EPISODE 1226

[INTRODUCTION]

[00:00:00] JM: The complexity of building web applications seems to have grown exponentially in the last several years. This added complexity may bring power, but it can also make applications brittle, costly and difficult to maintain. Suborbital is an open source project with a goal of making web application development simple. Its flagship project is Atmo, a platform that integrates three underlying projects also built by Suborbital. Vector allows developers to write self-contained functions called Runnables to handle business logic which are then built into WebAssembly. Atmo then automatically scales out a flat network of instances to handle traffic using Grav, a meshed message bus; and Reactr, an embedded job scheduler. Together these projects make it simple to create powerful WebAssembly-based server applications without worrying about infrastructure or writing boilerplate code.

Connor Hicks maintains the Suborbital open source project. By day, he is a developer and product discovery lead at 1Password. He joins the show today to talk about the exciting potential of WebAssembly, how Atmo is producing new design patterns and why we should think differently about complex web service systems.

[INTERVIEW]

[00:01:07] JM: Connor, welcome to the show.

[00:01:08] CH: Thanks for having me. Happy to be here.

[00:01:10] JM: We have done a number of shows about WebAssembly, and my takeaway from those shows is WebAssembly is transformative, but the applications in which WebAssembly is being used are thus far few and far between. Why is that?

[00:01:31] CH: So there's a number of things that I can see and that I’ve heard that is slowing the adoption right now. One of the big ones is debugging. It's a little bit too difficult right now to have a consistent debugging story across all the different programming languages that support WebAssembly. It's improving. There is active work being done to make that better, but it is still not the best experience.

Now, I know that Chrome and a bunch of the browsers are adding support for DWARF and various other things that make this a lot better. So I do think if you talk to me a year from now it'll be a pretty different story on that front. And the other thing is that everything you do in WebAssembly right now, it has to be kind of bespoke. There's no agreed upon convention. There's no React. There's no framework. There's no one popular thing that has come out of the woodwork to really make WebAssembly development as simple as you would want it to be. But once again, that is going to be changing I think pretty rapidly. There's plenty of work being done. I’m approaching it from the server side, but there are plenty of projects out there that are looking to tackle it from the client side as well.

[00:03:09] JM: Tell me about some of the work that you've done in the WebAssembly world.

[00:03:13] CH: Sure. So about two years ago I started a project. At the time it was called Task, which was a tasks as a service. That was kind of the tagline for that, and it was based on Docker and it was essentially functions as a service with a couple of bells and whistles on it. And I worked on that for maybe eight or ten months, and it was good. It was fine. It wasn't extraordinary. It wasn't all that interesting. So I took a step back and I asked myself what would make this project actually compelling and differentiate itself from the other things out there?

And I had heard of WebAssembly I think like most others in the context of web app development, and I didn't think a whole lot of it, but then people started talking about it in these contexts outside the browser; wasmtime, lucet, wasmer. These extra browser runtimes if you will, they started popping up. And it intrigued me. I thought it was really quite interesting. And so what I ended up doing was I took the core of that task project, which was a job scheduler, and I kind of tore it down to the studs, ripped out all the parts related to Docker and I started experimenting with WebAssembly. And at the time I knew nothing about it. I had never touched it before. I'm not a web app developer. I've been a server-side back-end developer for a long time now. And I started really slowly, like I had quite a bit of trouble getting all of the pieces to fit well together. The memory management and the ability to access the network and all of these kinds of things that are pretty straightforward when you're running in a Docker container that everybody else uses, all of a sudden you have to be very intentional about every single piece because of the way WebAssembly is designed.

And so piece by piece I started figuring out which types of capabilities I wanted this system to have and I started adding them. So with WebAssembly, it has this deny by default kind of mentality about it where you have to explicitly grant any kind of capability to the module. Anything that crosses that boundary outside into the host system needs to be granted explicitly. And so I started what pieces do backend developers need to build a standard REST API, right? And I determined, "Okay, a minimum viable version of this would be the ability to make HTTP requests, the ability to access a cache, the ability to access static files and the ability to access a database." Those were my four things that I wanted. And so I just started building that. I started building those four capabilities and I kept doing that in the context of this job scheduler that had come from the previous project. And so that became Hive. What was previously called Hive. And that was the first project in the suborbital family if you will.

And then from there these other projects have grown from where Hive started everything, and today there're five projects in total. The main one if you will is called Atmo, and all of them kind of build on that core job scheduler. That job scheduler is now called Reactr. It powers Atmo and the WebAssembly runtime at the heart of the Suborbital ecosystem if you will. And what it enables is three things. First, it allows you to package up a whole bunch of WebAssembly modules and load them all into this scheduler and have them available as kind of just a cluster of compute if you will that you can execute these modules whenever you need. The second is what I call the Runnable API, and that's those capabilities that I just talked about. So when you are writing code to run in Reactr or Atmo, it exposes these capabilities and it does so with libraries for the languages that it supports, which is currently Rust and Swift. And the third thing that it lets you do is essentially mesh among many instances of this job scheduler and share the execution among several machines or containers or whatever you're running this thing in. And all of those capabilities together are starting to form this pretty interesting development framework or platform depending on how you look at it. And the goal is to make it reasonable and actually exciting for a developer to build backend applications with WebAssembly in the real-world, not just demos or proof-of-concept, real-world production-ready applications.

[00:09:20] JM: So if I understand correctly, Atmo is a system for heterogeneous functions to be loaded into memory and ready to be executed. So like you can write functions in lots of different languages and you can execute them all in the same context because they're compiled to WebAssembly.

[00:09:47] CH: Yep, that's exactly right. So Atmo builds on top of Reactr and it provides what I call a batteries included uh WebAssembly server-side framework. And what this means is that you don't need to worry about anything related to the underlying web server. You don't need to care about binding to ports, about setting up your TLS, about any of the normal things that you would have to write boilerplate code for in a Go or Rails or a Java, Node.js app. And what it allows you to do is write what's called a directive. This is a concept that I don't think it's new, but it is something that I've been experimenting with in this Atmo project, which is declarative application development. And so you use this file, this directive file, to describe your entire application. You describe all of the end points and exactly how to handle those endpoints by composing your WebAssembly modules together. And this has become a really fun way to build REST APIs where traditionally you need to spend an hour, two hours, three hours just bootstrapping a new web server application. Like I'm a Go developer, and so I've written the standard HTTP server 100 times. You don't have to do any of that with Atmo. You write your functions. You build your directive and then you create what I call a bundle to then deploy your application.

And I've started showing this idea to people and I've been having people try it out and I've been using it myself for the last couple of months and I just find it really intuitive. I'm really enjoying just being able to add a new function, whether it's in Rust or Swift, and then add it to an execution chain of these other functions that are completely unaware of one another and then have it automatically handle API endpoints that I need for some project or another.

[00:12:19] JM: Tell me more about your overall vision for the bucket of tools that you're building in the Suborbital stack.

[00:12:29] CH: Sure. So the tagline for Suborbital is helping you build web services that are powerful but never complicated. That's kind of my guiding light right now. And this comes about from everything in server server-side development getting more complicated in the last five or ten years, right? Whereas running a monolithic rails app on a VM or an auto scaling group was the norm. Now it's microservices and a dozen different flavors of serverless. They all will get the job done. And depending on your scenario, depending on your use case, I'm sure one or more of them are perfectly appropriate for whatever you're building. But no matter what you choose, it's more complicated than it was back when monolithic rails apps were the norm.

And so I've been really trying to focus on simplicity for the developer. I'm going to be turning my sights on the operator soon, the ops person who has to run these applications, but for the last few months I've really been focused on the developer tooling and the developer experience related to building these server-side applications. WebAssembly in my mind will become almost invisible within a couple of years if I have my way. I really want it to just fade into the background and become an implementation detail for a system that allows you to just focus on your code.

Right now WebAssembly is very buzzworthy and everyone's getting excited by it and I'm really happy that that's the case, but I don't want it to be the focus going forward. I do want to use it to enable these better experiences for these backend developers where currently you need to care about so many different things. I want to ensure that you can open your terminal, run one or two commands and have a fully bootstrapped application ready for you to put in your business logic, bundle it and run it. I've got that process right now down to about a five-minute experience. I want to get that down to 30 seconds. You should be able to bootstrap, write a function, deploy it, as quick as that.

And the other side of the simplicity coin is making it hard to screw up. And what I mean by that is you don't want to introduce any kind of security performance or any kind of vulnerability or regression in those areas. And so what I want is for Atmo and the Suborbital set of tools to make it really hard to actually do something wrong in terms of security or performance. And that's why I've gravitated towards this declarative format for defining your application because you are defining the business logic. You have full control over what the application is doing, but you don't need to care about how that is executed for you. And that allows me, Atmo, Suborbital tools, to incorporate all the best practices. Ensure that everything is running the latest versions. That it's adhering to all of the patterns and things that you want to see in a web service these days, because I come from a security background, and that's really important to me. So developer experience and making it hard to screw up are really things that are important.

[00:16:51] JM: And just to emphasize here, what are we even trying to do with this WebAssembly flexibility? Is it just the flexibility of being able to run Rust and Python and whatever else in the browser?

[00:17:06] CH: So there's a number of things. The ability to use many languages is great, but on top of that, there is the inherent security principles of WebAssembly that was one of the things that drew me to it. The fact that it runs in a sandbox. The fact that it has the deny by default mindset. And then there's also the performance aspect. So WebAssembly modules can run it very near native speed and you can have these functions that start up extremely quickly, right? The cold start problem that traditional serverless has been plagued by is almost a non-issue here. And with some optimization it will be a non-issue. And so WebAssembly is a means to all of those ends of better performance, better security, and I hope better developer ergonomics.

And then when it comes to the web browser, that's not my focus. However, I absolutely do see a future where you can include WebAssembly modules for both your backend and your web app code all in one application bundle. And those modules could be intelligently executed either on the backend, on an edge network, or even downloaded to the user's web browser and executed right there on their device. And that's the thing that I think is extraordinarily cool about this technology, is that even though I'm focused on helping you build server side apps, somebody else is focused on helping you build web apps with it and one day those are going to coalesce, and I think that's when we're really going to see the dividends pay off with this technology.

[00:18:56] CH: So let's go into this in a little more detail. If I wanted to construct, say, a backend web service in your world using Atmo, what are the steps?

[00:19:09] CH: Sure. So there is a CLI tool for the Suborbital ecosystem called Subo, and that gives you – I like to call it a meta tool chain. It doesn't have a tool chain in its own, right? It's not a compiler, but what it does is acts as a facade over all of the tool chains that you actually would normally have to interact with if you were building WebAssembly from scratch. And so when you are building an Atmo application, you'll use this CLI tool to create a new project. It will set up everything for you. It will create your directive for you and it will allow you to create these Runnables. You can add as many of them as you want in whichever language you prefer and then Subo will build and bundle these WebAssembly modules for you.

So if I can, I'll give a very simple example. Say I wanted to fetch the metadata about all of my GitHub repos, formulate that into a report and then serve that report as an HTML page. I would create a set of Runnables to kind of encapsulate that chain of functionality. Communicate with the GitHub API. Format the you know the data the way that I want. Maybe apply it to an HTML template and then that would get returned to whoever was calling it.

And so I build each of those functions, probably only five or ten lines of code each, and then I go into the directive and I define an endpoint that strings those functions together. And what you're left with is no boilerplate code, just two or three very succinct pieces of business logic and this directive that describes the whole thing. And then Subo will automatically detect the language for each of your Runnables. It will build them each into a WebAssembly module and then it will take your directive, your Runnables and any static files that you want to include and it will package them into this bundle. And at the end of the day it's just a Zip file, but it is a Zip file that Atmo understands. And once you have this bundle, that is your entire application. You didn't produce a Docker image or like a binary executable. You produce this bundle that encapsulates your entire application. And then you give that bundle to Atmo and Atmo will start up. It will set up the entire environment, the web server. It will get you a TSL certificate. It will do all the things that a backend service just needs to do and then it will load your bundle and set up your application and then it's done.

What's really great about this is you can configure it at runtime. So what I mean by that is this isn't a static binary that you need to recompile to change the functionality of. These bundles can be swapped in and out while Atmo is running. You can roll forward and roll back your application. Because these web assembly modules can be loaded and unloaded from the job scheduler at will, it allows you to do plenty of really cool things that would otherwise be pretty difficult. And so you can actually be serving requests with a certain application version and then have a new – You've done a deploy, you have a new version of the application. It can be rolling out to Atmo while requests from the previous version are still being handled. And these kinds of things are possible because of how WebAssembly works.

And so what you end up with is this nice succinct, easy to understand project structure. You have a tool that will do all the hard work for you of wrangling the various compilers and shims and configuration settings for you. And what you end up with is a very small, easy to manage bundle that is just a file and is really easy to deploy.

[00:23:59] JM: And just to emphasize, why is this superior to other methods of deploying a web service?

[00:24:08] CH: Yeah. So everything has trade-offs, right? I think it's important to note that this isn't the silver bullet that fixes everything. I think it solves a number of problems. I think in terms of simplicity, it's much better than your average microservice setup where you have to build dozens or maybe even hundreds of Docker images. They all have to service discovery and they all have to be attached to a Kafka instance or they have to coordinate in some way. This allows you to have polyglot applications. It allows you to have a single deployable thing, and it allows you to abstract away most of the complexity to Atmo itself. So when you think of scaling this backend system, Atmo, it runs on a message bus, an internal one that you as the developer don't really need to care about. And when it does scale out and it creates replicas of itself, the execution of your endpoints becomes meshed. And all of this happens without you needing to explicitly configure anything and without hopefully in an ideal scenario without your DevOps team having to do much of anything at all. It will abstract away a lot of that complexity for you. And we always want to get to that place where you can write once, run everywhere, and people have been saying that for forever. But I think it gets us maybe a little bit closer.

Docker does a lot of magical things and I think it's an incredible technology and I don't think it's going away anytime soon, but it lives at the operating system level. And WebAssembly and Atmo is trying to do something similar for the application level where these bundles are operating system agnostic. They're platform agnostic. You can run the same bundle on Arm or x86 or wherever you need. And WebAssembly itself can run server, browser, edge, on a mobile device. They can run pretty much anywhere you need them to because these runtimes are so very portable. And at the end of the day I'm really hoping that it allows – That the developer will allow Atmo to just take on some of the complexity that they normally have to deal with themselves, and that may not necessarily be a direct consequence of using WebAssembly, but WebAssembly does enable us to simplify these frameworks, these different orchestration methods so that we can take on as much of the burden as possible.

[00:27:24] JM: Can you go a little bit deeper on the comparison between WebAssembly and Docker as a technology?

[00:27:32] CH: Yeah, absolutely. So I saw a really great analogy for how to describe Docker recently, and I want to repeat it. And I apologize to whoever posted this on Twitter, because I don't remember who you are. But they compared VMs and Docker to houses and apartments.

So whereas a VM is a house. It has its own plumbing, its own electrical, it's self-standing. Containers are more like an apartment building where there are many units that share some common things like the plumbing and the electrical. And this is a really great way to describe Docker in my opinion. And Docker, like I said, it operates at the operating system level. It lets you package all the dependencies. It lets you include whatever tools, whatever binaries, whatever things that you need to make your application work and then it gives you the interface to run those things easily and programmatically.

So while this is great and it's become extremely useful and it's been the basis for all sorts of things like Kubernetes, it is still the full operating system, right? It is still something that needs to start up. It needs to bootstrap itself. It needs to start processes. And there's always going to be some overhead. There's always going to be something that has to happen before your application even starts up, which is fine, because usually that startup is on the order of seconds or even milliseconds. But when you're in the hot path of serving a request or handling an event or something like that, you can't always afford those many seconds or milliseconds.

And so this is the kind of thing that serverless struggled with a bit at the beginning. It's gotten a lot better as of late. Things like Amazon Firecracker and other projects have shown advancements in all sorts of ways, but at the end of the day there's always overhead. And so what WebAssembly and Atmo and these WebAssembly based tools let you do is strip away all of that. There's no underlying operating system. There is no processes that need to start up. There's nothing you need to do first. When the WebAssembly runtime loads your module, that's all it is. It's just the bytecode of your code.

And so when you have something like Atmo handling the web service and all these kinds of things, it can start up these WebAssembly modules in like single-digit milliseconds or less in most cases. And that's because there's extremely little overhead. We're at the point now with WebAssembly where the overhead is determined by like the time of the language itself. Like C and Rust have very, very small runtimes. Something like Go has a bit of a larger one. And so we're down to that order of magnitude where we're needing to care about what the language is doing, and which is a much smaller surface area than the entire operating system or the entire container. And it's really pretty awesome when you are able to take 50 WebAssembly modules, have them at your disposal to execute at, like I said, single digit milliseconds notice and have that be done faster than a container could probably even start up.

So it's that kind of – I don't want to say it's an order of magnitude because serverless is already approaching this as well with other technologies that aren't WebAssembly, but it basically gives us that for free. We don't need to care about what's starting up, what's in the container, and it helps us with that problem of making it hard to screw up the performance of your application. You just write your code. It gets compiled to WebAssembly and the WebAssembly runs with little to no overhead.

[00:32:11] JM: How do you expect people to be deploying WebAssembly modules? Like obviously today the de facto standard is I've got a Kubernetes cluster. I'm deploying my microservices into containers onto my Kubernetes cluster. Voila. How do things change if we're deploying backend services on WebAssembly?

[00:32:37] CH: Yeah, it can go a number of different ways. It could either change a lot or it could change very little. With how Atmo currently works in the bundle, you can kind of think of the bundle as analogous to a Docker image, where you're likely going to it to a registry and you're likely going to tell Atmo, "Hey, go use this bundle at this location." And maybe the registry will let Atmo know when new versions are available for it to load and roll to. There's that version, which is where I am right now with the Atmo and the Suborbital tooling.

But one of the reasons I chose to go that route is because it's more familiar and because I think there will be less barrier of entry for people who are used to doing the Docker image way of things. But because that bundle contains many, many teeny tiny WebAssembly modules, like some of these things are hundreds of kilobytes or less, because they are so small, they could be loaded independently. So you could imagine you're not uploading an entirely new bundle. Maybe you're just uploading one single new WebAssembly module as part of your one tiny sliver of your application. And maybe Atmo could get notified of that and only reload that one single module. And there's things like that that could happen in the future that I just haven't gotten around to building yet that makes you see why this could have some pretty good potential and why it could change some of the expectations we have around deploying software to the cloud, whereas right now it's a fairly static process of you push a version. It gets built. It gets uploaded somewhere and then your cluster takes it and runs it. That's a pretty straightforward linear process. It could become a little bit more dynamic and it could almost be like individual handlers or individual pieces of your logic are being reloaded and upgraded in real-time. You could even roll back a single one of these functions instead of rolling back your whole application if you have a problem. And all of that is hard. It's not yet built. It doesn't exist, to my knowledge at least. And I think we could get to that point within a couple of years and it will let us change the way that we think about deploying these applications. It won't be any kind of arduous process. Even today with CI/CD, things can be pretty darn easy, but it could just get all that much easier and quicker and more reliable to roll forward and roll back your applications when you can have really fine-grained surgical control over exactly what's happening.

[00:35:47] JM: Are there any other changes to infrastructure that you anticipate WebAssembly causing? Things people might not think of or be expecting?

[00:35:59] CH: Yeah. Again, this is one of those things that could either change a lot or it could change not very much at all. And what I mean by that is in the same way that the way you deploy, the software could either be very similar to what you do now or it could be this radical dynamic minuscule type thing. The way that the software actually runs could also undergo any type of change on that spectrum. So you could look at a project like Krustlet from the Microsoft Team days labs. They are looking to embed the web assembly runtime right into like the Kubernetes ecosystem. They are putting the actual execution of those modules in a virtual kubelet and letting the Kubernetes API schedule work, right? And so that by itself looks pretty similar to what we have today. Again, those WebAssembly modules are a lot smaller and they're responsible for a lot less. And so you still get some of the benefits that I was talking about, but it is more familiar and it is more similar to what we do today. And so it's more accessible and it's a little bit easier to maybe shift your brain to thinking that way.

Again, one day, hasn't happened yet. Maybe I'll build it. Maybe somebody else will build it. We could get to a point where Kubernetes isn't running OCI or Containerd or whatever. It could actually be something more similar to Reactr, the job scheduler, where it is constantly loading and unloading dozens or hundreds of these small modules and the communication happening between them is asynchronous and abstracted from the developer and all those kinds of things that I was talking about. And it could be Kubernetes that is orchestrating that or it could be something else entirely. I don't know. So there's a whole spectrum of different ways that this could go, and I fully expect there will be paradigms that I haven't thought of and that hasn't been thought of yet that will kind of shift the way we think about this as well.

But I think the important thing to note is that like what we've been talking about here is all about the drastic ways in which WebAssembly could affect the way we build software. And the question is how fast do we approach that future? What does the tooling look like and how willing will people be to accept that there is a smaller unit of compute than the Docker container that they're used to.

[00:38:41] JM: What's your take on the cloud provider approaches to WebAssembly? It's kind of interesting watching Cloudflare and Fastly both pile into WebAssembly technology. You don't see it as much from the bigger cloud providers surprisingly.

[00:39:02] CH: Yeah. I think what Cloudflare and Fastly are doing is awesome. I think that WebAssembly on edge networks is one of the most compelling use cases for it honestly. And I think that they're approaching it in slightly different ways and the way that they're approaching it is slightly different from how I am and from how others are. And this is going to be one of the things we have to wait and see. Which of these methods, which of these paradigms is going to shake out as popular or as the winner? I'm sure there will be more than one winner, but they are pushing the envelope. Like they really are.

I think the fact the major cloud providers haven't really done anything in this space, there are a couple things, but no major pushes as far as I'm concerned. It really shows that there is an opportunity here for open source projects and smaller providers and startups to take advantage of this technology shift and provide serverless platforms that are differentiated from Lambda and Google Cloud functions and Azure Cloud functions. I think that what will really drive things is the general understanding that WebAssembly is not just for the browser. That's still something that a lot of people think. And when I tell them I'm working on server-side WebAssembly, they look at me funny and say, "What are you talking about?"

So assuming we can make people aware that this is something that is not only possible, but also really cool and really powerful and we can get them to see what these benefits are and what that future could look like, I think there's a really good shot at not necessarily disrupting, but really adding something new to the serverless ecosystem that doesn't exist today. And one of the really great concepts that I'm working on is this idea of deconstructing an application. And I touched on this earlier where the WebAssembly modules in your bundle, some of them could be running in your centralized cloud, like your VPC in U.S. East One. Some of them could be pushed out to an edge network to operate tens of milliseconds away from your user. And some of them could be downloaded into the user's browser or into a native application to run right on their device. And I would love for Atmo or the underlying framework to do this intelligently, right? I think there's a future in which the developer doesn't need to explicitly define what runs where. That's the beauty of a declarative format, in my opinion, is that Atmo can make decisions about where these modules should be running and it can optimize depending on what the module is doing, whether it should run in your centralized cloud, on the edge network or even on the user's device.

[00:42:23] JM: So let's put things in context. What are the ways in which WebAssembly is actually being used today? The applications I know are like compressing a file on Dropbox and maybe there's like some video game stuff. Still not a lot though, right? Is there anything?

[00:42:45] CH: Yeah. There's actually more than you might think, and I highly expect that this is – Because I'm actively part of this WebAssembly community, but I have actually seen them popping up all over the place. So I've got a couple that I really like. One is the internet archive is using WebAssembly as a flash emulator to preserve old flash animations and games. That's one that came up in the last couple of weeks or months that I thought was really neat. There's shopify who is currently working on the ability to run WebAssembly modules to customize your store, your Shopify store, in real-time. Making real-time decisions about purchases and discounts and all sorts of different aspects of your store that you as the store owner can write small snippets of assembly script and have that execute on shopify servers as decisions are being made, as your store is being rendered. And then there's other things .

I mean, I work for 1Password, and we use WebAssembly in our browser extension to analyze pages and figure out how to fill your credentials, your login information into the websites that you visit. And so there're plenty of different things popping up here and there. It's definitely not mainstream. Don't get me wrong. It's still up and coming for sure. But the performance gains that people are seeing, like you said, the video game applications, is not something that is a personal interest of mine, but I've seen several instances of like running Doom compiled into a WebAssembly module. And I believe Figma, the design program, is using WebAssembly to accelerate the canvas in their application. And so there are these niches that are popping up here and there and they're getting bigger. The niches are becoming less of a niche and more people are realizing the potential.

[00:45:06] JM: What are you focused on across your portfolio of projects today?

[00:45:11] CH: There are two things that I'm focused on right now. The first is expanding the capabilities available with the Runnable API. So I mentioned there were four things that I wanted to be able to accomplish, and three of those four are available. You can make network requests. You can access static files and you can access a cache. What's next is the ability to access a database. That's a significantly harder problem that I'm currently working through. Because of the nature of the WebAssembly sandbox, you have to be very careful from a security perspective to make sure that you're building something that can't be exploited. And there's also just the fact that there's such a wide variety of databases out there. I have to do my best to build something that is agnostic that can work with the different providers out there.

And the second thing is the developer tooling. So I mentioned the Subo tool and how it is kind of like a meta tool chain that abstracts the complexity of the individual language tool chains. It works well, but it's still early on in what it could be capable of. It's going to be the thing that you interact with the most when you're using Suborbital's ecosystem, and I want to make sure that that's the best experience that you can possibly have. I think that the developer experience is the utmost importance. I think it will be all the more difficult to convince people to change over to this new way of thinking, this new method of developing applications if the tooling is obtuse and hard to use. And so I've really been putting a lot of effort towards that.

[00:47:06] JM: What aspects of the future of WebAssembly have we not covered yet?

[00:47:14] CH: Yeah. There's a whole bunch of standards that are currently being developed within the WebAssembly community. And the biggest one or the one that gets talked about the most is WASI, which is the WebAssembly System Interface. And that will eventually become the standardized way of granting capabilities to a WebAssembly module. And I've been building these capabilities myself for Reactr and Atmo, but I do hope one day I can replace most of that custom functionality with WASI. It is progressing pretty well. There's a lot of work being put into it, but it's not fully ready yet. And so this is another thing similarly to debugging. Hopefully if you ask me in 12 or 18 months, there will be a totally different story and I'll just say everything is running through WASI and everything's great, and I hope that becomes the case. And there're other standards uh within the WebAssembly body like interface types and a bunch of different like module linking. These are things that will make working with raw WebAssembly easier and it's something that I'm excited about as the person building that part of the stack, but I don't know how much the average application developer is going to necessarily care about those things. They're all fairly low-level. Hopefully you would never have to deal with them on a day-to-day basis. They're underlying kind of roughly in the territory of system call, memory management arena, and hopefully they'll just be invisible when they become adopted. But it makes me excited as the person having to deal with that level of the stack.

And then on top of that, for the future adoption on both sides of the stack backend and frontend, I think there will be new frameworks. There will be new tooling and there will be new endeavors that will make WebAssembly more and more mainstream going forward so that next time you and I talk hopefully you won't be saying, “Oh, I only see it once in a while.” Hopefully you'll be telling me you see it everywhere.

[00:49:40] JM: Cool. Well, that sounds like a great place to close off. Thank you so much for coming to the show. It's been a real pleasure talking to you and congrats on your project success.

[00:49:48] CH: Thank you. I appreciate it.

[END]