EPISODE 1257

[INTRODUCTION]

[00:00:00] JM: Can increase their efficiency and stability and make debugging easier. The company Akita Software observes the structure of programs to visualize, map and manage API behavior. By monitoring the API's in your applications, Akita can catch code changes that may break production applications. While this work is normally labor intensive, Akita automates it by analyzing the source code and logs. To check the observed behaviors against intended specs and contracts, this information can then be generated into maps that help you document and version your API's across the entire service ecosystem.

In this episode, we talk with Jean Yang, founder and CEO of Akita Software. Jean was previously an assistant professor at CMU and a postdoctoral researcher at Harvard Medical School before that. We discuss modern API's, their role in applications and how Akita Software makes understanding and building API's easier for developers.

A few announcements before we get started. One, if you like Clubhouse, subscribe to the Club for Software Daily on Clubhouse. It's just Software Daily. And we'll be doing some interesting Clubhouse sessions within the next few weeks. And two, if you're looking for a job, we are hiring a variety of roles. We're looking for a social media manager. We're looking for a graphic designer. And we're looking for writers. If you are interested in contributing content to Software Engineering Daily, or even if you're a podcaster, and you're curious about how to get involved, we are looking for people with interesting backgrounds who can contribute to Software Engineering Daily. Again, mostly we're looking for social media help and design help. But if you're a writer or a podcaster, we'd also love to hear from you. You can send me an email with your resume, jeff@softwareengineeringdaily.com. That's jeff@softwareengineeringdaily.com.

[INTERVIEW]

[00:01:59] JM: Jean, welcome to the show.

[00:02:00] JY: Thank you.

[00:02:01] JM: You are working on Akita Software. And in order to get into what Akita does, I'd like to first explore the space of API Management. Describe the tool set that is used by a typical company to manage their API's.

[00:02:17] JY: Yeah, sure, absolutely. So other users we've spoken to, a lot of people use tools like Postman for collaborating on calling API's, testing API's, automating the engagement with API's. Sometimes people will use SmartBear or other tools for automatically testing their API's. Some companies will use API gateways for automatically managing what goes on across the API's as well. And the place where we saw there was a gap was this all felt – Well, so I am a programming languages person. And so I'm all about abstraction. But this all felt a little bit low level to me. So Postman, the way I saw it was, there's a lot of great UI and collaboration tools around cURL requests, or single API calls. But what's the sum total of the API calls? Similarly, for a lot of the other tools? It's very point-wise API call management.

[00:03:11] JM: So what are the opportunities for building a better tool? Like, what are the gaps in the tooling?

[00:03:19] JY: Yeah, absolutely. So something that was very inspiring to us at Akita was the observability space, because we felt like what a lot of the companies in the observability space are doing is they're saying, "Well, you can take these traces. And then you can like build on top of them. You can kind of like build pictures of this is what's really going on as your system is running." Your Datadog has all these visualizations on top of, you have logs, but like this is how the logs are over time. This is how your services are talking to each other. Like this is a story that your logs are telling.

And so I started Akita with the question of like, "Well, can we tell like a higher level story about what's happening across all the API's? Can we map out the graph of interactions across all of the API's? Can we have a way to talk about API behavior and not just a single request and responses, or collections of requests and responses? But here's how this service is interacting with that service. And this is what happens if you change some things. This is what the performance bottlenecks might be if you put all these services together. There's this like emergent set of behaviors that, well, I felt was missing, but I felt it was missing, because when I talk to developers and software teams, they're saying, “Look, there's really good like nuts and bolts tools. But the emergent behaviors of our effectively distributed systems are really unclear to us.”

[00:04:48] JM: So let's take a typical API example that we can maybe extend through the conversation. Let's say we've got like a checkout API for an ecommerce website, and the checkout API calls the payments API, the payments API calls some other API. So we've got here just kind of a typical trace of different API's. Explain what kinds of errors or problems that you would catch if you were using Akita.

[00:05:19] JY: Yeah, so I'll explain the vision. And then I'll explain the vision and original motivation. And then I'll say concretely what we do now, because I will fully admit that we're like on step one of N-steps. So the vision is – So, like right now, if you have like a checkout API talking to a payments API, like when you're developing, you develop them kind of separately. There might be documentation that they're related. But in development, these are isolated. Your tests probably – Like your checkout API mocks out the payments and vice versa. Like the together behavior is not quite there.

And then what likely happens today is, especially if you have more API's, a lot of your information about like, what happens when these API's are together? Do I have performance bottlenecks? Do I have weird data flows that are causing issues? Like all of that happens by observing production. So there's this whole great area of principle, testing and production. Someone tweeted at me yesterday, this is what I believe in, not the YOLO kind of testing in production. But principled Canary testing, AB testing, controlled rollout, stuff like that, like that's been the standard practice for if you have a bunch of services, how people are understanding their systems.

And so the key division is like, “Well, look, why should people wait until production to get all this information?” right? Like what's happening today is like things happen in production. You're kind of sifting through logs and traces. You get some visualizations, but what's actually happening? If there is a bug, you don't catch it until things are already happening. And then to back out what really happened, you're playing a little bit of detective with respect to logs and traces that like take a lot of work today to get the full context of like your code level semantics.

And so the Akita vision is like, “Look, at development time, we have enough of a picture of like if you're a checkout service programming and payment service,” we're like, “Alright, this is like what we saw before of how people are using payment service. We watched API traffic going to it. We watched API traffic coming out of it. We built a model.” So when people are coding against it we're like, “Okay, this is like how the API looks. Maybe like even further out, this is how people typically use the payment service.” And then we can alert people on like, “Oh, if you put this sensitive date of birth here, like that's going to go to this other service,” like some other service that that payment talks to. And are you sure you want it to go there? Or if you use some database, we're like, “Oh, well, when you send data off to payments, like they also use that same data resource. So like are you sure this might cause a bottleneck?”

So like the vision is like at development time we can start getting those kind of insights. And then when you're actually running in production, there's like this notion of instead of like logs, or not instead, on top of logs and traces, what I really believe in is lifting things up to the per endpoint layer of like can we talk about endpoints when we talk about performance bottlenecks issues, with data issues, with bugs? Like if a bug arises now like today in payment service, you might be like, “Okay, well, who called me? Where did this come from?” You're tracing logs and traces and doing a little bit of forensics.

What I would love to see, which is the ultimate Akita vision is we get to payment service, and the tool is like – Well, okay. So you were called by checkout service. This is the sequence of API calls that happened. And then you can dig into like logs and traces and things like that too. This is who called them. And here's a step of things that happen. And even here's a link back to the code changes that we observed around this time where this happened. So in short, the long-term vision is sort of having this per endpoint way of tying together this test staging prod, like development time, like the things people actually did at development time with production time behavior.

And so now, let me tell you what we actually do today. Because this is like a very – Like probably pretty abstract and far-off vision. So what happens today is when you're developing checkout service, at test time, what Akita does is it can watch your API traffic. So there's a few ways we watch API traffic. We either have an agent. So if you have network tests for your service, we can just watch that. Or we also integrate with different integration test frameworks so that we can plug into your flask, your fast API, your Django, Rails, etc., and we can watch the API traffic there. We build a model of what we see. So we build a model of the endpoints, the fields, what you would normally see in an API spec, and then we're starting to build more of a detailed model. So data formats is one example of something that usually right now in an API spec people will put like string. But we can detect you have this kind of RFC of date time versus that.

So there's a study that showed 21% of Azure failures that these people studied come from these like tiny data format changes. Like a lot of the people we talked to, they're like, “Yeah, cascading failures for hours because of tiny data format changes.” Like that's an example of something that gets folded into our API model. Eventually, we want to do things like the payments field relates to the date of birth field this way, or like relationships between fields, things like that. Like the eventual API model is very complex. But today, it's like endpoints, data formats, basic stuff.

And then what we do is we also – Whenever things call out to other things, we store those connections. So like if checkout calls out to payment, we keep track of that. And we start building up a picture of how everything relates based on starting from tests if people want, but we also were able to plug in at every stage of development. So we have Kubernetes, Heroku integrations that can listen at staging and prod. And the one feature that we are offering now is diffing across API models. So what people can do on our tool right now is hook us up on every pull request or hook us up into staging or prod to sample at some rate, at certain intervals, generate API models, and then say, “Tell me what's changed between this point in time and that point in time.”

So on a pull request, we can automatically say, “Hey, this data format used to be international phone number. It seems like it's US phone number now. You might be breaking some things.” We can say things like, “This field changed, or this path seems to be different now.” And even the paths, like that might sound like a trivial thing. But a lot of existing tools that diff on just raw traffic alone, like think timestamps, think unique ideas, like those things are all false positives right now in diffs. And so a lot of the work we did initially was just like get rid of that noise. And then we're starting to build up our models to more fancy things.

[00:12:01] JM: So I think the main thing we could focus on right now is the process of like I've got an API in production, and your agent is able to understand and record traffic to the degree that you can detect breaking changes. Am I understanding it correctly?

[00:12:21] JY: Yeah, yeah, that's exactly right. So like what we do, the goal is to detect regressions of interest by watching API traffic and modeling the API traffic in a way that we keep false positives low.

[00:12:34] JM: Okay. And what goes into that agent? What goes into the monitoring of an API that's already stood up?

[00:12:41] JY: Yeah, that's a great question. So one of the big goals we had for ourselves was to make this as passive as possible. So like introduce no production overhead if we can. So our standard agent uses pcap filters to passively watch traffic. So they're also called EBPF filters. But essentially, they are packet filters that watch the traffic going by. Look at what happens and then captures the events of interest.

[00:13:07] JM: And let's talk a little bit more about building that. So can you just tell me about some of the engineering difficulties in architecting that observability tool?

[00:13:14] JY: Yeah. Yeah, that's a great question. So I'll zoom out a little bit. And I'll say like the agent is one part of it. And then like how we get the agent, like how we like insert ourselves in like a non-invasive way as possible into like all parts of the system to watch traffic where people might want. So watch traffic has been like the main challenge we've been facing these last few months. So that's a great question. So for instance, to get stuff running in various staging and production environments. Well, I think one thing is just like where do we get it to run with the right permissions?

So we recently released the Kubernetes docs, for instance. And so it's like, “Where do we sit in the container? Or are we a sidecar? Do we run as middleware?” Like there's like various tradeoffs that we considered when architecting that. We ultimately decided to run as a sidecar because it seemed very standard. And then there's like a pretty standard way of doing that. But I think like that's one of the easier ones. Another one that was harder was some of our users were like, “Alright, we want to run on Heroku.” And Heroku doesn't actually let you capture traffic using pcap. Like you can imagine, there's a bunch of like cloud platforms where this pcap way of watching traffic is actually quite invasive, right? And so to people who, I guess, have tried to do this network level programming, it may be obvious that not all systems will you just get in there and watch the network passively.

And so for Heroku, for instance, we had to build some middleware approaches. So I personally did not build this middleware. So forgive me, for anyone who's listening who's like, “That's not how that works.” But essentially, for that we had to build a daemon that that middleware could talk to you. So essentially, like my understanding of how this works is like there's this daemon that runs Akita that can be like that middleware can talk to. And then there's middleware that sits with these systems. Gets wind of the traffic that way. I'll just leave it at that level of abstraction. And then talks to the Akita daemon to register like, “Hey, these requests and responses were called.” But it was extremely non-trivial to both build and design the daemon because you need something that's like – I think both from a usability point of view, right? You want the user to be able to like have certain amount of control over the daemon without having to script like a bunch of stuff where they're like talking to the daemon all the time, the daemon needs to be babysat like by some script. And then at the same time like you want the middleware side to be as non-invasive and easy, like, close to one line change as possible for whoever's using it.

And so, yeah, like even across different staging and production infrastructures, we're definitely not all the way there yet and covering all of them, but every integration we've built, I've been like, “Ha! This is a really interesting technical insertion problem,” because what you're essentially doing is you're balancing kind of like how do we build this so that we're not – An easy way to build it would be proxy everywhere, kind of like inject yourself, like introduce overhead at the expensive like you can just build this really, or like at the benefit of you can just build it really fast. But what we wanted was like, “This is long running. We can sit there. No one will notice us.” And so it's taken quite a bit of work to be as invisible as possible across all these systems.

And then the like thing that I've also thought was really fun, which I also personally did not build, was our test integrations. So there are people who were like, “Hey, we want to run you as early as possible. But we've got like flask tests, and they don't call across the network at all.” We're using flask to manage our requests and responses. So we also built stuff that got in there. You can swap out the flask test client with like one line, and then we listen to your requests and responses going through there and we record them.

[00:16:49] JM: And as far as maintaining those recordings, do you like just save all the recordings to a database? Because I imagine, in order to discover that an API response has changed, you have to have a database of some successful API responses.

[00:17:06] JY: Yeah, this is a great question. And it's actually my favorite part, and parts that are being actually worked on. But I'm really glad you asked that. So on the user side, how everything gets saved is as HTTP archive files. So I recently discovered – Well, not recently, but through the course of building this, we discovered this format that's really surprisingly ubiquitous. Like all of some category of services use HAR files for like performance and other debugging. But they're this sort of like nice universal format for storing requests and responses. So like how our tool works is on the user side, they either run the agent, or they run a collection through one of our integrations. Those all generate HAR files. Then the higher files get shipped to us. And also here, something that I'd like to point out is we only ship obfuscated HAR files to us. And so at that point, like that was also an engineering challenge. And as we're talking about engineering challenges, like one, how do we work with obfuscated HAR files? But there're some interesting things there.

But then in our backend, like in the Akita cloud, what gets stored is essentially traces that got made from these obfuscated HAR files. And then these traces are tagged in certain ways. So the traces are basically tagged with like if it got run on like a pull requests, or if you've got run on this service, there's metadata that gets stored with each trace. And then models get built from the traces. And then you can diff those models.

And so like, essentially, what's in the database is like there's just a bunch of traces with other information about it. And then what we've been in the process of building user interfaces for are how do people want to manage these traces? Like what do they want to even diff? We started out with like, "Here, you can diff across pull requests." And people were like, "Well, I want to run this in staging now. But there's no concept of a pull request in staging. So do we demarcate pull requests in staging?" Or do people want this hourly or things like that? And so that's something where we're currently working with our users pretty actively on trying to figure out, “Are we giving them the right interfaces for managing their API traces?”

And then there's the question I think that goes back to what you're asking, which is like what's considered good? If it's a breaking change, like what's our point of reference? And so that also different users have different preferences on. Like some people are like, “I always want you to update stable to like the latest one that actually got checked in.” Some people are like, “No. No. No. We say what's considered stable.” And so something we've been working through is like what are the right defaults there and how do we want to expose options to people so as not to overwhelm them?

But yeah, like the user interface questions around how to manage like – I guess the way you can think about us is like the way Datadog takes like a stream of logs and helps you make sense of it. We have a stream of requests and responses. And you can sort of take them off the shelf, build models, how you want. Say start here, and they're building a model, and then diff them. And then there's a question now of like, well, how do people want to manage them? What's considered good?

[00:20:03] JM: Are there other tools on the market for detecting this kind of breaking change? And how do you differentiate yourself from the other things on the market?

[00:20:13] JY: Yeah, that's a great question. So in the last six months, we've seen especially like Postman had to a blog post that said, “Well, if you use Postman, here's how we can help you identify removed end points and things like that.” And there's a company Optic, who was previously on your show. They're good for API's. And they've also started talking about this is how we can help you catch breaking changes.

There's also this tool called Diffy that came out of Twitter that helps you AB test by essentially comparing traffic going to and from services. So if you migrate a service, it'll tell you like, “Here're all the traffic differences between that service and this service.”

To us the big problem to solve, well, there are a few. I’ll say the two main ones that I think are relevant for this conversation. One is how to reduce false positives when you're detecting breaking changes, and like how do you as accurately as possible capture these breaking changes. And so our belief is that if you're just diffing on specs people wrote or API calls people new to make, you're catching a small part of what you could be catching total. So like data format changes and things like that don't necessarily get captured there.

On the other side, if you're just diffing on raw traffic, there's a lot of stuff that changes every time. So timestamps are always going to be different whenever you change things. Unique IDs are always going to be different whenever you change things. It's my understanding that a lot of these tools right now, like you have to filter that out by hand. We've put a lot of work even for those detecting with like very few data points. This looks like it could be a unique ID. This looks like a timestamp. Because like it's easy to generalize if you have like 100 and you're like, “They're all different across these.” But often the cases you only have a few calls. So it was non-trivial to get that to a good spot.

And so with some of these other tools, you're kind of doing all this annotation by hand right now. And then the other thing is like, moving forward, we think that like this kind of modeling is going to be crucial for like usable change detection. So like if you break an implicit contract, for instance, like for this kind of device type for this field, you need like this range of things for that other field. Like that's not stuff people are going to write. That's going to be stuff that's also like extremely noisy to detect. But that stuff that my team and I, like we've done a little bit of R&D into there's some stuff we feel pretty confident that we can do well across a lot of data, a lot of systems, a lot of data, but like that's the part that we're really excited to get into as well. And so the tools that do like workflow management collaboration for these, like the model part we think we still bring to the table.

And then the other thing is, I mean, I'm really excited about this abstraction that we're building that’s like the way existing observability tools are like, “Alright, we you have traces. We give you these visualizations.” We're like, “Alright, you have endpoints.” We’ll tell you things like authentications, changes to data type, all these things. And so giving people like a way to take raw traffic, shape it into models, and then manage that we think is like a very helpful way to manage complexity, understand complexity, see changes in their systems.

And so, like so far, we're the only ones to think about things that way. I mean, I would love to see other companies do it, because I think that the more tools sit at this layer of abstraction, the more people will be used to seeing it, the less we'll have to be like, “Alright, this is what it means to do that.” But I think that's a new take we have on understanding these complex systems as well.

[00:23:36] JM: Can you tell me more about the onboarding process for – Like if somebody's listening to this and they're thinking like, “Yeah, this sounds quite useful. I'd love to have better API observability.” What is required for them to get started?

[00:23:56] JY: Yeah, sure. So they'll do like the best case version. And then I'll talk about some things that might arise, because various users who like got fell into the other cases are like, “But wait, we got left out.” So like best cases, you have an API for a service that you own and you kind of know like how to point traffic at it. You have traffic that you can point at it. And so if that's the case, you can sign up for the Akita beta. I'll let you into the beta. You get an account. You fire up command line. You log into Akita. And then you get your service running. You say, “Hey, Akita, this is the port that I'm watching traffic.” And then Akita captures that traffic. That's your first API model. Then you can start diffing with that. You can start collecting more, etc.

And so the fastest we've seen this happen is like 5, 10 minutes. But I will say that not everyone is in such a smooth position. So there are a few other things that might happen. One is like you might not actually own the API you run. Like the service is running somewhere else. You can't actually co-locate an agent to listen to traffic. In which case, you can either run a proxy or a browser. Proxies and browsers all capture these HTTP archive files. And so for those like – Again, figuring out you’re in that case, if you know you're in that case, that should also be like pretty quick. And then this should be few minutes. You can capture that. You can send that up to our cloud. It'll work.

I think the harder case is if you're like, “Alright, I have traffic to go into the API, but I can't install you where I need to.” So like you can only listen to like test traffic or something like that. In those cases, what we've recommended people do is, “Here, just like check out a demo system we have.” But if you want to run on your own traffic, then we have some integrations now for like Python and Ruby frameworks where you would drop a one line change into your system. We would watch that test traffic. You would upload that to the cloud. And then that would work. I would say this is probably like longer than 10 minutes, because you'd have to get some stuff to work.

And so for the first run, like there's a small range of what that would be, but then I would say like the longer thing is people are like, “Alright, I've run out on one service. How do I actually like collect diffs over my whole thing.” And so then there's a few things you can do. So initially we're like, “Alright, just integrate us into your GitHub, and then we'll just diff for you.” What we learned was people were like, “Wait, wait a minute. Like you're basically asking us like marry after one date here. We want to run you for a while before you integrate us into GitHub.” And so now we're starting to beef-up our instructions for like, “This is how you run us in CI for a while and get reports from using Akita before you fully integrate us and get like comments on every pull request.” This is kind of like how you try us. Like try us on your latest version of the code and try us on like another, like check out something that was like some version ago, stand that up, run it. If you're running us in a stage environment, that's like a little bit easier. So just like run us every like N-interval of time and we'll tell you what changed.

There is like some config people have requested that we've built in. So if they want to ignore certain URL prefixes, if they want to ignore certain kinds of things, that would also be part of the setup. But I think like quick setup, like in short, quick setup, like you can get a taste of Akita in a few minutes to actually like get it in the right places to play. Like we can work with you on that, but that takes a little bit longer. But overall, like the intention is for it to be fairly non-invasive. So it should be pretty quick.

[00:27:14] JM: When you're talking to potential customers, is there any pushback due to tooling fatigue? Just beset by so many observability tools, people don't want to integrate with more? Or are they just eager to stack on more observability?

[00:27:33] JY: Yeah, that's a great question. I think we get a biased sample. I think that there's probably like – I would believe it if there are like thousands, millions of developers out there who are like, “We're tired of tooling. We just ignore Akita.” But like most of the people we get who show up, they're like, “Alright, I use X, Y, and Z.” So a lot of our users also use New Relic or

Datadog or some kind of observability monitoring tool. And those are the ones who are often the clearest about, “Here are the gaps.”

And I think like what you said about stacking on tools is right, because like the people who I actually talk to live will ask, “Hey, like what's your tech stack right now? What's your tool chain?” And usually, the people who show up to us, they'll run like the Panda bot. Some of them run Sonarqube. They'll run some kind of static analysis. They're like, “Alright, well, static analysis left us in the lurch here, here and here.” So then we have like these one-plus observability, or monitoring tools in runtime. And here are our gaps. And so here's like the shape we think you can fill.

And so at least for the progressive people who are adopting our beta, it seems like they're running one-plus static analysis tools, one-plus observability tools, and they have a decent idea of where the gaps are. I think that people who are like, “I have too many tools,” probably aren't showing up at this point. And then people who are like no tool sometimes show up, but I think they're less convinced this is the one tool to run, because they are not running any other tools.

[00:28:56] JM: So take me inside the product development process as it stands today. So you got kind of this API breaking change detection tool. That's obviously useful today. What are you facing right now and what are you working on? What are you iterating towards?

[00:29:15] JY: Yeah, that's a great question. So I'll be very honest. We're early enough in our beta that there's still stuff we promise. Like last November we're like, “Oh, yeah, you want to do this? We'll do it.” We're still building on some of this. And so this includes – So there's a few parts of what we're working on. So to actually get a usable tool to automatically detect regressions as low-impact as possible for install, as low noise as possible. One, well, there's the low-impact install. So we've had users who showed up last fall who were like, “Hey, like we tried this, this works great on our one system. We want to integrate this now, but we want to run it on like X.”

So like one of our next ships that are coming out is Django on Heroko. Like if you want to run on Heroku with Django, we have to install the middleware. We need it to build a daemon. So like they cannot run this on the full complexity of their systems without this integration. And so integrations are one thing we've had to spend a lot of time building. Well, like one, because it's necessary. But two, we didn't just ship like the quickest thing a lot of the time, because we're like, “Alright, these are people who are requesting this for real without overhead, all these stuff.” So we had to take all those considerations into place.

And then there's, “How do we actually have low-noise diffs?” We had various ideas for like, “Yep, we think we can be lower noise for reasons X, Y, and Z.” But we get requests like, “Hey, I'm getting noise for these reasons,” and we're like, “Oh, man, we got to fix that.” And so as we're testing out with our beta users, we're learning all the dimensions that we like actually need to do better modeling to do better automation to actually cut down on the noise. And so that's one dimension that I think like we're focusing on that I don't see other people focusing on. And then there's like how do people actually want to run us in a very pain-free way?

And so like to be very honest, I think our early users, they put up with their share of pain. They'll like Slack us on like a Sunday and be like, “Yep, I spent my weekend like trying to make like – To script you in here. But if you gave me this flag, my life would be a lot better.” And we're like, “Oh, my gosh. Thank you so much.” But like they're kind of cobbling together a bunch of pieces. And so we're like, “Okay, to actually be like a widespread, easy to use, good developer experience product, we kind of need to meet developers here, here and here, where they are.” And so like part of this is building interfaces for managing like API traces across different environments.

And so like once people started running us like not just in tests, but in other environments, they're like, “Hey, how do I even tell you that like these are all the same thing and like this is what I care about across my environments?” And so we're like, “Yeah, good point. Good point.” So we've been building a product to meet them there. And then there's like, “Okay, well, I care about these changes, but not those changes.” Or, “You're finding this stuff. But I totally don't care.” Or you're just like – An example is in the beginning. We're like, “Cool.”

So before we thought about things as API models, we thought about things as API specs. And I think that's an iteration that you saw previously, Jeff. But we saw them as API specs, where we put a bunch of random annotations in them with all the other information that we had inferred. So we're like, “Here's your spec. And then here's data formats, link all these other properties.”

And people are like, “What the heck, man?” Like I don't write specs, because I don't want to read them.

And so like an extreme and proven we did was we stopped showing people the specs. We started showing people like here are insights. Like we break down your authentications. We break down request type, response type, data formats. We let you filter, we let you search. So I think a lot of our product development has just been driven by like watching people use stuff. Seeing like what are they really trying to do? Giving them the things that they're trying to do much more easily? So like if we start seeing a bunch of people scripting something, we'll be like, “Yeah, we need to do that.” Or if like no one is showing up to some page that they asked for and then we talk to them. And they're like, “Well, I wanted this information, but not this giant dump.” Then we dig in, and we try to fix some stuff.

So I think a lot of the core technology we like finally built up over the last couple years, but a lot of like what experience do developers really want? That's something we've been actively working on, we're still actively working on, and I think like getting it right will be very, very powerful.

[00:33:19] JM: Tell me a little bit more about the process that you're discussing where you get a lot of feedback from customers and sort of learn what to fix or what to improve on? What kinds of issues have you discovered that way? And how has the iteration process gone?

[00:33:35] JY: Yeah, that's a great question. It's something I'm very passionate about. I wrote a blog post a bunch of months ago about what's the role of developer experience and experimental programming tools stuff. And I think it's very, very important. And so I'll say a couple of anecdotes about things we learned. And so the whole reason we're building this product is we're previously building a different product. But this is the part of the product that people actually liked. So we started out building an API fuzzer, because people were like, “We want to know blackbox. What's going on with our services. We want to know like what's coming in what's going out? And we want you to be non-invasive.” And so we're like, “Okay, cool. Like maybe we can just like probe it with a fuzzer.” And people were like, “Yeah, that seems reasonable.” But then what we learned was people were like, “Well, we don't have API specs. So you can't prove it.” And so then we had to build this whole API spec infrastructure. And I don't know how much – You know about fuzzers, or the audience knows about fuzzers. But in order to know how to fuzz an API, that like requires a lot more than just knowing what the endpoints are. You have to know how the endpoints are related. What order they're called? How to generate data? Fake data for all the types.

So like that's how we got started building a lot of this like inferring, like specifically the relationship of API functions to each other. And then people are like, “Wait a minute. If you have all that, that tells us like most of all we need to know about our API.” And we're like, “Hmm.” And then like we heard that from one person and we're like, “Okay, we're going to survey all the other people.” And they're like, “Yup, yup, pretty much.”

And so like we even had like contracts at the time where people were like, “We will pay you to fuzz our API.” And then we went back to them. We're like, “Okay, stack rank. Like fuzzing, like API learning, data format analysis, all this other stuff. And like all the stuff we had built to make the fuzzer work, it was just like higher on the stack rank. So we're like, “Okay, like, first big learning. We're not doing the fuzzer yet.” We still have a fuzzer. It's on a shelf. But we're going to make this other stuff work, because it seems like it's way more valuable. And then as we were doing that, like the regressions actually came through that as well. So people were like, “Yep, we want to use this.” A bunch of people showed up. And more people showed up to use our API. Like back then it was like API spec inference. We're like, “We don't know. Something's not valuable to us.” But like people seem to want it. So we'll just like put it out there. See what happens.

And then we're like, “Well, okay, like why are you collecting so many specs?” Because, really, it sounds like a spec is something you want like once. People are like, “No. No. No. We want specs like every time we check in our code.” And so we're like, “Well, on the one hand, that makes sense, but like what are you really trying to do?” And what we got from that was people are like, “Well, we want to diff those specs.” But then they showed us how they are diffing with specs right now. And they're like, “This is bad. Like this makes no sense. Like this diff is so noisy.” And we're like, “Okay.”

So then we caught on to the fact that people seem to want specs for regressions and they wanted something. But what we were doing wasn't there yet. So then we surveyed more broadly, and people were like, “Yes, we want regressions on API behavior. But like everything we have is really noisy.” So then we like embarked on this very long journey of like making it less noisy, which I'm describing to you. Because like I think in the beginning we're like, “Oh, yeah, everyone is like it's simply a matter of just different specs.” And then we're like, “It's way not.” Like we got stuck with a timestamp issue. We got stuck with a unique ID issue. We got stuck with like – No one wants to see diffs of YAML files. Like I think we're lucky enough that we have very blunt users. They'll like call us. They'll be like, “Here, I will on video tell you like this is so bad.”

And so in the beginning they were like, “Look, we don't want to read a spec. Who wants to read the spec?” They would show us their spec and they're like, “This is terrible.” And we're like, “Okay, so then like why do you want us to make you the spec?” And they're like, “Well, what we want is like we want to know about authorizations, like bla-bla-bla-bla-bla changes.” And we're like, “Aha.” And so like that's how we built the visualization on top of it. And now for diffs, they're like look, “Like you're just dumping YAML diffs. Like you need to do all this other stuff.” So like that's what we've been working on. We're still iterating on that.

But pretty much like every, like product improvement has been like at least one person like shows us like what they're doing on their data and they're like, “Ugh, I hate this.” And then we're like, “Okay.” Then they're like – Many of them are kind enough to like let us play with that data too. Then we like come back with something. And we're like, “Is this what you wanted more?” And more often than not, it has been an improvement. But I think it'll take a few more iterations to kind of get like the right user experience for engaging with these things at the endpoint level.

[00:37:50] JM: Tell me about the hardest part of running the company thus far.

[00:37:56] JY: So I think that for us, it's been like a collision of a non-software thing with a software thing, which is for Akita, it's actually been getting the right team to work on bringing great developer experience for an extremely technical problem. Because as you can see, we're like extremely user-driven, we're extremely product-driven. But like pretty much everything we've built has been really freaking technical. So like how do we insert like the best way into a Kubernetes environment while we like ideally want a daemon. We want this. We want that.” But like the pieces are like super, like for lack of a better word, hardcore systems engineering problems. But like the experience we want is like very, very clean. And so we're actually on our second team iteration. I think, like I mentioned, we sort of pivoted from a fuzzer. But actually pivoting our team has been the harder thing, which was when I first started Akita, I was like, “Well, I really believe in good developer experience. I should get people who are like very product-oriented, very user-focused, very developer-focused engineers,” and like that's what we had for v-zero of our team. And it turned out to not quite be the right group of people to work on such a gnarly, technical problem. Because I think that to work on something that requires both like technical innovation and like non-trivial product development. It’s not like “Hey, we're building like a better GDB or something, same GDB interface, better debugging.” It's sort of like there's on both fronts, there needs to be innovation. I think that like maybe this is like a very arrogant point of view. But I'm like, “Well, like innovation is innovation,” like you just innovate. But like getting the right people in place, you kind of like get excited about both, and they're charged up. So like there are people who are like, “I love product innovation, but if I have to innovate technically, like that makes me nervous. I don't like it. That uncertainty is not my cup of tea.” There're also people who they're like, “I love technical innovation, but like just fix the products for God's sake. Like tell me what the product is. And like I'll innovate technically.” But to like get the right team in place who they're like, “Yeah, like I will answer our user requests at like 1am on a Friday night, because like I love that.” I'm not saying like people have to work 1am on a Friday night. But people who are like, “That's what I live for, and like I'm so excited to like, go and like do some like really intense like systems hacking to like make that user happy.” Like that is like a very rare like intersection of things. And fingers crossed, I think we finally gotten the team who like – It gets really excited. We're like moving fast. We’re doing this stuff. But that was really hard to figure out like what is the right DNA for that.

[00:40:35] JM: Any broader reflections on where you see infrastructure software going, trends and opportunities or problems?

[00:40:45] JY: Yeah, so there're two that I've been thinking about a lot. We’re part of both, as you might expect. But one is, as there's been the shift from – So like building and shipping is sort of like how people think about software today. Like you have dev, you have tests, then you ship. And even agile, just condense that cycle. But there's still this notion of building and shipping. But really, this has become building and running. And so like with the rise of SaaS, like it's a lot more complex than building and shipping. There's a lot more I'm operating software. I'm operating software that I didn't write. I'm operating software that I wrote in concert with software that I didn't write.

And so this is where I see a lot of existing tools just kind of falling down. So like your test tools, they like obviously don't help you test against like Confluent, Kafka nodes or things like that. And then even existing observability tools, they're kind of like built from this tradition of like they expect that, okay, like if you can get logs, that's okay. But ideally, you like to find all these spans, and you like instrument your code very precisely.

And so the thing I had gotten really into at the beginning of Akita was I had this intuition. Like things – Well, okay, so it was an intuition from talking to a lot of developers. But I had the sense that things need to be blackboxed. And it took me a couple years to realize the reason they need to be blackboxed, because in this shift to operating, like things are a blackbox now. That's why. And so I think that's a big trend. I mean, I think that something I had a lot of trouble explaining to people, because I come from like a static analysis, dynamic analysis background, like you instrument, you have access to everything. People are like, "Jean, like what the heck are you doing now?" And like the technical way I describe it now is I'm traditionally like an expert right of spec and the other tools analyze against it. I think if things are blackboxed, what you really want especially – Yeah, what you really want if things are blackboxed is you infer the specs by watching. And then you like either show the results to experts. Or better yet, you just detect deviations. I think regressions are extremely powerful if you don't know the spec.

And so as things shift to being out of the control of the developer, or you don't know the spec, you can't look inside, I think like this is both in terms of the product landscape and in terms of technologies, I think that the kind of thing that I'm talking about is going to become more and more important.

And then the second thing is something I tweeted about yesterday. I've been thinking about it a lot. Is a lot of developer tools that have been successful so far have been what I'll call simplifying or abstraction tools. So it's Stripe-like payments. They're really hard. So we'll just abstract them away from you. Or EC2, or managing the cloud, managing a data center is really hard. So we'll just abstract that away from you. That can only get you to a certain point. And so I think that we've reached the point where like – I think Liz Fong-Jones tweeted AI ops is trying to be a simplifying tool when they're actually a complexity and racing tool. So I had this tweet where I said, "Look, like there're really two kinds of dev tools; abstraction, simplifying tools and like tools where you just have to sit with the complexity." So the classical example is like a debugger. Like your debugger can't just like – You can't have like push button, like I find the bug for you, right? Like the whole point of the debugger is it lets you explore your system. And I think that like most of the tools for distributed programming up to now have been simplifying tools. They're like, "We'll abstract away your cloud. We'll abstract away your infrastructure. We'll will abstract away this, that, the other." And I think that like what this latest generation of observability tools has been about is like, "Look, you cannot always abstract away. You need to look inside sometimes." And I think that's going to become more and more important. Like I hope people realize that they can't have like silver bullet abstract away for everything. Now that all systems are distributed systems, we actually need better ways of engaging with them. And so I think that's a trend that I've seen start. I hope it gains a lot of momentum, because I think that we can't just keep abstracting our problems away forever.

[00:44:41] JM: Cool, great answer. That sounds like a good place to close off. Jean. Thanks for coming on the show. It's been a real pleasure talking to you.

[00:44:46] JY: Cool, thanks.

[END]