EPISODE 06

[INTERVIEW]

[00:00:00] JM: Nick Schrock, you are an engineer who was working at Facebook for a very long time. You are now working on your own company. Welcome back to Software Engineering Daily.

[00:00:08] NS: Thank you.

[00:00:09] JM: When you joined Facebook, the application code was not in great shape. Facebook had grown very rapidly. They were not many unit tests. The code had anti-patterns throughout. Tell me about the state of Facebook's software monolith when you joined the company.

[00:00:26] NS: Yeah. So I'd like to explain this story through metrics. So I believe I was engineer 180. The company was about five-years-old. We were just on the cusp of getting 2 million active daily users, part of me active month at least. Then the number of unit tests in the code base, I'll give you one guess, it was zero. Correct. It was zero.

What I would say is that I was shocked at some of the state of the engineering at Facebook, and then amazed. Maybe amazed and appalled simultaneously. The things I was amazed at was the bias towards velocity. I never worked in an environment like that, and when I was told you're going to be expected to commit code on day one," and that code chipsets production every single Tuesday, and I didn't even process it, because I had never worked in an environment like that. That's extraordinarily liberating, and the company was forced to build infrastructure around that.

But then when you hopped into a particular file in the PHP code base, I would lose the healthy color of my skin and looked like a scandalized ghoul of some sort. So it was a really interesting time. It was actually – It was all these ironic things, where it was a testament to the overall quality of the engineers that they were able to be productive and not much of a hostile environment. But I always like to give the analogy that every engineering organization needs to decide how much time they're going to spend cutting down the trees and how much time they're

going to spend sharpening their saws, and it was clear at that juncture, and the company realized that there was an imbalance there.

[00:02:07] JM: What was it that made it so that Facebook knew it had to shift from building new things really rapidly to doing some cleanup? Was there some acute set of problems that were actually causing Facebook some issues?

[00:02:22] NS: Yeah, I think we were just pushing a lot of bugs to production. They were starting to hire more experienced engineers, and the feedback those engineers would give to the powers at be was very strong, blunt and consistent. Then I think all the engineers just knew intuitively that there was something wrong, and the leadership to their credit understood this as well.

[00:02:46] JM: How did you prioritize what needed to be fixed?

[00:02:50] NS: The tools that Facebook typically used in terms of prioritization was a process called headcount. Meaning, it's actually very simple. You decide what company priorities are and then you set goals in terms of staffing. So the way that a company communicates its priorities is who gets assigned to what project. So the company effectively, both in terms of bottoms up, because the engineers in those days had a lot of latitude to work on they want, work on what they wanted. But also the top-down – In a top-down way, the engineering leadership gave space for a lot of really talented engineers to work on the core infrastructure.

[00:03:31] JM: Now, that top-down versus bottom-up management, you did hear a lot about Facebook being a fairly flat organization. What was the management strategy in the earlier days? Was it just kind of people figured out what needed to be done and it sort of spontaneously happened, or was there management that needed to occur?

[00:03:55] NS: So there's definitely management. It was definitely flat. The team I joined out of boot camp, which is our process for spending up inside the company, it was called Comapps, and this is a collection of what I like to call probably the 18 least manageable engineers in the entire company that were responsible for 80% to 90% of the surface area of the product. So that one team was responsible for managing messages, chat, events, groups, the profile. Effectively,

the entire site except for the newsfeed and except for the kind of account sign-up flows that the growth team worked.

I mean, now if you count that, I don't know how many engineers today cover that same surface area product, but like hundreds, and managing that group was a challenge. But, meaning, if it's bottoms up and flat, it doesn't mean that there's no management. It just means that the managers need to approach their job differently and need to guide and persuade their engineers rather than command them.

So the management was certainly able to direct the work, and then Zuck implicitly did have a effectively dictatorial control over the product and was very influential, and he could set priorities and then the engineers would have to respond.

[00:05:18] JM: So you were referring earlier to some metrics-based ways of describing how the infrastructure was starting to bump up against issues. The number of users and relative to the number of unit tests was infinite. So it was clear that there had been a pattern of people putting band-aids on things and not fixing the fundamental problems because there were such scale and there so much to build and so many new ideas.

Was there a problem with the engineering culture or was there an adjustment to the engineering culture that had to be made in addition to recognizing that, “Okay, we need to switch into fixing mode.” Were there cultural elements that needed to be fixed?

[00:06:05] NS: Honestly, I think there is just the collective knowledge and an acknowledgment that it was a serious problem. To be clear, I don’t know if the company made the wrong decisions in terms of – I think they went too far in terms of not building infrastructure I think with just establishing a little best practice early on. But I think that most companies bias towards building too much infrastructure.

I like to say that infrastructure is a problem you have to earn, because if you build infrastructure for a product that no one uses, what's the point? I think engineers naturally want to build too much infrastructure, myself included. In my previous companies, I had built all these infrastructure for products that didn't achieve product markets fit. So what was the point?

[00:06:47] JM: Meaning internally, like internally you built tools or you built –

[00:06:51] NS: I build kind of the tools that the other engineers use to build the product, but we didn't know what we were building. So the infrastructure has to match the product that you're building. So unless you have super high confidence you product market fit, it's actually good to bias towards fast iteration and working in the domain of your business rather than the domain of your infrastructure.

[00:07:13] JM: It's funny, when I talked to Brendan Burns, who made Kubernetes. Before he made Kubernetes, he actually built internal search infrastructure that nobody used. He built it. He spent a ton of time crafting it and architecting it, and then nobody used it. Then he took those lessons and built Kubernetes. It sounds like you had similar lessons, because within Facebook you did build a ton of tools that people actually use. You built frameworks that actually had earned their worth in existence.

Can you give an overview of some of the frameworks and tools that you built at Facebook?

[00:07:49] NS: Yeah. So the bulk of my career was spent building the abstraction stack that ended up being externalized as GraphQL, which is now a relatively successful growing open source technology. So that journey started. I was assigned a project that all the other – And I wasn't senior at the time. Meaning that the existing engineers who probably had the capability of actually doing it avoided the project, which is probably a warning sign I should have heeded. So I really gotten over my head, and that project was trying to unify our infrastructure between our concept of pages, which are public. Meaning, a public user's product, an entity that has a follow graph and not a friend graph, and our traditional accounts, because we wanted the site to behave more like Twitter, where everyone was on the same account abstraction. It became relatively clear early on that that was a software abstraction problem, because migrating everyone was not feasible and there is huge investment in both products.

So that kind of started me on this journey of trying to unify those concepts in software, which effectively led to the realization that we needed a coherent business object model for our site, which hitherto had not existed. So that ended up being reified or made real in a framework

called ent, short for entity. So this is kind of the base object model of the site now. So there's an end user object and an end event object, etc., etc.

Then on top of that was stacked a system called end query, which was effectively a query language you could express with a PHP DSL. It was mostly modeled on the Link, which is the.net framework abstraction, but it was modeled on Link spiritually. Link is super powerful and beautiful because it has explicit language support for it, and ent query did not have that.

Stacked on top of that was a system called FQL ent. FQL used to be the primary way that developers would access the Facebook platform, meaning our external users. A SQL-esque DSL over our object graph with the higher level abstraction that ent query provided, we're able to express our FQL tables in terms of a very, very thin wrapper on top of the ent query framework. So that I was able to replace a massive amount of platform-specific infrastructure.

Then on top of that was our API to serve our internal developers, meaning companies that worked within Facebook, and that's what GraphQL was. So that was kind of the final materialization of that entire journey, because we had built this internal abstraction stack that really fit the needs of our product. Then GraphQL was kind of that PHP API, but then expressed as a query language that you could express as a string and send over a network.

[00:10:48] JM: Now, you had a lot of success at Facebook compelling people to move in the direction of these abstractions that you were suggesting to them. Was there any lesson around convincing engineers to move in a particular direction that you learned early on?

[00:11:09] NS: I don't know if I learned it early on. I had instincts and then I was able to backwards engineer kind of a coherent or semi-coherent philosophy around this. So I call this evolutionary means for revolutionary ends, the art of changing large software systems in place. Now, that's a mouthful, but I am a firm believer in incremental process. But incrementalism doesn't mean heavy incremental goals. So the goal was to transform and revolutionize the way that product engineers at Facebook worked. But the key thing was to chop up the work into incremental milestones and objectives such that it was actually feasible.

This interoperates with reality on a number of dimensions. One is that you actually don't know what the end state is until you're closer to the end state. Two, you need to bring your users along the way, because if you jump directly from zero to N, it's going to be too much for them to absorb. So you need to bring your users along the way, incorporate them into the process, because if you do incremental chunks, they end up participating in the infrastructure, which is key, and then you can also be flexible in terms of aligning your journey while still holding that vision in your head to the organizational needs at the time.

I think this approach is critical, because programmers naturally are often ideological creatures, and they want to proceed. They believe that a certain way is correct and they want to proceed from point A to point B in a direct line and get to the final answer. If they're unable to persuade people to do this, or they run in some technical barrier, but the non-technical ones are the most fraught actually, because it's extraordinarily frustrating when you believe you have the right answer and other people don't believe you.

That's when you hear engineers complain about politics. There're a couple kind of politics in companies. One is the really bad kind when someone is locally optimizing and not maximizing for the global optimum. That's about politics. The good politics is the process of deliberation about what's the right path forward. Unless you have a totalitarian company, there's going to be discussions and compromise, and sometimes that means you don't always get what you want, and that's very frustrating. So you need to be able to navigate that process, and that interacts with the technical process.

So, incremental process, but without compromising on your eventual long-term goals, and part of that incremental process is also picking your right clients. So I would always say you start with a single client. You deeply engage with them. Hopefully there's a pre-existing trust relationship. If there is pre-existing trust and you're on the ground with them, they're willing to tolerate mistakes, deal with changing APIs. Through that process, you kind of start to dial in on the fact that you might be building the right thing. Then you might expand to three clients, ones that are overlapping, but not totally overlapping, because you don't want to over fit to one use case. You want to kind of capture a minimal N-clients that represent, say, like 80% of the use cases in the company. Then only once you're really confident that you have product market fit, then you really go aloud.

Then I used to tell the people I work with just like, “You should think of this like a political campaign, but with extremely informed constituents that have a lot of skin in the game.” What I mean by that is just like in a political campaign, you need messaging. So it's not just about having the right technologies. Having the right messaging that feels authentic and aligned with that technology. In any sort of non-trivially large organization, you will have to repeat that over and over and over and over and over again until you are so sick of yourself saying the same thing. But it's just absolutely critical to be able to have message discipline and understand what everyone needs, and understand that people, when you say something, hear different things. That can often be a fraught process to navigate.

[00:15:38] JM: And what's so important about it is at a company like Facebook where you have some degree of engineer autonomy and some degree of bottoms up process, you really have to make your case to people, because it's not just like you can introduce some tool and force people to use it. They're going to adapt it if they like it and there's probably no competing ideas in the company. You're going to really have to make a strong case. So can we make this concrete with some example of a tool or some fix that you made within the company?

[00:16:14] NS: So let me think about that. I think, broadly, at Facebook, the internal dynamic, and this is where you have to understand the dynamics of the business, is that feed was the center of the universe both in terms of the business, meaning that's where the company drove all its engagement and eventually made all its revenue, but also in the technical dynamics of the product. Meaning that every single team in the company effectively needs to integrate with feed.

So if especially in the data fetching layer, if you were able to convert or have the momentum to convert feed, naturally, the abstraction would spread organically across the company. I used to say as feed goes, so goes the app. Meaning that that technology which is deployed to feed that is generalizable, obviously there's some feed specific infrastructure. That will end up leaning through the rest of the organization.

Yeah. So I think that's kind of the generalizable answer, and I think the other thing was just maintaining relationships. Understanding who were the – I mean, I hate this term now, because

it's been polluted, especially post-fire festival, but the influencers within the company. Not Instagram models, but the engineers who actually did stuff and who could in turn convince other people to do stuff.

So I attempted always, especially when I was more junior and didn't have the credibility myself to affect change, maintaining very good relationships with the senior engineers, getting feedback early in the process, responding to their feedback so that they felt they had some skin in these abstractions. Then they advocated for it. Then once you get a sufficient corpus of kind of influential engineers who both just do stuff on the ground, but also have a voice, then things end up kind of unrolling as they will.

[00:18:15] JM: How does one become an influencer engineer?

[00:18:18] NS: We had this internal tool later in the company that tracked – Had a graph of relationships between people in terms of code reviews. So every time you submitted a PR, in today's parlance, what we call the dif, you would say, “Hey, I want these four people to review it,” and naturally the people who were the most influential had the most kind of inbound edges in that graph, and it actually did really match the perception within the company of who were the strongest engineers.

This is kind of frowned upon a little bit in today's world, but I still believe that engineering quality and productivity is a power law, the mythical 10X engineer. Well, I don't think it's mythical. I think there are 10X engineers, and those 10X engineers naturally become influential.

[00:19:07] JM: And is that the case whether or not they are good vocal communicators?

[00:19:13] NS: I considered multiplicative. Meaning that the really impactful people, at least if you’re doing infrastructure, are those who can both do work on the ground and therefore intuitively understand the needs of the developers. But then you can think of this like a multiplier effect. If you're able to communicate that effectively as well, that kind of multiplies your impact.

So there are so very influential people who aren’t are as communicative, but often they did it through other means. Instead of going and doing a lot of schmoozing and public speaking, they

would write documents, and documents can be extraordinarily higher leverage as well. So you need some form of communication capability, but people on the ground are – We used to say code wins arguments, and that was a culture we try to maintain, and that serves us well.

[00:20:04] JM: Facebook had a platform that was totally unprecedented in software architecture is my sense, and some of Facebook's weaknesses eventually became strength. So it was the first high-scale PHP application. So it was forced to build HHVM. It's one of the first products to really deal with mobile infrastructure and performance at scale. So that led to GraphQL, arguably, React Native.

When did you realize that the problem set at Facebook was unprecedented and that you were going to need to slaughter some sacred cows, build some completely new abstractions that might fly in the face of conventional computer science?

[00:20:53] NS: I mean, I don't think I ever realized it. We're just working trying to solve these problems. I never like – I didn't even realized at the time how that anything was particularly innovative was happening, because we didn't really have – It was a fairly cloistered world. It's kind of actually only after the fact that I realized that a lot, the PHP code base at Facebook between, say, 2008 and 2013, had a innovations that are only now just spreading across the industry, which is kind of bizarre if you think about it.

So now one of our favorite tweets ever. I forget who tweeted it, was in like every React presentation for the first two years of life of that project was someone in a critical way tweeted, "Facebook: Rethinking best practices," and it might have been even worded more aggressively than that. It was not an approving tweet, but we were like, "Yup! That's exactly what we're doing." Because we maintain – It's called like the beginners mentality, where you try to strip away previous orthodoxy and try to approach problems from first principles. Then I think that combined with some of the unique properties of the problems that Facebook was trying to solve kind of, plus the personalities and capabilities of the engineers in your organization was really kind of a magical mix.

Now that a lot of us have kind of sailed off into the sunset or have worked at other companies, only after the fact did we really realize that that was a special time and a special group of

engineers, but at the time we were just solving our problems and doing our jobs and just very focused.

[00:22:38] JM: In trying to think about the archaeology of the Facebook infrastructure versus the Google infrastructure, this is may be like overly contrived or like telling a story in my head. But one thing that stands out to me is that Facebook is like this huge opportunity that just grew and grew and grew and there were emergent patterns in the software architecture that, as you're saying, only in retrospect look beautiful and important and innovative. Whereas Google is like this – They do all these planning in advance and they come to this beautiful abstraction that is really well thought out and informed by traditional computer science principles.

Do you have any perspective on what the difference is in Facebook software development strategy versus Google's?

[00:23:31] NS: So I think there's a couple dimensions here. A lot of this is just downstream from the personalities and background of the founders. So Zuck didn't graduate from Harvard. Was very into the kind of hacker mentality of often compromise solutions, but in favor of velocity, whereas [inaudible 00:23:55] with PhD's, and then the nature of the applications they ended up building – I mean, Google is a text box, and then a list, but backed by extraordinarily sophisticated infrastructure.

Facebook is almost the complete – If Google is the ultimately verticalized app with a tiny sliver of surface area that is user-facing, Facebook is like the opposite of that. Meaning that an incredibly wide surface area with tons of concepts that are interrelated in complex ways both in the product and in the infrastructure, and then the technology ends up matching both the nature of the product, which is in some ways a personification of the founder's personalities and the staff who implements that, which again is also downstream of the founders and the early team.

So they are just two very different companies, especially in the early days. Kind of all – While the tech companies end up like kind of have – As they grow and have broader, they staff their companies from a broader subset of the population. They end up somewhat converging culture-wise. As they expand into other domains, that often overlap. But I would say that's kind of the primary – The reason I would say.

[00:25:21] JM: The tool that your best known for is GraphQL, and GraphQL grew out of issues with mobile infrastructure and data fetching, and we've covered it in detail in previous shows at a technical level. But to come back to what you said about the evolutions and revolutions and change within a software organization, you built GraphQL, but you also evangelized it effectively internally. Can you describe the process for your evangelism of GraphQL?

[00:25:51] NS: Well, It it was one of those tools that evangelized itself in a way because of a couple reasons. So one is there is the origin story of GraphQL. So this is a great story. So we had transitioned – We had tried to build the mobile app on top of HTML5. This is called Faceweb. It's notoriously one of the biggest clusters to redact language in Facebook history. Other events have superseded this, but market at one point said that this was the biggest mistake in the company's history. So the decision was made to do a full pivot to mobile and rebuild the apps using more traditional native technologies.

Now, the issue was is that we had hired a lot of outside expertise who were incredibly talented mobile developers, but didn't have deep grounding in the Facebook infrastructure and actually they were prototyping this thing and building it over FQL, which serves our external developers. So they were building is over a three-year-old feed. It was missing all sorts of feed type. It just like wasn't – They weren't going to succeed doing that.

One of my friends who had been kind of one of my – Both a close friend and also early consumer of almost every technology that I participated in. A guy name Bo Hartshorn. He joined that team and realized very quickly what was going on and he got started stopping by my desk every week and be like, "Schrock," he would say this more colorful language, "but we are in trouble here, and if you are someone doesn't engage in fix as data fetching thing, like this isn't going to work."

I was working on some – Something that wasn't nearly as important, but I'm stubborn. So he kept on doing that. Maybe the fourth time I was like, "Oh, this is important." When you're doing a massive technology shift like that, you're kind of shaking up the snow globe with every crisis as an opportunity. While the snowflakes and this snow globe are settling, you can kind of change the landscape a bit.

So this is both a critical juncture in the business, because we are about to IPO. We are about to be accountable to shareholders every three months. So we couldn't just like stop development on things. As a private company, we needed to start making money on mobile. We needed to start having a good mobile experience. I don't know if you remember, but like the mobile experience was awful. It was so bad. This is a type of shift in technologies that kills companies. Seemingly, unassailable empires get consigned to the grave on the stuff, like Yahoo is an example.

So it was totally obviously mission-critical. So just kind of the timing was right for this type of thing, and was spoken at this, and then I've seen some things that Dan was prototyping, Dan Schafer, one of the GraphQL your co-creators. Then kind of inspiration hit and built this thing. Now, we were working very close to the iOS team, but what really kind of made this the easiest thing to sell that I've ever dealt with was when we ended up building a tool, which is now called GraphIQL, GraphIQL. We love our puns. This was a tool – Yeah, I was talking before about messaging. The easiest way to have messaging that's consistent is to somehow embed it in the tools that the engineers interact with.

Literally, I could go those same influencers that I mentioned, the engineers, not the fire festival people. You could go to them and slap this thing in front of them and be like, "Start typing," and then things would fly up and they'd be able to construct query very quickly and they're like, "You want that?" They're like, "This is incredible."

So having a quantum leap in tooling that's visual, that's instinctual, makes it very straightforward, and then to double on to that, the first thing we built this for was feed. So that natural dynamic of as fee goes, so goes the app also kicked it, which made this – Actually, our problem was that technology spread too quickly. After we launched the iOS app, Zuck started exerting pressure to get GraphQL to be driving our profile like the second-biggest surface area in the site. So then you're in the stage where Mark is just kind of like walking by and being like, "When is profile going to be done?" Then we call this the Eye of Sauron where you don't like it when it's on you, but then you miss it when it's gone.

So that was actually one that I wish – I retrospect, I actually wish we could've clawed back the adoption and done it in a more modest pace and broadens more mobile expertise, especially in Android, which ended up being kind of our most fraught GraphQL offshoot and being able to go. So kind of lesson learned there in terms of mistakes that we and I have made in my career. So, yeah, embedding something in the tooling to make it blatantly obvious what the value is is very, very effective.

[00:31:15] JM: How did the evangelism strategy changed as you went from this project being internal to open source.

[00:31:23] NS: Well, once it went open source, Lee became the architect of the project, partially because he came up with the launch in the evangelism strategy, which, to be honest, I can be a little more upfront about this now, I guess, is that I didn't think it was going to work. So I was a GraphQL manager at the time and I was trying to convince Lee, "Oh! We should open source this thing," because we were kind of an open source momentum. React had been open source great success, and we were trying to open source Relay, which you can just think of, take GraphQL and React and smash it together, and that's Relay. In order to open source Relay, we'd have to open source GraphQL. So I called this my Byronic fantasy, because his name is Lee Byron.

He's tough to convince of things and then he came back and he's like, "This is what we're going to do. We're going to write a spec, and then we're going to write an unproductionized reference implementation of this in JavaScript." I was like, "You're out of your mind," and they show me some of the ideas and I was like, "You're completely changing the whole system." It took me a while to come along with the changes to the system and then being one of those brilliant redesigns. The number of very elegant subtle decisions that Lee made were was extraordinary. It's an extraordinary piece of work.

But I was still skeptical about this spec. My position was like no one's going to read it, let alone implement it, but I was wrong, very wrong. Within six months, it was implemented in most major programming languages. Then, in fact, people were already trying to implement it before we release the spec. Actually, one very prominent member of the GraphQL community, Mikhail

Novikov, he was explained to me how they had actually implemented a server of it before we release the spec by sniffing the network traffic from their iOS devices. I was like, “Okay.”

So that changed the evangelism, for sure, but by the process of the spec did not just improve the technology, but also improved the messaging, which made it we could come out, we had the spec we really strive to make it written in plain English. It’s one of the only specs that I know. I mean, I'm biased, but that where you can go and look it up and read it, and it kind of like reads like a normal document rather than like an insane spec. You can read it and kind of graph the system, which is good. It kind of opens with a preamble of like, “Here’s the philosophy of the system.” That is still quoted today at large.

So I think the evangelism changed, now that I think about it, that it was more formal. We had to formally write everything down both the technology and the messaging, and then you have this surreal experience, where then as the technology grows, you see people on YouTube like [inaudible 00:34:13] your shtick. You’re like, “I wrote that in a document. Now it's being repeated by someone who is explaining almost better than I am, or I can, or we can.”

So I think the lesson I would take away from that is, one, believe in the external developer communities, because branding a GraphQL engine is hard, and people did it over and over again. So that's extraordinary. Then also having this formalized written out messaging is very high leverage, because if people end up reading it, absorbing it, and then repeating it to other people, you have this kind of viral spread of the ideas, which is just as important as technology. Not maybe as important, but is a very necessary complement to the technology.

[00:35:01] JM: Which brings me to the question of Facebook open source versus other open source solutions. So with GraphQL, and JSX, and React, and Facebook's suite of open source technologies, Facebook really changed the world of open source, and it really changed the – I mean, in the React world, the way that people use JavaScript, and JavaScript frameworks. Do you think Facebook open source was better from a technological standpoint than Angular, and was GraphQL better than something like Falcor, or was it a matter of the evangelism and the communication skills of Facebook that made the Facebook technologies win out?

[00:35:48] NS: So I'm inherently biased, but I do think it was the quality technology that won out. I think also part of it was that Facebook style of app is far more aligned with the majority of business apps in terms of their requirements than Google. Meaning that – Like I said, the core product of Google is a text box and a list backed by very sophisticated infrastructure, and obviously they spread in other domains, but that's the core culture of their company. Whereas Facebook is a big complicated app with lots of interrelated concepts and really complicated application ontology, etc., etc.

So the solutions that we built were far more geared towards that, which is much more like any other business apps that you see anywhere. Every business app I know has tons of different concepts, and like the UIs change all the time, and it's complicated. They have complicated business logic. They don't necessarily have complicated infrastructure problems. Now, Facebook solves complicated infrastructure problems as well, but we have the complicated business logic.

So I think I'm biased. I believe that those frameworks are qualitatively better. I might get I ntrouble for saying this, but I like to say Angular is when you take the worst ideas from Java and port them to JavaScript. Whereas React feels like a much more JavaScript native solution in my view. Now, JSX haters will probably disagree with that, etc., etc. But I believe that the solutions are qualitatively better, and they fit the domain.

Let’s take Falcor versus GraphQL. One of the big selling points of GraphQL, this also plays in what I was saying before is that you modulate your messaging to the current technical reality sometimes, because one of the big selling points of GraphQL is that you could coalesce all your interactions with the server into one request rather than having a chatty interaction, which can be devastating on mobile networks.

Now, there’s nothing inherently in the abstraction in REST or Falcor that using something lower in the stack like H32 could solve these issues to some degree, but it was a huge selling point, and something like Falcor was designed for Netflix, and Netflix app sends your streaming video over it. You are, therefore, definitionally on a network that you can successfully stream video on. So they were going to tolerate far more chatty interactions with their server. I think that was one of the reasons why GraphQL ended up having wider adaption.

[00:38:26] JM: We've talked about one strategic inflection point within Facebook that was perhaps the biggest strategic inflection point while you were there, and that's the mobile transition. Were there any other strategic inflection points within that company that you recall that affected you as an engineer?

[00:38:44] NS: Sure. A big one when I just joined, I'm blocking on the name of the project, but it was effectively response to twitter that made our feed much more real time. That required a feed view, right? I guess we kind of like had one of those every 12 months. It was kind of the nature of the beast. But that was big, but I mean the shift to mobile is by far the most decisive technology shift, at least from a product perspective. Therefore, I was building infrastructure for product engineers, so that's what we felt.

So I don't think anything is comparable, but that's only from my view, and I don't have global view of the organization. So someone else might disagree, but from my perspective, the mobileship was the biggest strategic shift that affected the technology stack.

[00:39:35] JM: Were there any other "best practices" within Facebook, or I guess in software engineering that you rethought when you were at Facebook?

[00:39:45] NS: Yes. So two come to mind, one is ORM's. Facebook did not use an ORM as traditionally defined. One good way to make me mad when internally someone will go like, "Oh yeah, ent! That's another ORM, and that caused me to start pounding tables and be like, "Its exactly the opposite of an ORM. There's relational store underneath."

But that one was driven almost by technological requirements. ORMs are famously the Vietnam of computer programming, which I firmly agree with.

[00:40:16] JM: You actually said that in our last interview also.

[00:40:18] NS: Yeah. No, I mean, I hate ORMs. So I think the more interesting story, because I don't think it was driven by Facebook's particular technical requirements, but it's certainly a

dogma in the industry as model view controller. Model view separation was a dogma in the industry, and this is an amazing story, actually.

So this is one of the battles that was fought in our PHP code base. So we used to strictly separate the stuff and we had XHP, which was our JSX precursor, which is in PHP. Then we had a thing called preparable, which was our way of orchestrating asynchronous fetching with our backends.

So what you'd always had to do is that you'd have to, in separate files, you would have to edit your data fetching logic in a preparable file and then go do XHP and then match that and change your render, and this is the way you did things.

All kind of the beard stroking senior types, including myself, thought this was just the way you did it. Then it was one engineer just – These two engineers are brilliant. One guy named Ben Mauer who had been an intern of Myth and Legend, who then joined the company full-time, and he was fresh. I think nine months in. Then another intern-intern, whose just there for the summer, a really talented young man named [inaudible 00:41:46]. They wrote this thing that was like, "Model view is done. We need a thing called preparable XHP, which kind of co-locates these things in one file." This is actually – This ends up being materialized externally eventually as Relay. Apollo client also has that concept, and this happened in like 2011 in our PHP code base.

In retrospect, this is the most obvious thing in the world, which is like by having model view separation, you are pretending that something is decoupled, and it's very coupled. It turns out in order to display a piece of data, you have to fetch it. Therefore, you create this unnecessarily decoupling. At that point, very young engineers were able to kind of – It's actually one of my proudest moments culturally, because you can see the discussion on the PR unfold and like someone would weigh in, "This is terrifying," blah-blah-blah.

Then Zach and [inaudible 00:42:49] would go talk to them and they'd be like, "They have some good points," and I'm very personally proud that I was one of the earliest ones to turn. You can see me turn against MVC right in that thread, and they were able to convince a preponderance of engineers enough to commit this. Then basically like a 22 and a 21-year-old were able to

fundamentally shift the structure of our entire product code base, and then that same insight that they had was moved into other domain and programming languages. It really is extraordinary when you think back on it.

[00:43:27] JM: What do you miss about Facebook's engineering culture?

[00:43:31] NS: Am I miss mostly my ex-colleagues. When you go through a lot of the – There are very intense, very stressful times where you're building technology really under the gun, and you kind of become a lighter weight version of war buddies where you kind of remember you're in the stuff together. I'm trying not to curse here. Especially kind of the subset of the engineering teams that I interact with were just a group of extraordinarily brilliant and action-oriented and pragmatic people, and that mix is just lovely to work with. So I really do miss a lot of my colleagues, is definitely the primary thing. I don't miss the micro-kitchens, and the fancy pants, benefits. I certainly don't miss the HR and performance review processes. But just the work with extraordinarily talented colleagues doing, well, at the time was definitely the best work of their lives, and mine. So that's what I miss.

[00:44:43] JM: Any larger organizational reflections on what made Facebook special and what other engineering organizations could learn from Facebook?

[00:44:53] NS: That's difficult for me to say, because I don't have wide-angle contexts, and a lot of the struggles that other people have in engineering organizations. Giving your senior technical talent space to determine their own objectives and work relatively independently, but with guidance from management. I'm not a no management person, but I think that Facebook was able to really strike a great balance there of having non-coercive, but effective management styles combined with engineers who could execute independently. But then we get very direct feedback about what they were doing right and wrong.

So the performance summary cycle that I was just complaining about, it did have a very valuable second-order effects of changing behavior, and it's a really good mind, like I got some very negative performance reviews, or aspects of performance reviews that criticized some of my methods and communication styles, but that was invaluable. So that was the way the management –

[00:45:58] JM: So you agree with them in retrospect, the criticisms.

[00:45:59] NS: Oh! 100%! I agree with them very – I mean, it took me a week or so to emotionally deal with it. But then, on reflection, I was like, “Yeah, that's right.” Not everyone, not every criticism. But I think the management teams also did a very effective job of collecting feedback from your peers and then synthesizing it into a coherent message, that, and the best managers were that in a language that you can hear.

So my most effective manager, who most aligned with my personality and was able to kind of alter my behavior the most, is Arturo Bejar, and he synthesized all the feedback into very simple things. He would just give me one sheet of paper and would say, “Here's where you exceed your expectations, here's where you met your expectations, and here's where you fell short. Here's what we’re going to work on the next six months. Here's how we’re going to measure it.” Very simple, very actionable. It allows you to focus on one thing. Because like if you just get an unfiltered feedback on every – It’s like, “Oh! He did this and that,” and like you need to synthesize it into a message that a person can actually act upon rather than just like getting five piece of feedback and being like, “Well, I guess I'm a terrible person, and that's the way the world works.

[00:47:20] JM: All right, just a few more questions. My sense is that people underrate Mark Zuckerberg as a leader. What is the most underrated characteristic of Mark Zuckerberg as a leader?

[00:47:32] NS: So what I was always struck well with Mark is that in intense situations he maintains great deal of calm. So you set up this organization where you set very ambitious goals, very high-power people, and that naturally engenders some degree of chaos. Then Facebook did controversial things and you get criticized for it in the media. Not usually the same fever pitch that today is occurring. But I was very involved in this kind of – One of the are many privacy crises, were we had to really change the fundamental way that privacy worked in the app in a very condensed period of time. The whole management team was kind of running around with their heads cut off and really kind of – I wouldn’t call hysteria, but it’s like, “Oh! We got to see the company,” blahb-blah-blah. Then Mark would just kind of enter these meetings

and be like, “We just have to sound satyrs.” Fix the issue and calm everyone down, and that was – And he just kind of like – He's steady like a rock. He is the same person that he’s been for a long time for good and ill. He’s grown, but like the core personality is there.

Yeah. So I just really appreciate the combination of not being a pushover and not being not ambitious, not having strong goals. With that in combination with like a steadiness and calmness is very, very effective.

[00:48:58] JM: You are now working on your own company, and you are taking reflections from your time at Facebook, your eight years at Facebook?

[00:49:09] NS: Eight years.

[00:49:09] JM: Eight years, and then time spent outside of Facebook reflecting on what is going on in the broader startup ecosystem and enterprise software ecosystem. Tell me about your reflections and what you're working on to the extent that you can talk about it.

[00:49:26] NS: Yeah. No, I can talk about it. So in terms of what I've seen externally in the startup ecosystem that I did not see while I was at Facebook is that I was totally blind to the extraordinary innovation and progress in the public cloud infrastructure as well as all the technologies surrounded it.

I love Facebook, and like I had heard of Docker. But I was like, “What the hell is Docker?”Then I started to play with it and I was like, “Oh, this is super interesting.” Then coupled with these new hosted SaaS services were like – Even simple stuff I was kind of amazed by. I was, “Wow! You can spin up your own CICD infrastructure where you are running a container? That's a beautiful – Like containers are what processes should have been the whole time.”

So I was blown away by the innovation, the pace of innovation. The fundamental underlying change of economics of the public cloud infrastructure, and all the associated cloud native technologies, Kubernetes, the entire ecosystem. So that was a total – That took me a few months to really absorb and learn about.

What I'm working on today, the company is called Elementl. So it's Elementl without the A, and I left and I didn't want to use social media anymore, and I started talking to people in more legacy industries; healthcare, finance, etc., and I was asking them what their technological problems were. This whole notion of data processing, some people call ETL, data cleaning, a data infrastructure problem just kept on cropping up again and again and again. So I was always end these meetings and I would say, "Wait. Wait. Wait. Wait. Wait. Wait. You're telling me that's what preventing you in your mind from transforming the American healthcare system is the ability to do a regularized computation on a CSV file," and they're like, "Yeah, pretty much." I was like, "What on earth is going on here? This is crazy."

So I have difficult resisting the urge to rathole on infrastructure, and I just dove right in. I've kind of discover this data engineering ETL world, and I think more and more programming going forward is going to be in this structure. I think more engineers are going to have to participate in this as like, "You need feedback loops that integrate with the apps to do ML." Then when you would talk to people in the industry, every data scientist ever, like half their talks start with, "Well, I spend 20%, 10%, 5% of my time doing what my job is supposed to be, and I spend the other time data cleaning."

It's like, "Well, first of all, if that's 90% of your job, that kind of is your job." But what it actually reminded me of was talking to, let's say, a frontend engineer in 2008 and they would say something like, "I spend 10% of my time on my business logic and 90% of my time fighting the browser." As a result, frontend engineering at that time had a lot of pathologies. It was consider this engineering backwater. There's like no culture testability. Engineers would like dive in, do just enough JavaScript to make something fly around and then get the hell out. People didn't want to work on it, because they would look at this code base and be like, "I don't want to touch that. Please."

Then eventually things changed. Yes, browsers improved, but I actually think – Again, I'm very biased, but React, in some ways, Angular, and these other frameworks, really did change the trajectory of the ecosystem fundamentally, and there is now testing frameworks, all the sophisticated infrastructure, and now you don't hear people say that. You'll hear people say, they're like, "Oh! There's JavaScript fatigue. There're too many tools," and there's not like kind

of a vertically integrated solution. But no one says they spend all their time fighting the browser or none of their time dealing with their business logic.

So I think that echoes to me in the data engineering space, because when a data scientist says I spend 90% of my time data cleaning. That's how they experience it, but that's not the root cause. It's not just like it's hard. It's a system-wide ecosystem why pathology, like testing is super difficult. These pipelines are – Or these data processing apps. I hate the word pipeline. These data processing apps are kind of multi-phase. They go from like a Spark job, to a data warehouse, to a Jupyter Notebook. Every time you hop from face-to-face, you fall off what I like to call semantic cliff, where you lose all context in the data. This is ends up getting reified as you hire this genius data scientist, pay them God knows how much money, and then their workflow is you hand them a Jupyter Notebook and a CSV file and they're expected to reconstruct the entire domain of your application from scratch. That's crazy.

So high-level, I believe there's a company to be built and a technology to be built that serves the same function at a very high-level that React played in the frontend ecosystem similar in the data engineering ecosystem. So kind of like high-level, what React did for UIs and what GraphQL did for APIs. I want Dagster, which is the open source technology that Elementl is hosting, to kind of be that, but for data processing, ETL, ELC, ML pipelines, or whatever you want to call it, and then to build a sustainable business on top of the that, that is a hundred percent aligned with the success of the open source technology.

Because I think constructing businesses that can sustainably host open source tech is a challenge, and I'm excited to also kind of move the needle on that or attempt to build an instance of that that works.

[00:55:35] JM: Okay. Well, we will save the rest of that can of worms for another conversation. Nick, thanks for coming on the show. It's been great talking to you.

[00:55:41] NS: It's been a fantastic conversation. Thank you very much for having me.

[END]