EPISODE 1473

[INTRODUCTION]

[00:00:00] ANNOUNCER: In software engineering, telemetry is the data that is collected about your applications. Unlike logging, which is used in the development of apps to pinpoint errors and code flows, telemetry data includes all operational data, including logs, metrics, events, traces, usage and other analytical data.

Companies usually visualize this information to troubleshoot problems and understand patterns and opportunities and how their applications are used. The company New Relic is a modern observability platform built to optimize your entire software stack from one place. New Relic includes a telemetry data platform that acts as a single source of truth for telemetry data. Built on top of that are tools for full stack observability to visualize and troubleshoot your data in milliseconds. When a problem does occur, New Relic's applied intelligence will detect and understand the problem to help resolve it faster.

Nic Benders is New Relic's Chief Architect and GVP of Engineering. As part of the engineering team since the early days of the company, Benders has been involved with everything, from agent development, to Zookeeper deployments, and all of the pieces and products in between. In his role as Chief Architect, Benders looks after the long-term technical strategy behind New Relic's product and the experience of all the engineering teams who built it.

He joins the show to discuss the engineering behind New Relic.

[INTERVIEW]

[00:01:37] JM: Nic, welcome to the show.

[00:01:38] NB: Hi, Jeff.

[00:01:39] JM: You've been working at New Relic for some time. And you have helped architect the platform. And New Relic has been around for a long time. As the platform has evolved and it's taken on more and more users and higher and higher volumes of data, I'm sure you've had to re-architect the systems of data ingest, the agent of data collection and probably just about everything. I'd like to start taking a top-down approach talking about the agent. If I'm monitoring my infrastructure, I have an agent that's going to be sitting across the different places in my infrastructure and relaying that information to a central server, which is sitting on New Relic somewhere. Tell me how the agent has evolved over time.

[00:02:33] NB: Yeah. I like how you started the framing. When I started New Relic, we were a small company. We served just a couple of markets. The Ruby on Rails was really our home. And we've recently gotten into Java applications.

Our biggest customer then was not that big. Our world was very different, that as we've moved over the years, those decisions don't last. And so, we've had to of course rebuild and rebuild and rebuild many times.

A great example of this is we look at the Java agent. The way that the java agent works is the JVM has – It's actually the very named agent. It's got a little flag you can launch on the JVM that says -agent, and it says, "Hey, as I'm booting up and I'm loading your program, I want to give you the opportunity to rewrite the program as it loads. You pass every piece, every class that's loaded through this class loader, and you can inspect the bytecode and you can make modifications to it."

We said, "Great. This is like the heart of agent-based monitoring." It's what New Relic has done from the beginning. It's also what our founder, Lew, his first company, Wily, with the Introscope product, what it did, was to use that bytecode agent interface.

The problem with this is, of course, you can rewrite an application to do anything. And so any bugs that you introduce can have a huge impact. We had to be very conservative. We have to make sure that like all the software is exactly correct. That we're not introducing any mismatches in our system. This is time-consuming and slowed down our ability to evolve the platform. And it meant we certainly we couldn't give like the keys to instrumentation to our customers, which is really what they wanted. They wanted to be able to write their own instrumentation.

At some point we went back and said, "Let's rethink this problem. What are we really worried about?" We're really worried about ending up with invalid bytecode so let's extract out those signatures of the methods in Java as they're coming through and make sure that we're matching those using the JVM's capabilities itself.

And so, the way our instrumentation now works is we weave in replacement method implementations by looking at those method signatures and then being able to replace them. If you put a bug in there, it won't work.

And so, instead of like creating a potential surprise down the road that's going to create a production issue, we're able to detect you directly at instrumentation time, "Hey, this class isn't the same as the class you're trying to replace." And we just avoid it. And so, we're able to bring in that safety.

I think that that pattern of starting from, "Wow! This is amazing. We can do anything." into, "How do you make it possible for not just our own top engineers, or all of our engineers? But all the way out to our customers and everybody in the world, how can they add that functionality?" requires re-implementing not just around performance, but around safety.

[00:05:57] JM: Can you tell me about the language choice of the agent?

[00:06:02] NB: Yeah. Our application monitoring is easy, right? For each application, you want to have a native experience for that environment. For our Ruby agent, it's in Ruby. Our Java agent, it's in Java. .Net agent is in C#. And then we've also got a little bit of C code there. And each of our language agents operates in that model.

When we moved into the infrastructure monitoring, I actually worked on the first version of that system. And, I don't know. I guess this is audio. You can't see how gray this beard is. I wrote it in C, because C is a great language. And it featured memory leaks, and compilation and portability issues and all the things that we know and love C for. But it was able to access those low-level interfaces. And we could compile it for lots of targets.

We went back for the second version. They're like, "Nic, it's time to take you safely away from the keyboard. Let's get some professional developers in here." Wrote the second version of – The infrastructure agent was in Go. Because it has the same ability to access those native calls. And it has a great cross compiler tool chain. But we didn't have to worry about me introducing a memory leak or something like that as the code was going through. And so, that's been a big win for us. And so, our infrastructure agent and the ecosystem around it are Go-based.

[00:07:29] JM: If you have a really high volume server, like something that's just getting slammed with lots and lots of requests and has a high volume of activity, you might have so many events that you need to buffer those events and be smart about how you send those to the central server. Can you tell me about buffering and batching of events that get sent from the agent to the server?

[00:08:02] NB: Yeah. When we think about the role of an agent in an agent-based monitoring system, that Hippocratic oath is like step one. First, do no harm. No matter what kind of amazing information you're able to gather from a system, the most important thing is that you do not impact the runtime, like make the application you're monitoring fail.

As you said, sometimes you'll see these like really huge volumes, and those are on your most important apps. People have worked really hard to make them run well. It's not going to be very popular if you say, "Hey, I’m able to show you all the things you need to know and view into your software." All you have to do is destroy your system. That's kind of where we were 20 years ago with profilers that you could only run in test environments instead of doing in real-time.

Our agents use a mechanism called reservoir sampling. You can think of the basic flow of an agent is that bytecode rewriter has rewritten the customer application. In those rewrites, what we're doing is we're inserting some timing measurements or putting a, "Oh, what time did we start? And let's capture a little bit of the call context." And at the end, now we've got the time, and we want to report it somewhere. And we can't just report it back all the way over the internet to us. You don't want to really report it to anything that's going to involve a blocking call. We can't even go to disk with this.

What you do is we go to a memory structure with that. That memory structure then – It buffers it. But it's more than a buffer, because it's bounded. It's our reservoir. And it will say, "We're allowing 5,000 requests into this reservoir," as an example size that you might see for one of the reservoirs.

If I've got an application that only does 3,000 requests a minute, and we're sweeping the reservoir once a minute to send it back to New Relic, no problem. If I have an application that's doing 10,000 requests a minute and you were sweeping it once a minute, the naive approach is going to be you're going to fill up the reservoir in the first half the minute and then you're going to not get anything from the back half. That doesn't give you a good representative sample of what the actual performance of the application is.

Every minute, it retunes the sample rate. In that example, you would say, "In minute one, we got twice as much data as we can hold in our reservoir. So let's adjust our sample rate to be a 50% drop." Minute two, assuming that our throughput is similar, we should get just about the right amount. And it continuously tunes and learns as it goes to make sure that, instead of early filling, we are able to get a even distribution over time and do so cheaply, because we drop before you have to capture and filter whenever possible. Sometimes you guess wrong. You have to manage those samples inside the memory pool and things like that. But when the system is working right, it's tuning its own performance to keep that constant overhead no matter how high the throughput is.

Then we're able to take those events because we know they've been fairly sampled not based on inspection or time. And we actually send them back to the server with a little bit of extra tab on it that says, "Hey, here's an event. But I want you to know that in this category we dropped nine events for every one event we stored." Then when you go in to the back end and you're querying this you're displaying at the UI, we can say, "This is a sample size. We had one recorded event and nine omitted events. So draw it on a chart like it's 10 events." Most of our systems are able to extrapolate that. That's statistically fair. Works really great for most methods. Doesn't work for some other methods. And we have to put a warning up and tell people, ", you're looking at sample data here." But for the majority of cases, that allows us to offer that consistent performance in the agent, consistent network usage, and also good query performance while still being statistically relevant and safe, because we're doing a fair sample.

And we get way into like tail sampling and head sampling and other stuff later for some of these span trees and things like that. But that reservoir sample is like the heart of the New Relic approach to agents.

[00:12:29] JM: How do you handle a network partition if you can't send events for some period of time?

[00:12:39] NB: It depends on the agent. We have different approaches for different scenarios. You picture something like a mobile application. Mobile application, the network is extremely unreliable. I'm interacting with my phone, some app on my phone. And then when I close the application, it's attempting to send that data back to New Relic to report on, "Hey, here's a user, they had these many errors. They clicked on the thing you told us to report on." Whatever that activity is.

The problem is the network might have gone away. Or my phone might sweep out the application and say, "I need this memory. I have to run a very important streaming application or something. And I'm sorry, you get de-scheduled."

The mobile application expects high volatility. Sometimes days' worth of disconnection. It just stores locally like the events. It buffers them up. When it's able to connect, it sends it in. And our backend is expecting it's going to get some late data.

On the other end of the spectrum is a browser. In a browser, when you load up a page that is instrumented with New Relic, we are – As your DOM is being built, we've got a little bit of JavaScript at the start. We start our timing hooks. We watch the page load. We say, "Oh, we've got our first content full paint here. We've got interactive here. This is how long your resources took to load." And we need to get that data out of the browser session and to New Relic as soon as possible. Because you can click at a link at any point. It could be just a couple of seconds into that page load and the browser is going to tear it all down. The browser doesn't care what JavaScript executes after you've navigated away. There's a very short window. In those cases, we have no buffering at all. We're just like, "You gather the data and you shoot it one shot. If there's no network, you miss it all."

Our other agents generally fall in various categories where a lot of the agents that we've built will have a small buffer. Where if you think about that reservoir pool again, instead of attempting to save 10 minutes' worth of data and send 10 times the normal data if you're out for 10 minutes. What you want to do is actually just decrease the resolution of your data. You say, here's a one chunk of data that represents 10 minutes. Our infrastructure agents behave a little different. Everything is about the environment that it's in. Both kind of what are the resources that the agent has to work with? And also, what's the expectations for that type of a network.

[00:15:06] JM: Now let's start to talk about the server-side of things. Are you using an off-the-shelf database to store events. Or are you doing something custom?

[00:15:16] NB: It's almost a little bit of each. We've got a few different databases that make up New Relic. When you think of the big picture architecture of New Relic, you started correctly right. The agents aren't the first thing. The purpose of our tool is to give people visibility into their running applications and allow them to make useful decisions based on that.

I got started at New Relic, I was a customer. I was a customer of Lew's previous company, because I had production applications that were just driving me nuts. And I couldn't figure out why they crashed all the time. I couldn't figure out why they failed. And suddenly somebody came and they shown this flashlight on to the system and they said, "Hey, if you install this tool, you can look inside the application." And save that project. Save my sanity. It was fantastic.

And so, when I think about that flow, you start with a real thing has occurred out in the real world. It's then captured by the agent. It's sent first to our edge layer. Our edge layer is looking at what kind of a payload it is and what customer it's for and deciding where does it go. And so, this might go for some category of data to – Our backend is divided up into cells. And we'll go into – We need a cell of type A is containing this type of data. This other type of data that we get is going in a type B cell. This other one might be going to type C. We've got a few different systems that are all federated together to appear as a unified whole.

The heart of it when you get into that storage system is a database that we've written ourselves. Lew used to tell a story where he's going around to his technical advisors and people on board of directors. He's like, "I'm going to write my own telemetry database." They're like, "You're a crazy person. It's going to take you three years just to get that thing off the ground. It can't possibly pay off."

And it only took about nine months to really get that initial version. We were able to get it going. The tool set available to developers these days is super powerful. And we had some fantastic people working on that project. And what we wanted didn't really exist in the industry at the time. Because what we wanted was a database that could take those individual things that happen, the events, and ask it a question that you didn't know you wanted to ask.

And so, when I talk about like multiple backends, we've got a data type that is like a time series-oriented metric. How many times did somebody click on this link in this minute? That's one data type. That's our metrics data type. Then we've got blobs, essentially, like, "Oh, I have an error here. Here's the whole stack trace from this error." And so, it's got a lot of data, but it's a big thing that you might want to dig into to figure out exactly what went wrong.

And then we have this event data, and trace data, and log data. Each of those, they tend to be small, that they can contain a huge number of dimensions, and the cardinality on it can be fundamentally unlimited. I might record in my event, "Nic clicked on this thing, or ran this query." And you're going to want to record, "Well, what query did I run? What was my username? What's a Guid for my session I'm in?" Super high cardinality data.

Answering that in the past has required building – Like you'd see an analytics at Cube. You say, "Well, I want to ask questions about users. Let's compute the average for every user." I want to ask questions about geographies, "Let's build a dimension on geography." And this data set just grows and grows.

The simple answer turned out to be don't pre-compute it. Just write down each of those things. Nic ran this query. Here's the full text of the query he ran. Here's the Guid of his session. Then like if I run another query, like, "Here's Nic again. He has a totally different query. He's still part of the session." Just record each one of those things.

As you do so, you build up just a physically large file. And so, it needs a database that's able to do almost full table scans at maximum speed. You build that worst-case performance as your best case performance and just get really good at it. That allows our backends for those events, for logs, for traces, which are all built on New Relic's NRDB, that proprietary database that we built, in order to answer questions you didn't know you had.

I feel like I – Maybe I didn't answer your question directly there, though.

[00:20:21] JM: Well, can you shed more light on the architecture of that custom database? The NRDB?

[00:20:29] NB: Yeah, absolutely. The way that NRDB works has evolved over time. When we first started, we saw like network was at a premium. Like the best interfaces you could put on a server were like one gig network connections. Storage was medium cost. We did pure SSDs, high-performance storage. Network is kind of expensive. Compute was pretty reasonable.

We said a lot of the traditional systems you would see, Hadoop, or something like that, function by having a bunch of storage nodes that have your data on them. And then you have a job that needs to be run. Let's say, count the number of pages that Nic viewed. Then you perform this MapReduce operation. You go out you say, "Hey, storage nodes, send me all the files to this set of compute nodes. They're going to process through it and find out all the times that Nic was mentioned. And then you merge it again."

There's a few key aspects in that like MapReduce pattern that we want to touch on. One is the identification of where's the data I'm interested in. Then it's the shipping of the data and putting query against it. And then that funnel in. How you bring it all in at the end?

For NRDB, it's not quite a MapReduce system, but it follows a lot of those same patterns. To identify the data, we've created an index in the structure on an account ID and a type of data. I know, I'm looking for log data for my account. So, "Hey, NRDB, identify all of the files that contain log data for this account ID in this time range. I'm looking over the last week. Show me just the ones for the last week."

Now, because when we first created the system, we looked at the resources available. Our first step was data was always kept on disk. Instead of taking your data and shipping it to a processing node, what we did is we shipped our queries out to the data nodes. We said, "Hey, storage nodes, run this query for me." And that query would be the like look at all the files that have been identified. Just rip them as fast as possible. Look for every instance of Nic. And then count it up."

Now, we've got a bunch of separate counts on the storage workers, which they're hundreds or thousands of storage nodes that might be answering a single query. Now we have to turn those individual partial results into a final result that you can give to the user. They send it up in a hierarchy. You merge it first to like one set of – Our internal parlance, we call the routers, because they route the queries. But it's just like an aggregation step. And you go to another aggregation step for final aggregation. And then it goes to the user. You can see easily how to do this with a count. You just take all your numbers and you add them again. Average is the same thing, right? Average is the total divided by the number of items.

Instead of computing an average on those storage nodes, what you actually do is you compute the total and you compute the number of items and you send both of those up. Those get added up. And then at the final step before it goes to users, you do the division and you turn that partial result into a final result.

This is also possible for more complex data structures. If you draw a histogram of data in New Relic, it's doing the same thing. We can't actually track every single request and send it up at each level. It'd just be too much data. We use a mathematical structures called Sketches, which take advantage of some mathematical properties of the data. And you build a sketch, you say, "This is a good representation of the amount of responses in each section of a histogram." But it's a mergeable structure. When you get to that intermediate level, you can merge it, and you can merge it again at the final level and actually give the histograms.

We can do the same thing for unique counts. All kinds of operations that at first glance you might say, "I don't know how you could possibly store an intermediate form." You can store it. That's the fundamental structure, was query comes in. Identify what are all the places that have the data. Send it to the machines that have that data. They perform the initial computation. Then we merge. And we may merge multiple times. It could be three levels of merging. It could be one. You get that merging up. And then you hit the result.

Over the years, as we moved to public cloud, we got a lot better network, honestly. We now have every machine that's got at least 10 gigs of network in it, a lot of them have much more than 10 gigs. We started looking at the hot region in the data and saying, "Maybe we don't actually need to store all of the data on disk all the time. Let's create a tiered system."

Now, when you ask your question, it still goes out that same pattern. But when you're getting to the machine that's going to perform the computation, it's going to have some data that's commonly requested. But other data, it might not. Other data might just be out in the object store. And so, it has to call out to the object store. Fetch the object onto disk and then perform the pattern.

That shift, as we went to public cloud into having that super high-speed network in between our workers and the object store, allowed us to change the scale of the system and change the economics radically, which you turned out to be very helpful to the business when we're able to use that to drive our pricing change that happened a couple years ago.

[00:26:27] JM: do you do any kind of periodic archiving of data to save on costs? Maybe moving it to a lower tier storage?

[00:26:39] NB: Yeah, it happens naturally in the product. You can think of the NRDB as having a super-hot area, which is in-memory partial results caching. If I ask it a question and then I ask it the same question a few minutes later, it doesn't need to go and recompute most of the week. Most of the week hasn't changed in between five minutes ago and now. Only the last five minutes. Most of the week, it'll have an answer ready to go. The most recent five minutes will be recomputed. That's the hottest part of the system.

The next tier down is data that is in the page cache on the workers. That data exists on disk in some kind of SSD. But it's also in page cache. When we perform I/O on it, we have to do the math. But we don't have to do physical I/O. We're just doing I/O against memory. Tear down from that is every worker has more physical storage than it has page cash. It's able to fetch that larger set. And then we're aging out of that based on – As people are requesting things in and out, just the same way the page cache manages its storage, the worker manages what data is on disk versus what is out in the object store.

[00:27:57] JM: To what extent do you leverage the cloud versus – Do you use colos? Or manage your own data centers? You're on AWS or something, right?

[00:28:07] NB: We have both. When we started our work, the public cloud wasn't really where it needed to be, especially for high-performance apps. But we always knew that we didn't want to be in the hosting business. We want to be in the observability business. We kept a close eye on public cloud over the years, and we started a little bit of toe in the water back probably five years ago. Three years ago, we started a major migration. We are in AWS both in the United States and for Europe. And we also have data centers with a number of different data center vendors in the US and with IBM in Europe.

[00:28:50] JM: Are there any complexities that come with that multi-footprint architecture there?

[00:28:58] NB: Yes. All of our engineers would rather that we had exactly one provider, exactly one region that handled all traffic everywhere. That would be the most efficient from a design perspective. It's not something we're ever going to have. We will always have our feet in multiple areas, whether it's data center and cloud, or multiple clouds, multiple geographies. I don't think that you can run a successful SaaS company today without having a global presence and having this kind of multiple backend presence.

And when I talk to other companies, it's very rare I'm able to find one who is like all-in on single provider, single region, single technology stack. Either they've evolved over time or they're branching out. Everyone is always in a period of transition.

And we look at our customers as a great example of this. A lot of customers who were all in on a single cloud provider five years ago are now multi-cloud. Some customers who were all-in on data center are now in the period of migration. And some who are all-in on cloud now also have some unusual data center components.

I should have mentioned, actually, when we talked about public cloud, I describe that architecture as being agents, edge, then our cells. And the backend is our data center where our finance and user management things happen. Edge is actually a different public cloud provider then cells. We are perpetually in that state of working with multiple systems. There is no single vendor solution that I believe can solve everybody's needs for everything.

[00:30:48] JM: We've talked about the agent. We've talked about the data platform. I think one missing piece of the story is the actual retrieval of data. If you take a dashboard, you've got APM and log management to retrieve, can you talk about the querying stack and how that's changed over time?

[00:31:22] NB: Yeah. When you think about the system, like we turn the whole thing upside-down. And so, now I'm not an agent sending my data in. I'm looking from the other end. I'm a user. I go load a page in New Relic and I want an answer to my question. There's a bunch of different elements that go into that.

The first thing, of course, is drawing the page itself. That page is stitched together from a lot of different data sources. In ancient times, when I started, it was all a Ruby on Rails application. It did all the fetching in the backend. It built it all into a big HTML page and pushed that final HTML out. That was great, except it led to what we call like the click and wait user experience, where anytime you wanted to view any other thing, you click, you wait for the page to build. It also had limited interactivity.

Over the last decade, we've introduced a lot of interactive elements and JavaScript, just like everybody else in the web. Our application is no longer a web application being run on the server-side. It's really a hybrid application with JavaScript in the very frontend running in the browser. And then kind of a smart mid-tier that's stitching it together.

The first version of that was pretty naive. And we said, "It's all JavaScript. And we just call APIs. And we build it up." When you first load the page, the first thing you do is you draw like a blank rectangle and you load four megabytes of JavaScript and execute it. Well, that's pretty slow. We don't do that anymore either.

Now, what we want is a hybrid, where you first load the page. It's able to put together the basics of the page on the backend. We know you've come in with some kind of user authentication. That user authentication goes through our API gateway, where we turn your authentication tokens into a principle on the backend. We say, "Oh, this isn't just some random Base64 nonsense. This is Nic Bender's." Nic Benders has access to this set of accounts. He has this set of features on his user, and all of that. That gets attached onto the request as it flows through the system. API gateway has figured out who I am. It has attached simple to use information decorating on to the request. The request goes down to the next step.

The next step is the thing that's going to draw the UI. It now knows who I am. It can say, "Oh, here is what the UI should look like for Nic." And we're able to populate, "Here's, the list of accounts. Here is what his UI version looks like. There's its time zone." That type of stuff can come in that initial response back so that the window is drawing the regions and drawing the right things.

Then inside that are going to be charts. Each chart is fetching its own data. The charts then, they go back through that same API gateway path again. They're fetching their data using GraphQL. And that GraphQL is that merging point of some of that federation. It is where we get maybe a piece of inventory data. We said, "I want to know how many applications I'm running. And what the names of them are. Or I want to know what is the operating system version on all of these servers." That data is stored generally inside of Cassandra or some system like that that's well-optimized for it. But it's all hidden behind that GraphQL facade.

If the chart I'm drawing, instead, is a telemetry data chart, it's going to go into NRDB. It goes into NRDB. NRDB then starts that fan out. Everything is represented inside of New Relic in the form of an NRQL query. You think it's similar to SQL. It's a text-oriented query language that can represent any telemetry query for New Relic data.

As a user, users can access that directly to get the full power of the platform. We try to wrap it up in chart builders and UIs and things like that so you don't have to become an NRQL expert in order to do it. But it's the language our internal developers use to define their charts and every page experience that uses telemetry data. NRQL query comes into piece of software that is a multi-cell query gateway. Its job is to start answering that question of, "Where is my data?" It might say, "Oh, your data is in public cloud in this region, in this instance like of a New Relic cell." Okay. Maybe it's in two different places. My query then gets split and I go to both.

I'll go to those cells where I ask the NRDB inside that cell, "Hey, I'm serving part of a query for this user. Where's my data?" Then it goes and identifies, "Oh, your data is in these files. We use a consistent hashing approach to map the files on to workers." You need to build up this whole list of files that might contain your data and then go take it – Like, we're going to fan it out to the workers. That fans-out then to each of the workers that have the data. They're the ones who are looking at the data, whether it's in page cache, on disk, or whether they have to bring it in from an object store. Pull it all together. And then the whole thing starts marching backwards, right? That partial data goes back to the NRDB inside your cell. Partial data again. Then it finally goes back to that initial point, our query gateway. It gets merged. Then that response is then being stitched together in GraphQL with the other GraphQL components. And finally delivered back to the calling JavaScript application that is going to be able to use that data and draw a pretty picture, or make a table, or light a set of traffic lights on the page. Whatever that functionality is about.

At every step from a user request down, what we've attempted to do is make use of fast access to high-frequency data. Things like who am I? That needs to be part of every single request. And then break it up into separate requests as it fans down so that the UI is not attempting to make a thousand separate calls. But it's taking advantage of parallelism at every layer.

[00:38:01] JM: As we begin to wrap up, maybe you can tell me, what's the hardest engineering problem you've worked on in your 12 years at New Relic?

[00:38:10] NB: The hardest things in engineering tend to be pretty well-known. I spent a lot of time as Chief Architect looking at the system of people. Because the system of people built the system of software. We talk about Conway's law a lot around justification for microservices. That any – Like, an organizational structure is going to be reflected in your final product. It really is a law. There's no skirting it. It will always be true. So you can fight it or you can embrace it. Try to embrace it where possible and build an organizational structure that will create the software and the product structure that we want.

Even though this isn't really like a technical problem per se, it requires a deep technical understanding. Because you have to be able to look at a bunch of different software systems and see what is the way to carve a boundary between them. And that's not something that you can do strictly from management experience or product experience. It's something that you need a long-term view of evolving software and a broad view of how software systems work together.

Because of that, at New Relic we've drawn our architects into that organizational design and product design a lot more than I think is done at other organizations. We make a very technical product for a very technical user. The people who can really say, "This is the API surface here," isn't just a technical decision. It's a technical decision that we're going to expose straight to our users.

And again, a product decision, therefore, of, "Here's the future we want to offer," becomes a technical decision because we're a platform. Having that tight relationship between what is it we're trying to build? What's the organizational structure we want to achieve? And then what is the underpinning engineering work behind it requires like that collaboration, and is what really determines, I believe, the long-term success of an organization.

For me, that's been the hardest problem, is looking at how do people work together? How do we apply the same rules that we use when designing software systems on to human systems? And a little bit of vice-versa. I think that that's the real art of engineering at a growth company, is not necessarily like writing the best loop. Although, we have some super high-throughput systems. We deal with billions of things every minute. So you have to be a good developer. But also having that broader view. And I'm a pretty questionable developer, I'll be honest. I've gravitated into those more strategic and longer view problems in architecture.

[00:41:09] JM: Awesome. Well, Nic, thank you so much for coming on the show. It's been a real pleasure.

[00:41:13] NB: Thanks it's been great to chat with you. It's a topic I love.

[END]