Vault API, but there's some that isn't really designed for that kind of idempotent usage, like setting up root CAAs or something that has to be done offline or manually once and you don't want to reapply it later.

So it's complicated to make a secure system in general. So we use Vault as sort of the cornerstone for our secrets management, but then just because Vault is something that does secrets management doesn't mean you've solved it with just Vault. You still have to do a lot of management of the role binding and the groups and the policies and the integration and the validation things using Vault. Thankfully, Vault has a pretty good API for that.

[00:53:45] JM: The service mesh abstraction, is this something that everybody that uses Kubernetes also needs?

[00:53:57] KI: I don't know. So we are doing an Istio pilot now to sort of answer the question of whether we can roll it out to everybody. So the biggest I think drawback to the service mesh as implemented by Istio or even Linkerd, is that you have a layer 7 proxy load balancer or a reverse proxy or a forward proxy as like a sidecar daemon next to your application as opposed to doing layer 4 IP switching, like Kubernetes does natively with iptables very EBPF.

So the layer 4 solution is just inherently faster, but it also doesn't allow you to gather the metrics. It doesn't allow you to do quality of service so much. So there's a lot of investment in this space for various players to try to make the network layer provide more value. So many times, you can get more metrics and performance from an overlay than you could from like an underlay, because the underlay does less work. It has less hooks into everything. The same is true with the sort of difference between using IP switching at the layer 4 versus a proxy at layer 7. The layer 7 can gather all sorts of metrics and enforce quality of service and inspect headers and get in your connection and do connection pooling and things like that that you couldn't do at the layer 4 level or is much more difficult.

So I think I know like Google is investing in trying to make some more functionality available in Kubernetes and the ecosystem is as well, because I talk to [inaudible 00:55:33] at KubeCon recently, was talking about how Kubernetes already is kind of a service mesh. It's just not of the feature complete one. So other alternatives provide more functionality. So that's kind of why Istio is popular, is that it has a lot of problem space that it's trying to solve for.

Some of the things we want to get out of it is metrics for ingress and egress, quality of service, isolation between applications and tenants, and then also the ability to do sort of canary rollouts, or also the circuit breaking. So you can sort of decide at the client which dependencies are accessible at any given time and then like remove them from the load-balancing list, whereas if you have a reverse proxy doing all of that, then it doesn't really know enough about the client to make that decision, and every client is then making the same decision as opposed to moving into the client side where you can have a more dynamic system that is based on where the client is and where the service is and where it got a load balanced to. So there're more capability for metrics and observability and making decisions on the fly dynamically. But that also means that there's a latency impact.

So not all applications can tolerate that latency impact. So there's been a lot of work in Istio to minimize that by removing Mixer, which was involved with sort of a metrics aggregation. Then you can sort of decrease the scope of addresses or names that each application or service workload knows about sort of by blacklisting everything by default, and then you have to whitelist your dependencies and the things that depend on you in order to set those up. That requires more operations, but also is more secure and makes the latency hit less bad, because you don't have to propagate everything everywhere.

[00:57:28] JM: Let's say I'm an engineer at Cruise. I wanted to deploy a new service. What is that experience like today? Is it turnkey as beautiful as you'd like it to be? As smooth as you'd like it to be? Hey, you're smiling. Do I need to like sync up with a bunch of platform people and DevSps specialists? How much work do I need to get this thing out the door?

[00:57:54] KI: I mean, I can say that we're not where I would like to be yet. But to be honest, I have never worked at a company where they were where they wanted to be.

[00:58:02] JM: Right. Yeah. Plus, like do you really want people to have carte blanche to like launch a service internally?

[00:58:11] KI: I think there're a couple different ways. That's a permission question I'll get to in a second. But that the first question is sort of answered by when we onboard engineers, we actually have a class that they go through where they learn how to use Kubernetes basics and deploy an example service.

So we do some of that right out of the gate, and we'd like to do some more training around integration of CICD. We'd also like to do some more work around making it trivial to create an application and have it integrate with all the things. Because a lot of times that integration points and learning all the tools and becoming proficient at all those tools enough to use them is a lot of the time rather than writing code necessarily. So it's pretty easy for our developers to come in and in their first week or second week go like use our PaaS clusters. But they might not know the full extent of functionality. I would like to get –

[00:59:07] JM: That a good experience.

[00:59:08] KI: Yeah. So we'd like to get to the point where there are better defaults and there's better tutorials and people can get further faster, but they're already in the right groups have the right permissions get attached to the group that their team is in and then kind of deploy to their dev namespace to play around with and get a field for what their team is doing.

[00:59:30] JM: Yeah. One of the reasons I asked this questions is because I think three out of five companies that I have worked at before Software Engineering Daily. My first like probably two months were spent understanding the Java monolith that I was working on an understanding this internal systems that were built around this Java monolith. I don't think that is even the – I think maybe the majority of developers. They spend their first two months in some kind of painful onboarding process learning legacy systems and internal snowflake systems, and it's not a smooth process. My sense is the industry is improving, and part of the reason it's improving is for – Because of standardized platform systems, maybe.

[01:00:16] KI: Yeah, sort of. I think because of the popularity of Kubernetes, we get people that either want to know Kubernetes because it's good for their career or they already know it from somewhere else that was using it, or they don't want to know it at all and would really rather write software code and then deploy it and not have to worry about how to deploy it.

I think there's usually kind of two different camps there. So us having some shared there and open source tools that are familiar to people when they onboard, lowers the barrier to entry as opposed to if we had a full abstraction that completely hid Kubernetes and all the components. We could build something like that. It would be a lot of work and we could sort of hide all the defaults and make them all smart so that you have a 10-line command or a YAML file in your repo to deploy.

But that would mean they have to learn that interface, and then that hides the underlying interface from them and all the tools that are going around. If you do that and your abstraction is sort of imperfect, which invariably it will be, then you're allowing your developers to know less about the system, which is great for the initial developer productivity. But if you're in a DevOps model where those developers are operating that code, it fall over it later and they won't have known the tools to get there.

So there's a little bit of frontloaded pain in understanding the system that helps you later when you're operating it especially in a DevOps model like we have, because you're not just punting it over to an operations team and the operations team knows Kubernetes. We haven't been doing that model, because of the standard benefit of DevOps is that your customers are not just the people the use your application. Your operators are also your customers, because they use the code more than you do after you've written it as an application developer.

So DevOps sort of treats those both as customers and allows the developer of the application to have more empathy for both of those use cases. If you are also the operator, then you optimize for code to make your operations easier. Really that's what the platform team is trying to make better, because if we let all the teams do it, or if all the teams had to do it themselves, it would be a little bit of chaos.

I think, in general, most companies will try to identify chaos and reduce it in like an iterative pattern where you say, “I have 10 teams, and they all need to do something similar. They’re all going to go build their own solutions.” Then when we've identified what the common pattern is, then we’ll abstract that and make a team to handle the abstraction and build some tool that solves it for everybody. Then they have to do less work. But there's always a little bit of growth of chaos to create the pain that motivates the software development and the new obstructions.

[01:03:12] JM: Well, the platform abstraction of GKE or AKS or whatever, like the Kubernetes cloud things, this seems like a pretty good abstraction for us to be working with as an industry where you can go from working on a Google Kubernetes installation to Amazon Kubernetes installation have some modicum of consistency, and then you just have whatever, like a cloud-specific permission systems and cloud services. But you have the consistent layer of Kubernetes and you kind of have the best of both worlds. If you want to, you can just spin up your own Kubernetes cluster and so on. It’s cool.

I had a conversation with somebody at Heroku recently where they’re kind of thinking like, “Well, they have this kind of Heroku platform, right?” which is very great to work with. If you’re a small team, then there are some big companies on Heroku. But like Kubernetes is sort of – They’re thinking about how to, “Look, what do we do with Kubernetes?” Do we try to make Heroku like a Kubernetes platform, or do we try to, like you said, hide the gears and wheels and stuff underneath Heroku, whether it's Kubernetes or not, and just keep the experiences smooth as possible. It’s kind of an open question.

Anyway, we’re up against time. So I want to ask you about one other thing. Do you think we’re in an industry mass hysteria about microservices? Do you think that like why not have just a couple of monoliths at each company? Why not have like a monolith per team, or a monolith per 15 people, or something like that? Are we going crazy with microservices just to sell more software?

[01:04:55] KI: Well, I don't think it's to sell more software. I think obviously with any sort of a hyped term like that, you're going to get some marketing from companies that motivates behavior in some way or another that might not be ideal for the entire industry. But I think the

intent of microservices was always to separate sort of development lines across teams and really to align to maximize velocity by not having to have everybody work in the same codebase.

So if everybody works in the same codebase, say, like how Google runs their software internally, you have to have a lot of tooling to make that possible and to improve the developer experience of that. I don't think a lot of companies have really invested that way the way Google has. So GitHub, for example, which I think is one of the primary reasons why open source has had a big resurgence in the last 10 or 15 years is modeled for a unit of code in deployment and management at the repo level.

So you can have multiple repos and then have different teams working on those repos, and they don't have to step on each other's toes. They can release software faster, because they can release a smaller chunk of it at a time. If you try to do that in a monolith, you end up building your own build and release platforms, or frameworks, or tooling to slice and dice the repo up to be able to do that. So it's more work spent on the automation rather than using existing tools that are at the repo level in order to do that. So you can easily pull off most CIs available on the market now and get them to build repo code for you, like run a makefile or something and build a container with a Docker doctor file and then publish that to a registry and then wrap into Kubernetes pretty quickly for most small microservices, and that availability of tooling that you can get not necessarily for free, but relatively inexpensively to integrate, allows for a developer velocity that you might not get if you have to build all the tooling yourself for a monolith.

I think still that value still exists. I think there is a little bit of hysteria around making everything – Or the choice right up front is like do we do monoliths? Oh! No, that's crazy. That's old school and legacy. We want to do microservices, because that's a new thing. We've gotten sort of past that now that everybody's like, “Oh! I want to do like functions as a service,” and that's the new thing. That's like my nanoservice. I'm deploying that.

So there's always a risk in taking things too extreme, and I think monoliths have their place and they’re useful for some things, but they're not necessarily useful for everything. Microservices have the same thing. They're good for dividing along team lines or development lines or release lines really so that you can release things in more granular chunks. Then functions as a services is the next level up where functions as a service works really good if you have like a small set of

functions. But once you get into the like thousands of functions, you have a dev ops problem where you can't release and deploy and iterate on those in any reasonable manner, because every team has a thousand each.

So you spend all of your time doing the release management and the deploying, whereas like in microservices, you'd spend less doing that. In monoliths, you'd spend even less doing that. Provided you shipped it all as one unit. But I think when people say monolith, they mean different things. Sometimes they mean I had a repo and I shipped one binary and I ran that one binary. Some people just mean like sharing a repo and building a lot of different artifacts and then deploying them separately sort of like microservices, or a service-oriented architecture.

[01:08:47] JM: Karl Isenberg, thanks for coming on the show. Great conversation.

[01:08:50] KI: Thanks. Thanks for having me.

[END OF INTERVIEW]

[01:09:00] JM: I remember the days when I went to an office. Every day, so much of my time was spent in commute. Once I was at the office, I had to spend time going to meeting rooms and walking to lunch and there were so many ways in which office work takes away your ability to be productive. That's why remote work is awesome. Remote work is more productive. It allows you to work anywhere. It allows you to be with your cats. I'm looking at my cats right now. But there's a reason why people still work fulltime in offices. Remote work can be isolating. That's why remote workers join an organization like X-Team.

X-Team is a community for developers. When you join X-Team, you join a community that will support you while allowing you to remain independent, and X-Team will help you find work that you love for some of the top companies in the world. X-Team is trusted by companies like Twitter, Coinbase and Riot Games.
Go to x-team.com/sedaily to find out about X-Team and apply to join the company. If you use that link, X-Team that you came from listening to Software Engineering Daily, and that would mean that you listen to a podcast about software engineering in your spare time, which is a

great sign, or maybe you're in office listening to Software Engineering Daily. If that's the case, maybe you should check out x-team.com/sedaily and apply to work remotely for X-Team.

At X-Team, you can work from anywhere and experience a futuristic culture. Actually, I don't even know if I should be saying you work for X-Team. It might be more like you work with X-Team, because you become part of the community rather than working for X-Team, and you work for different companies. You work for Twitter, or Coinbase, or some other top company that has an interesting engineering stack, except that you work remotely.

X-Team is a great option for someone who wants to work anywhere with top companies maintaining your independence, not tying yourself to an extremely long work engagement, which is the norm with these in-person companies, and you can check it out by going to x-team.com/sedaily.

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

[END]