**EPISODE 1037**

[INTRODUCTION]

**[00:00:00] JM:** Facebook Messenger is a chat application that millions of people use every day to talk to each other. Over time, Messenger has grown to include group chats, video chats, animations, facial filters, stories and many more features. Messenger is a tool for utility as well as for entertainment. Messengers used on both mobile and desktop, but the size of the mobile application is particularly important. There are many users who are on devices that do not have much storage space.

As Messenger has accumulated features, the iOS codebase has grown larger and larger. Several generations of Facebook engineers have rotated through the company with responsibility of working on Facebook Messenger, and that has led to different ways of managing information within the same codebase. The iOS codebase had room for improvement and Project LightSpeed was a project within Facebook that had the goal of making Messenger on iOS much smaller.

Mohsen Agsen and is an engineer with Facebook and he joins the show to talk about the process of rewriting the Messenger app. This is a great deep dive into how to rewrite a mission-critical iOS application, and this team became very large at a certain point within Facebook. It's a great story and I hope you enjoy it as well.

[SPONSOR MESSAGE]

**[00:01:27] JM:** When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

Go to softwareengineeringdaily.com/g2i to learn more about what G2i has to offer. We've also done several shows with the people who run G2i, Gabe Greenberg, and the rest of his team. These are engineers who know about the React ecosystem, about the mobile ecosystem, about GraphQL, React Native. They know their stuff and they run a great organization.

In my personal experience, G2i has linked me up with experienced engineers that can fit my budget, and the G2i staff are friendly and easy to work with. They know how product development works. They can help you find the perfect engineer for your stack, and you can go to softwareengineeringdaily.com/g2i to learn more about G2i.

Thank you to G2i for being a great supporter of Software Engineering Daily both as listeners and also as people who have contributed code that have helped me out in my projects. So if you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[INTERVIEW]

**[00:03:17] JM:** Mohsen Agsen, welcome to Software Engineering Daily.

**[00:03:19] MA:** Good morning. Hello.

**[00:03:20] JM:** You worked on a rewrite of the Facebook Messenger app for iOS. Why was it so important to rewrite the entire application and reduce the size of the Messenger app?

**[00:03:32] MA:** I think in our case, it was kind of a matter of goal setting. We decided to go for a fairly ambitious goal and to basically become kind of the smallest messaging application in the world and preserve all the rich features of Messenger had. As we looked at our existing codebase, we realized that would be a very difficult thing to do in a codebase that had grown over five years, and we had actually spent about a year beforehand trying to make the existing product smaller and more modular and we were reasonably successful, but not to the extent that we wanted. At some point we decided to kind of go for the more ambitious goal, which is again rare in this industry these days of just a complete grounds-up rewrite all the core messaging features.

**[00:04:11] JM:** Doing a complete rewrite is generally not a great idea. It can be very risky.

**[00:04:17] MA:** Yes.

**[00:04:18] JM:** But sometimes it can be the right choice. How did you evaluate that option of doing a complete rewrite versus retaining a large portion of the codebase?

**[00:04:28] MA:** Yeah. I mean, I've been in the industry for a long time and have seen a lot of – I mean, I think it's probably well-known that most software rewrite fail. So it's not something you want to take lightly especially for these large complex projects. On the other hand successful rewrites can be truly groundbreaking for the company's that pull them off. Obviously, you might be familiar with the history of Microsoft Windows and the transition from Windows 9X to Windows NT codebase; or in Mac OS, the transition from Mac OS 9 to the next step and ultimately what became Mac OS X transition which ultimately give birth to iOS. We obviously do not take that decision lightly. It's a very difficult decision.

What we decided to do is actually kind of go through kind of a fairly lengthy prep process where instead of just replacing our existing product, for example, we built a new codebase and we tried it on something like the Apple Watch and we released an Apple Watch version of Messenger, which was well- received actually and did well.

We went to a number of steps and it took I would say almost about a year of decision-making and trial to make sure that we have made the right decision. We kept weighing both options. Can we just take the technology, the new technology, and put it inside the existing product or are we better off rewriting? Again, we basically released two products. We released a watch product and we released a standalone Messenger Lite iOS product. It wasn't released very broadly in some countries around the world. Those two in combination kind of gave us the signal that we needed and the confidence that we needed that this is something we can actually do, but it would be difficult and it would take a while, which turns out to be very true, but we kind of went for it and we did it.

**[00:05:57] JM:** Give me a brief history of how the size of the Messenger app has grown over time. I mean, when you set down to rewrite this thing there was –

**[00:06:09] MA:** Yeah, Jeffrey. This is a great question. I don't actually have the exact numbers. I recall a point where and we were, for sure, we had approached about 120 MB in size. Now, if you go look at most messaging apps, they're actually not that much smaller to be honest than Messenger, and Messenger was by far not the largest app.

We had approached the whole problem, actually, instead of thinking about others, we actually approached it, maybe this is where the word LightSpeed came from, the absolute limit. What is the smallest messaging application that you can build? Then the question became, "Would you actually be able to do it and make it as full-featured as Messenger?"

That's kind of – I think Messenger had peaked, like I mentioned, over time. It grew and features. It grew in users. The more users you have, the more features you have, and the more features you have, actually the larger the app and a lot of independent teams working, contributing to the codebase each bringing their own dependencies, their own framework, their own libraries, their own way of doing things that accumulated over time.

I think it's probably well-known that at this point we went from something like 110 to 120 MB product all the way down to a 30 MB product, which is, again, it's not the kind of thing we could've ever done if we did not do a ground-up rewrite.

**[00:07:19] JM:** As you're looking at this thing, Messenger was a gigantic app when you started this, and as you just alluded to, you had different generations of teams that had come in and done their work and brought in their own libraries and probably little sections or large sections of the code that nobody wanted to touch because it's just like –

**[00:07:41] MA:** It's going to run for a while.

**[00:07:41] JM:** We don't really know what that does. It's a black box. It's been papered over a thousand times. How do you get your head around the architecture like that? What was the information gathering process?

**[00:07:53] MA:** Oh, yeah. That's an amazing question. I think that for us, actually, we approached it, we almost – If you're familiar with the approach of bisecting kind of a diff or something, we approached it kind of that way where we took the two extremes. On the one extreme, we started ground-up with the basic messaging product. We started with kind of file a new project and said, "Okay, I just want to be able to send and receive text messages. Login to Facebook and send and receive text messages. How much work is that?" We use that is one extreme of the baseline.

On the other side, we did a very comprehensive partnership with the team where we literally – Again, it's a very difficult thing. Everybody takes it for granted. We're able to kind of amass very large amount of at least just feature definitions. What are the features that the different teams have contributed? That was number one. Second, a relative rank order of how often are these used? How key are they to kind of the user experience? Were there features that we had already been planning on deprecating? Is this a good opportunity for us to deprecate features? We basically, on one hand, we had a running piece of code, did basic messaging. On the other hand, we had giant spreadsheets that had all the features. Then we began – We built a baseline. We said, "Okay. Well, if it took us this much to do the messaging features, let's now look at the long tail of everything else," and kind of extrapolated. Obviously, it's not quite linear like that, but we generally extrapolated from that and started coming up with estimates. We partnered with a lot of the feature teams, which again a very bold decision on their part of saying, "Hey, if we rewrote this, would you be able to rewrite all these features?" It took a while on some negotiations and we set out doing it. Like I mentioned, the rest is history.

**[00:09:25] JM:** Okay. Describe the process for builds, and testing, and releasing Messenger, or at least, when you started on this rewrite process, what was the lifecycle of the Messenger product when it got built and how it got tested? How the release process proceeded? What's the day-to-day heartbeat of Messenger?

**[00:09:47] MA:** Yeah. Facebook is quite famous as a company that moves fast. I think we have a well-earned reputation that we tend to be very fast in both execution. We obviously do a substantial effort to maintain quality and we do a high volume of release. I think often you will see the products are updating every couple weeks in a mobile product that is releasing through

the apps or this updating every couple of weeks or so. Messenger, the way the Messenger codebase and most of Facebook codebases are developed is you can think of it as a form of trunk based development. We have a master branch. Actually, we live in a single repository that's well-known. We have a single repository that has all the codebases. LightSpeed and the existing Messenger were basically in the same repository. There were not in different branches. We're building them both at the same time.

The Messenger build, basically, there is a continuous integration system as developers land diffs as they check in code. There are nightly runs that run labs and do certain additional tests. There are QA passes that work on these builds. It gets the leased employee dog food. It stays in employee dog food for a while. We have a test flight branch that's just basically direct fork of, I believe, release candidate, that we do a release candidate, and that's the one almost all Facebook employees use. Master is kind of our branch that we use for development. Then it goes through release candidate, which goes all Facebook employees, and that's the one I believe that detours to test flight. Then that is the build that I believe every couple of weeks rolls into the Apple Store. We did actually accelerate that cycle I believe on the LightSpeed side, but I don't know if that kind of answers a question.

We have kind of an active trunk based development model, and one of the key challenges is we actually had to do this not with one iOS Codebase. We had to do it with two actually. We could not ignore obviously large deployed users on what we called at the time classic Messenger iOS, which is the product we replaced with the LightSpeed as well as the LightSpeed version.

**[00:11:35] JM:** I guess I'm a little confused here. So you had two versions at a certain point that you were maintaining. You were maintaining the old users?

**[00:11:44] MA:** Yeah.

**[00:11:45] JM:** Okay.

**[00:11:45] MA:** Yeah. I mean, we could not – Again, as you can imagine, Messenger is very feature-rich and was very highly deployed to hundreds and millions of users on iOS. We did a couple things. Like I mentioned, in the beginning we had two products. One was a light product

that we used for country testing and some parts around the world. The second one was actually our classic product and we're maintaining both builds.

At some point, we actually did start doing kind of these dual-boot tests where we actually put the LightSpeed codebase inside the classic codebase and we were able to kind of deploy LightSpeed and roll it out more gradually. Think of it as what we did, is we took features in classic and we replaced them with their LightSpeed versions on the inside and we used our kind of testing and experimentation infrastructure to allow us to get a signal on how good are these features working. We did that gradually for about a year, and that's why most people probably didn't notice this. When we actually finally went to LightSpeed, the size of the product was quite large, because what we had done is we had kept the old codebase, added all the LightSpeed features and it wasn't until I believe actually a week ago or so that we actually now removed all the classic stuff. Now when you go to the app store, if you install Messenger, you go to settings, storage, iPhone storage, you look at Messenger app size, you should probably see it somewhere in a 29 to 30 meg range.

**[00:12:57] JM:** As you were doing this rewrite – Actually, I guess more broadly, can you just tell me about some of the internal tools and technologies that Facebook has developed to make large mobile apps easier to work with? Obviously, I know about React Native, but I don't –

**[00:13:17] MA:** Yeah. Facebook is – And as someone – I've been in the industry for a long time. A lot of my tenure had been actually at Microsoft prior to Facebook. Facebook's development infrastructure I think is probably has a well-deserved reputation of being legendary. It is unbelievable what the company is able to do.

I think Facebook's approach is if there are things that are off-the-shelf that work, absolutely use them. But if things don't scale, a lot of infrastructure that Facebook generally develops, and actually I would say probably for the vast majority of it, contributes it back to open source. Those things like Mercurial source code control is something that we use internally that obviously open source and deployed well beyond Facebook. That is one of the pieces.

We have a product called Fabricator. Think of it as our GitHub kind of kind of pull request equivalent system, where you can do code reviews and you can go do code approvals. We

have, again, the build farms and the build machines and we also are the open source sponsor of the Buck project management. Think of that as our version of Bazel or CMake or something like that.

Facebook developer infra has a nontrivial amount of both standard. At some point, we use Xcode to code and debug. We use Visual Studio code in our editing and some of the native environments. We use Android Studio or ItelliJ on the Android side, but on the other hand we complement them with technologies that are really designed for the Facebook scale of thousands of active engineers lending, I kid you not, probably thousands of diffs daily into a single repo, repository.

Mercurial, Buck. On the website, we have all the expansions of the PHP programming language with Hack, with what we called HHVM, Fabricator for source code control and review, and I'm sure I'm missing many.

**[00:15:02] JM:** Right, and I guess a large swath of these do fit into the development of Messenger.

**[00:15:06] MA:** Yes, absolutely. Again, I think in the beginning, to be honest, when I first showed up at Facebook I'm like, "Wait. Why aren't we using more off-the-shelf?" and the more I use these tools, the more I recognized, "Okay. At some point, the scale and the velocity with which Facebook operates requires really specialized tools for build, for code review, for nightly tests, even systems. For example, we have a system that lets you subscribe to changes to certain files or directories so that if I am working on a key area, I want to make sure – I mean, we have an environment that's very open. Any developer at Facebook can change any part of the code. We tend to empower everyone to do that kind of engineering, but we have these systems that let me subscribe to these files to at least see the changes that are happening and allow me to maybe work with a team to either do them differently or review them to make sure that they are kind of well-done.

Again, it's an amazing scale and it's kind of a sight to behold when you finally wake up one day and you realize, "Oh my goodness! We are building some of the largest and most complex

mobile products the world has ever seen, and it feels like a well-oiled machine. It does work," which is pretty wild.

[SPONSOR MESSAGE]

**[00:16:19] JM:** Looking for a job is painful, and if you are in software and you have the skillset needed to get a job in technology, it can sometimes seem very strange that it takes so long to find a job that's a good fit for you.

Vettery is an online hiring marketplace to connect highly-qualified workers with top companies. Vettery keeps the quality of workers and companies on the platform high, because Vettery vets both workers and companies access is exclusive and you can apply to find a job through Vetter by going to vetter.com/sedaily. That's V-E-T-T-E-R-Y.com/sedaily.

Once you're accepted to Vettery, you have access to a modern hiring process. You can set preferences for location, experience level, salary requirements and other parameters so that you only get job opportunities that appeal to you.

No more of those recruiters sending you blind messages that say they are looking for a Java rockstar with 35 years of experience who's willing to relocate to Antarctica. We all know that there is a better way to find a job. So check out vettery.com/sedaily and get a $300 sign-up bonus if you accept a job through Vettery.

Vettery is changing the way people get hired and the way that people hire. So check outvettery.com/sedaily and get a $300 at bonus if you accept a job through Vettery. That's V-E-T-T-E-R-Y.com/sedaily.

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

[INTERVIEW CONTINUED]

**[00:18:08] JM:** You wrote this – Or I guess there was this article about project LightSpeed about rewriting Facebook Messenger and it outlines several strategies that we'll get into. Things like

using the mobile operating system more aggressively, using SQLite more aggressively, some other strategies. But just from a high-level, were there any particular features that were causing a lot of trouble or causing a large percentage of the bloat of the codebase? From a high-level, things that users could understand or people who have used Messenger. What are the areas that were causing the glut of the excess codebase?

**[00:18:48] MA:** Yeah, that's a great question. Again, that was part of the reason we decided to go for a rewrite instead of a remodel, and basically it turned out it wasn't anyone thing. If we take a quick step back and – Again, I used to jokingly say, "Look, Messenger is a two-screen app. You have an inbox and you have a thread, a chat conversation. How bad could it be?" Well, it turns out there's a lot more. There's audio and video calls. There's group calls. There is AR effects in group calls. There is the experience of – Again, we have an inbox unit for stories. We have an inbox unit for who's active. We have a friends tab that shows you who's online that lets you consume stories. We have an entire visual production system for stories in Messenger. We have photo editing tools built to their product so that it's very quick for you to edit a photo and send it. We have thread customization where you can change the color and the style of a given thread. We have nicknames for participants in a thread. We have group conversations where you can admin control. We have business features where you can interact with bots, or businesses. The business can control the menus in these threads. I can keep going on and on and on.

Every one of those are incredibly important aspects and we can't just rip them out. That was our conclusion. If we could just say, "Hey! Well, let's just rip 80% of the features out. After all, we can just do text and media messaging be done." It turned out that is not the right answer for us. We are a full-featured messaging and we want to stay as a full-featured messaging app. Then became kind of attempting what feels counterintuitive. Well, you want to be full-featured, but you want to be the smallest. How on earth would you do that? Our answer became, well, rewrite.

**[00:20:20] JM:** Okay. Well, let's start to dive into this. The first strategy for improving the size while maintaining the performance was to use the mobile operating system more aggressively. I would assume that when you're writing an iOS app, you're already using the operating system –

**[00:20:40] MA:** Great question. Yup.

**[00:20:41] JM:** In a pretty aggressive fashion. What do you mean by being able to use the operating system more aggressively?

**[00:20:47] MA:** Yeah. Actually, let me take a quick step out of even how – I think the way we approached the entire project, some of us are kind of fans of cars, was the way we would actually build a sports car or a racecar. Basically, kind of when you build a racecar, there's kind of two things you would do. One is kind of there are set of rules you have to follow. We can cheat and say, "Okay. Well, we'll compromise safety or we'll do something like that," and then you scrutinize everything. The rules for us were actually fairly clear. This has to be the full Messenger. There was no compromise. We're going to do the full messenger. It had to be full-featured future product.

The scrutiny is that's when we began asking ourselves questions, and one of the first question is we looked at our existing UI code and we looked at existing entire codebase of Messenger and we said, "Is there a better way to write this?" Messenger, like I mentioned, had grown organically over four or five years, and at some point Messenger had acquired a lot of features that Messenger ultimately took out.

You may or may not remember, we had like a games experience inside of Messenger that we took out long before project LightSpeed. We had a rooms' experience where we can try and do bulletin board style, large-scale room kind of experiences that we had tried. We had built a lot of infrastructure, user interface infrastructure, for those types of experiences. Those basically are abstractions you start putting on top of the operating system. You've probably seen a lot of frameworks.

For us, one of the first questions we asked was do we need to use frameworks or could be just write it as plain – The question we asked ourselves, "Could we just write it the way the Apple programmers guide say?" Write your view controller, write your view. Avoid writing custom views if you don't have custom draw. Going back to basics, that was number one.

Second is we started asking our assumptions. We had looked – Again, the operating systems are not static things. They evolve every year. You probably see Apple and Google do very

substantial advances in their operating systems. Large applications like Messenger is very hard for them to leverage the latest and greatest the operating system has to offer. Not necessarily because of a platform reach, because Android addresses that with Android X, and iOS obviously addressed that with very large update deployments of iOS.

We started asking ourselves, “You know, auto layout was not very performant on low-end devices a few years back. Is it still that nonperformant?” It turns out the answer is no. It's actually not bad. So we started using more and more of these things. So that became a new principal, which is look at the latest and greatest that Apple and Google have to offer and find a way to do it that reduces your need to incorporate a framework, to incorporate additional libraries, to add levels of abstraction. Basically, the use the OS was a euphemism for, “Okay, we know we need to write this code. Is there a better way to write it?” That's kind of how we approached it.

Examples like I mentioned. We use Storyboards in cases we felt it's okay to use iOS storyboards. If you're familiar with those, which is kind of a declarative UI where you can build view controllers and views without having to do any code. We used constraint layout in some cases. We used auto layout, auto guided layouts. In other cases, we use standard iOS view controllers. Again, took a lot of the abstractions out and just tried to get closer to the OS and remove any kind of abstraction between us and the OS.

**[00:23:51] JM:** Some parts of the rewrite were in C, right?

**[00:23:54] MA:**

**[00:23:54] JM:** Can you tell me about the parts which you rewrote in C?

**[00:23:57] MA:** Yes, and that was one of the big decisions before we even made the decision to do project LightSpeed and kind of start with iOS. We definitely had our eyes set on the full user base of Messenger, including Android. One of our realizations was – Which is kind of again, it goes back to looking at our existing codebase. Messenger is a database app. It’s very database-centric. It's not that hard to kind of imagine you have tables for your threads and your

messages and your participants and your attachments inside a message and your contents. Those five tables are kind of the heart and soul of an app like Messenger.

When we looked at those, we started looking at the logic that you run when you do things. When you send a message, what happens? Well, you put it in a database because you want to be able to run the UI off the database. You schedule something to try and send the message. Ultimately, it's going to go to the server. The server may actually produce a more complete version of the message. It may resolve it to do integrity checks and things like that for it.

When you send a message, the thread, the conversation you're in, you probably want to market is read clearly. You probably want to change this timestamp. We had all this database-y logic. When we started looking at our existing codebases, we're where finding a lot of that logic is sitting in a platform-dependent way. So that would be in Objective-C in iOS, and in Java on Android. The more we looked at it we said, "Wait. This is all basically very general-purpose logic. It does not have UI views of UI view controllers on iOS. It does not have views and fragments and activities on Android. It's basically pure data manipulation."

Our insight was what if we actually took that and we made the core part of the schema, the business logic and the sink logic of everything in Messenger be portable and cross-platform? That was the first question. We decided the answer is after studying enough of our codebases, the answer is yes, and then we asked ourselves, "What's the best way to do cross-platform?"

You can probably tell we are friends of SQLite, and SQLite is a very, very popular, very successful cross-platform library. It's one of the most highly deployed libraries in the world. It is in portable C code. We decided to kind of follow the footsteps of SQLite and we had one more reason for why we chose C on iOS that we thought would actually give us an advantage. I'm happy to kind of get into that. Had to do with core foundation if you're curious.

**[00:26:18] JM:** Actually, you mentioned that this idea that it's easy to think of Messenger as a database- centric app. I don't know if that's actually intuitive. I mean, for me, when I was reading about that, that was actually a little bit confusing to me. So when I think of Messenger, I think of like I load this thing and maybe it fetches some data from the server, and then it loads things into memory. It loads lists of –

**[00:26:42] MA:** Yup, conversations.  Yeah.

**[00:26:45] MA:** Like in-memory representations of what my friend list is, the conversations I have open. I don't imagine an embedded database sitting on my phone that is representing the objects of my Messenger app. I literally imagine just in-memory object representations.

**[00:27:04] MA:** Yeah, do a fetch from a server and then render a thread list. That is an awesome question. Yeah.

**[00:27:07] JM:** Yeah. That's what I imagine.

**[00:27:10] MA:** That is an awesome question, Jeffrey. Yeah, here's the deal with messaging though. Messaging is a transactional experience. People have seconds, right? You want to launch the app and you want to be able to – Imagine, again, you're sitting at your desk and you running late. You're getting up. You grab the phone. You unlock the phone. You launch Messenger. You want to type a message. You want to send it. You literally have seconds.

If you have to wait on network conditions to even get the conversationalist, to even resolve your contacts, to even do anything. One of the key attributes of messaging very different than say a feed product like Instagram or Facebook is that you actually truly try and avoid getting off the network. You can be dealing with – You don't want the user to wait. They don't have time to wait. You can await downloading a video on your feed. You know you're downloading a video from the Internet. There's nothing you would not want the phone in the background to go download that video for you.

One of the unique parts of messaging in general is very transactional, very high-volume, a very large number of small interactions. This is by the way not new. I would say my belief is actually almost every messaging product starts with an online experience. They just fetch messages from the server and they let you reply and they quickly – By the way, email is not the different as well. It doesn't take you long to realize, “No. No. No. No. One, I'm spending time re-fetching data that I can just keep. It doesn't change that much.” So you start keeping the data. Second, you say, “Wait. Do I really want to block the user as they're sending multiple messages while I

try and communicate over the network?" No. The answer is I want to defer. I want to batch things. I want to differ them. Third, the data that you actually are trying to render in the user interface over time grows and becomes more complex. You're not just trying to get a conversations list. You're also trying to get a list above it of users and who's active. You have nicknames. You have themes. You have profile pictures. You have group profile pictures.

You basically start moving gradually less from an online to actually – I don't want to use the word offline, but a more device-centric experience that only goes on the network to do the things it absolutely has to, which is deliver a message or give you a new message. That's the key of how a lot of messaging and actually email applications try and achieve speed. Unless they are HTML-based, which is that's a different kind of category of experiences.

Again, if you're familiar with something like Microsoft Outlook and Microsoft Exchange, you could almost raise that exact history back to almost those two products as well. I happen to have worked on both a lot many years ago. History kind of keeps repeating itself as well. You start with an online thing and then you realize, "Wait, the client is being very sensitive to every network hiccup, every server load hiccup, in cases where it doesn't need to. It's doing resolving things it should've already had. Why not keep them? Why not use them?" That's what leads you down the database path.

**[00:29:54] JM:** I guess it's worth talking about SQLite here. SQLite is a popular embedded database that people use for mobile applications in many cases. What does it mean to – I think we've done an entire show on SQLite. But for people who haven't listened to that show, why is SQLite useful for a mobile application like Messenger? I mean, why do we need a specific type of database like SQLite? Why not use – Or can you use Postgres, or Mongo on your client?

**[00:30:24] MA:** Oh, that's an awesome question as well. SQLite, again, it has a light term. It was built to be an embedded database. It was actually built exactly for that purpose. Postgres or SQLServer or MySQL or Mongo or any of those products, those are database servers. They are built from ground-up to support thousands, sometimes hundreds of thousands of concurrent users. The manage threads. They run their own memory pools. They manage connections and concurrency. They do all of these things. That is an enormous overhead for a mobile device.

Basically, that database is fundamentally single-user. It exists for one app. One user on that app using that experience. SQLite is unique and maybe you could say both fortunate. Again, both lucky and good, and that it was built for that purpose. It is small. It is lightweight. It is embedded. It does not take over your threads, your process, your memory. It lets you manage all of these things. It's there to give you access to persistent data and for you to be transactionally so that you have kind of correctness and you can do multiple operations on one transaction and they're all consistent. It also have a rich SQL query language where you can do these complex queries that you need to do to construct your user interface for your messages or your conversations.

It's very well-built for this purpose. It's also C code and very portable library. It's already widely deployed in iOS. It's widely deployed in Android. It is easy to deploy to the Mac. Actually, it's pretty deployed into Mac. It's easy to deploy to Windows. We also have messaging and messaging-related experiences in some of our other devices like Oculus headsets or portal videoconferencing devices. SQLite is a great asset in that regard. I would be surprised if most messaging apps don't use SQLite. We just chose to use it maybe a little bit more methodically is the best way I can describe that.

**[00:32:13] JM:** Does SQLite sit in memory or is it on disk? Does it just have just have different caching layers, like many database systems?

**[00:32:22] MA:** Yeah. SQLite, basically, the way you want to think of it is a set of C APIs. Use them to kind of open a database file and then you can execute SQL statements against that file and you can insert update delete or select from that file. It does have a built-in caching layer and it's actually quite good and it's smart about figuring out. Obviously, it has an index system that lets you create indices and figure out how you can fetch data very quickly. But it also has temporary tables and memory databases, and we actually do take advantage of temporary tables for things like typing indicators or online status, state that does not need to survive the application restart. Again, we use tables for everything. It's kind of our dominant data structure. SQLite is a very good fit for us. Again, it allowed us to avoid building dedicated data systems for every one of these experiences.

**[00:33:10] JM:** Did you build some kind of ORM system for accessing the database from the messenger client?

**[00:33:16] MA:** That's another good question. ORM is an object relational mapping system. SQLite, like most modern databases, is a relational database. It has tables and joins. It uses relational algebra at its heart for you to express, and it's dominated more by queries. Most teams indeed end of building a mapping layer between the relational system and some object hierarchy that they can then consume in Objective-C or Java.

We actually decided not to do that, Jeffrey. We decided to actually embrace the fact that it's relational and instead just make it easier for you in Objective-C or Java to consume those results sets and for you to issue these queries including these complex queries. We ended up producing kind of a stored procedure language on top of SQLite that lets you express these complex SQL-related business logic and made it easy for you to call that as an API, and that API returns a result set, and the results, basically think of it as a table, a memory table instead of an array of objects. The memory table has clean APIs both on the Java site on Android and the objective side in iOS for you to be able to access columns. We embrace the relationalness of SQLite in our product and we expose that all the way up, and actually avoided us creating more mapping layers. Again, more frameworks and more abstractions.

**[00:34:32] JM:** Just to spend little more time on the database side of things, the embedded database center things, tell me little bit about the schema. What are you modeling in the SQLite database?

**[00:34:42] MA:** Oh, that's an awesome question. I mean, again, this a topic I could probably – Let me say this. Messenger, like I said, is deceptively complex. We literally have hundreds of tables. That's hard to imagine in a product like this, but mostly because we decided to model the entire application as relational. We have what we call the core tables. Those are the things you would expect. I have my threads, which are my conversations. I have my messages, which is text messages. I have attachments, which is the media that kind of attaches to these bubbles as how you see a photo or a video. I have contacts, as you can imagine, and a conversation may have multiple participants. Maybe be one-on-one or a group and you have – So that's kind of the foundation. We call those the core tables.

Then we have a bunch of system tables. These are things that we used to be able to sink data. How far are you and your conversationalist? What new messages we need to fetch? Any mutation you do? If you change your profile picture on the device, how do we upload that profile picture up to the cloud? That sort of thing. We have kind of sink and system supporting tables.

The vast majority of the additional tables are all related to the features. Like I mentioned, we may have tables to track in-memory as you're typing in a thread to show you the typing indicators. We may have tables that show you the online status. The little green dot you see in Messenger against any contact anywhere you see them. You can now imagine the join that we'd do between contacts and presence to show you that green dot everywhere we show your profile picture. Again, when you render names in a thread, we have to do a join because there may be nicknames and you can actually give people nicknames in a thread. We have nicknamed tables.

When we load the thread, we have to load the participant list. We have to load any thread customization you might've done. You might have picked a custom emoji for sending in a thread and so on and so on, the last set of stickers you used, your favorite stickers. The actual sticker packs that you have subscribed to. Metadata about sticker packs in the sticker store. I can keep going on and on. All of these become tables. For us, they all get integrated into a – Like I said, it's a scheme that we have divided into what we call regions, and we have a core tables region. We have a system reason. We have these regions that are partitioned across the feature keeps.

**[00:36:49] JM:** When you load something like a friend list or an active conversation or something like that, I assume you put some kind of in-memory representation into memory, and then an update to that in-memory representation, there must be some transaction that occurs to propagate that to the database layer, right?

**[00:37:11] MA:** Yes. It's actually an awesome – Again, another meta-design question. We decided to build LightSpeed from the ground-up with what you would consider a reactive design pattern.

**[00:37:20] JM:** Interesting.

**[00:37:21] MA:** In our case, what we mean reactive design pattern is you don't do two-way data binding. You don't mutate data. What you do is basically you read data. You have an old representation of data. You get an indication that data has changed somehow. In our case, again, it can change in a lot of different ways. We'll talk about that in a minute. Then you get the new version. You diff them, and then you produce a diff and you render the UI changes based on the diff between old data and new data.

In our world, the way mutation happens is I don't want to call it out-of-band, but we almost want to treat it out-of-band. You mutate by calling functions, by calling store procedures that mutate the data, but you're not the only one mutating the data. Remember, the server could be coming in and mutating data for you. That's how you receive a new message. You didn't do anything on the client. The server deployed data to the client, ran a particular store procedure, which inserted that message into your messages table. The user interface gets the notification. There's new data. It gets the new data diffs it against the old data and renders an up-to-date message list. But that's no different than if you were in the composer in that very same thread and you had send. It was you calling that sort of procedure that mutated the data, but you didn't know. Anything could've called that store procedure.

We're very big on the reactive design pattern. There's many other terms word. We call it kind of a of the database-centric maybe view of programming or a database-first design pattern or a single source of truth. You hear a lot about kind of the single source of truth and these reactive design patterns especially with the emergence of even more modern, not just React and React Native, but also SwiftUI and Combine on the iOS side, and Jetpack on the Android side. It's actually not that different than those worlds.

**[00:38:58] JM:** Totally. The single source of truth in this case is the backend database.

**[00:39:03] MA:** Ah! Actually, the single source of truth as far as the user interface is concerned is the client database.

**[00:39:08] JM:** Got it. The SQLite database.

**[00:39:10] MA:** Yeah, the SQLite database. The reason for that is a lot of the things – Remember, when I mentioned earlier, we're trying to not get on the network all the time. We do a lot of things optimistically in the local database and then we sync those with the server. Most of the time, the server comes back and says, "Okay. Yes, this is good." What you had done optimistically work. There's no need to make changes.

Sometimes the server may actually do things. Again, sometimes a server may expand it. For example, the delivery receipt, the little blue check that you see or the small profile picture that you see that someone read your message, that's the server mutating the status on your message. It was something you optimistically sent. It went to the server. As far as these are interfaces are concerned, this message is great. The server did not reject it. So there's nothing new to do, but eventually a server comes back and says, "Hey, I need to mutate this for you because the status has changed. It was delivered. It was read." Now you do the status update on.

[SPONSOR MESSAGE]

**[00:40:10] JM:** Over the last few months, I've started hearing about Retool. Every business needs internal tools, but if we're being honest, I don't know of many engineers who really enjoy building internal tools. It can be hard to get engineering resources to build back-office applications and it's definitely hard to get engineers excited about maintaining those back-office applications. Companies like a Doordash, and Brex, and Amazon use Retool to build custom internal tools faster.

The idea is that internal tools mostly look the same. They're made out of tables, and dropdowns, and buttons, and text inputs. Retool gives you a drag-and-drop interface so engineers can build these internal UIs in hours, not days, and they can spend more time building features that customers will see. Retool connects to any database and API. For example, if you are pulling data from Postgres, you just write a SQL query. You drag a table on to the canvas.

If you want to try out Retool, you can go to retool.com/sedaily. That's R-E-T-O-O-L.com/sedaily, and you can even host Retool on-premise if you want to keep it ultra-secure. I've heard a lot of good things about Retool from engineers who I respect. So check it out at retool.com/sedaily.

[INTERVIEW CONTINUED]

**[00:41:46] JM:** I've done a lot of shows about React, and tracing the origins of this one-way data binding in and why this emerged in popularity within Facebook I think is an interesting story. From what I can tell, it was necessitated by the reality of Facebook being this highly-transactional multiuser application where you have all these little changes going on across the application at a single time, because like you think about why did React come out of Facebook as supposed to Amazon or Google. Well, I mean Amazon and Google, there's a lot of data. There's a lot of stuff going on, but you think about the biggest parts of the application. They kindly get updated in batch. Like you have a big search index that gets updated every – In epochs, then the search index is updated. Then you have the same thing with Amazon, like the product catalog gets updated every night. But with Facebook, it's this totally user- driven thing where you can't really predict how the changes are going to occur. So you need some scalable way of really having a nice update system, and that seems to be why it was like the necessity of having something that would do a better job than two-way data binding that led to just the dominance of that reactive pattern throughout Facebook.

**[00:43:10] MA:** You said this quite a bit better than I would available to. That is spot-on. Any environment where you have a large amount of change that can occur to the data, keeping that in-sync with the user interface becomes an almost intractable problem. Like I mentioned, mobile apps started being very simple, but it's not surprising. We have grown in complexity. That's why you see iOS embracing Combine and embracing SwiftUI, which is a purely reactive design pattern, same on the Jetpack Android side.

I think that inside that Facebook had on the server initially in our HTML world, in on our web world, which was dominated by the enormous amount of data change in UI change and coordinating all of that became impossible until the inside was, "No. If you go to a single source of truth, basically you decouple the read and the write path. You let the write mutate data. Give you a signal the data has changed, then the UI layer can then do the proper read and then a diff model, which also produces higher-quality user experiences. You can now animate the changes. What's changed between the time I was looking at this feed and have the fee changed or somebody – How many people like this story or the comments that came in on the post?" I

think that is the reality not just – It used to be the reality of the server and the web. Now it's becoming the reality of mobile apps because mobile apps are becoming quite a bit more complex, which is not surprising. That's just the nature of software.

**[00:44:31] JM:** Did this rewrite entirely encompass the client-side, the iOS side of things or was there significant work done on the server side as well?

**[00:44:40] MA:** That's a great question. One of the reasons – The answer is yes on the server as well. What we had done with the server, one of things, we had some constraints. So one of the constraints was we can't change the actual backend that stores and delivers messages. One that's very highly deployed, that is probably the largest most deployed messaging environment in the world because it's also not just using Facebook Messenger. It's using the little chat bubbles when you chat with someone on the web and Facebook. It is used FBLite products. It is used in our Android product, which was not yet LightSpeed-based.

We knew we could not change the actual messenger infrastructure. But one of our design goals was this single source of truth and having a single uniform database-centric view of the world required us to build a new unique asset on the server. At the last FA, two of my colleagues, Joshua Evanson and Jeremy Fein, talked about the details of that broker. One way to kind of quickly describe it, the existing product had somewhere between 20 and 30. We don't really know the exact – I don't know the exact number, of independent systems. Some of them were dealing with messaging related things. The stuff you would expect, sending messages, receiving messages. But the contact system was a very separate sync stream with a very separate schema, with a very different system that relates with Facebook user accounts, Facebook users' integrity, who's allowed to message whom, etc. Stories were coming from a different backend. Presence was coming from a different backend. Typing indicators was coming from a different backend.

What we decided in project LightSpeed is to build a single uniform client way of doing that. The only way we could do that is to build kind of a counterpart in the server. So we built a technology on the server we called the LightSpeed Broker, and what it is, it's kind of the brother or sister of LightSpeed on the server. It's job, if you're actually familiar with Django 4 design patterns, probably it's kind of an aggregator adapter design pattern. It basically takes one uniform sync

approach and data approach and the federated back out to the existing system. We are not in a position to go redo all of these systems.

The broker was an absolute key part of what allowed us to keep the LightSpeed code small because we don't have all these independent sync systems. We don't have independent threads each one going on the network. We don't have network streams kind of fighting with each other going on top of each other or around each other. We have a more – Again, we built a freeway basically between the client and the server and we regulated the traffic between the clients and the server and the old product was probably closer to an organically developed cities where people can kind of find shortcuts and go down the streets and find another part of the city and eventually you end up with kind of unmanageable traffic problems as a result. So that's what we did.

**[00:47:18] JM:** There was a platform that you built called MSYS.

**[00:47:23] MA:** Yes, MSYS.

**[00:47:23] JM:** To handle much of the logic for Messenger. Can you talk about what MSYS is?

**[00:47:28] MA:** Yeah, that is the C codebased implementation of Messenger. Like I mentioned, when we had looked at it, we realized we were doing a lot of business logic, a lot of schema and a lot of sync logic in the UI layer. We were doing that in Java and Objective-C, and our intuition was – Again, I think sometimes teams go very extreme about cross-platform one way or the other. Sometimes teams try and do the entire app that's cross-platform, and that's very difficult to do especially given how rich modern UI platforms are, like UIKit, WwiftUI, Android Jetpack.

On the other hand, then people shift to the opposite, "Well, let me just do everything the Java way. Let me do everything the Objective-C way," and we felt that there is a balance in between. We felt that the core schema, the core business logic, the thing that says when I send a message [inaudible 00:48:11] thread is read. That did not need to be Java. That did not need to be Objective-C. There's nothing UI about it. There's nothing iOS-y or Android-y about that idea. It's a database idea.

So we took all of that. We took authentication. We took the core schema. We took all the core business logic. We took that universal sync system that I just talked about earlier and we made that into a single subsystem. We call that MSYS. It's unclear whether M is messaging or Messenger, but you can take it either way. It's probably very Messenger-centric, and that is the asset that we believe is going to be the cross-platform asset that will allow us to take a lot of the LightSpeed experience not just a iOS, but to Android and the rest of kind of the family of applications mantra.

**[00:48:53] JM:** I guess it's worth talking a little bit more about the long-term implications of this project. This was completely focused on iOS, but I imagine that there are other principles you could take from this rewrite and go and apply to Android, the Android Messenger application or the desktop web Messenger application, or like you said, Instagram, or WhatsApp or I don't know if there's some kind of messaging system in Oculus, but messaging is so important to the whole lifeblood of Facebook. What are some of the areas of this rewrite that you can reuse in future messaging rewrites or greenfield projects, in one word, MSYS. That was our answer.

Actually, one thing I should mention. When we had started, the very, very beginning bootstrap was 100% in Objective-C, and then we actually paused for quite a while we decided it was because – Again, we have an enormous number of users on Android and we have – Again, Mark had talked last year, I believe, at F8 about kind of this messaging-centric evolution of the social network. I think it became clear to all of us how central messaging is to the future of all social networking.

We paused and we actually took that Objective-C code and we redid it all at C code and we bootstrapped it with JNI on Android early throughout the development cycle. We always had Android in mind when we built MSYS. We worked with a partner team. We have a great product in Android called MessengerLite for Android, which is a very small and very focused version. It's kind of an amazingly executed piece of technology not as feature-rich as the full Messenger product.

We partnered with that team to kind of keep us honest and to see if MSYS could be something that even in that extreme case, which is something that is very focused only on messaging, only on very low-end devices would be able to use it. We partnered with that team and they kept us

honest throughout the whole cycle. We're fairly confident that MSYS really is our systemic approach for how we bring messages outside of iOS, Messenger iOS to Messenger Android, to Portal, to Oculus. I believe we have actually launched a desktop product. We've talked about a desktop product, which is a rich, complete application. That one actually – When it launches, I don't actually know if it's launched or not. When it launches, it also has MSYS inside as well. That one is on the Mac and Windows and does not use the web. It uses Electron for UI, but it has a full sync MSYS-based. So it works offline. It works kind of as a rich desktop app.

**[00:51:27] JM:** That MSYS platform, I'm assuming that's tightly coupled to Facebook infrastructure. This is probably not going to be something that could be open-sourced in the future, right?

**[00:51:36] MA:** Not that part of it, because – Yeah, I think it's highly dependent on things like Facebook authentication and the representation of users at Facebook and the Facebook integrity systems that tell you whether someone can – It's definitely kind of I would say it's very focused on that. What we have done instead is there are parts of the way we built MSYS that as we look back, we feel like – I think our approach at Facebook is anything that we think – It's not even a propriety issues. It's just a practicality issue. Anything that is highly bound to data structures, the presence of services that would take somebody, it'd be impossible for someone to implement. We tend to keep those in-house. We don't bother. Anything that we feel can generalize, those are the ones we start looking at and saying, "Is there value in open sourcing this?" That's kind of the process that we're doing right now, is kind of doing that triage of saying, "We believe the answer by the way is yes. There are things we did that don't really have a unique tie-in to Facebook infrastructure that would be valuable for other people just like they were value for us, and we're happy to share those."

**[00:52:27] JM:** Cool. I'd like to know a little bit about the management of this project. How many people were working on project LightSpeed?

**[00:52:35] MA:** Project LightSpeed literally started – I mean, the inception was actually really two engineers and one of the VPs of engineering on Messenger who kind of caught wind of that and started going that route. I would say it went from two quickly to about 10, then to 20, and it was probably at 20 I would say for about that year when we were kind of – Before we decided

we're just going to do the full rewrite. Then I would say it went well over 100 engineers probably in the scale of 120, 130 engineers just on the iOS side. Remember, we were doing parallel work in other areas to make sure we're still engaging on Android. We're still doing – It's a very large and complex project.

**[00:53:10] JM:** 120 engineers that were working on the Messenger rewrite.

**[00:53:15] MA:** Yes, at some point.

**[00:53:16] JM:** Incredible.

**[00:53:16] MA:** Again, it's not – Yeah. If you can imagine, again, it goes back to, "Well, we needed the presence experience. We need the bots experiences. We needed the business experience. We needed the business search experience. We need, again, the stories experience. We need the visual composition experience.

One of the challenges of Messenger, if you are on an iPhone, you don't use iMessage to manage your account, right? One of the things we tend to do at Facebook as we allow you from any app to do full account management. You can actually sign up. You can disable your – You can do all of these things on your account even in the app itself.

We take on things that a lot of – Again, in Messenger, we include audio and video calling integrated into the experience and not in a separate app that we can build and deploy independently. You can say that Messenger is not an app. It's actually kind of a platform of a number of communications, experiences between people, and that requires a lot of people.

**[00:54:09] JM:** Understood. If you were going to start over from scratch today, how would you have done this project differently?

**[00:54:18] MA:** Honestly, I would say the only thing – Again, even now, it might still be too early. We would take a much closer look at Combine and SwiftUI on iOS. An Android, again, I would definitely take a look at Jetpack. Again, we made the assumption two years ago, obviously

SwiftUI was nowhere around and Combine in these types of things. I would say that probably would be the top thing.

MSYS itself, maybe one of the differences – Again, there are some internal mechanics. For example, we got the teams to build these sort of procedures for their logic and we kind of made them more embedded on the MSYS side instead on the feature side. We're doing some refactoring and cleanup of things like that. Those are minor things, but I would say the user interface layer – Again, going back to our principles of use the operating system. I think it's time to start for a lot of teams actually start asking fresh questions. We might be a little too early still, but there is the rise of the new user experience and data architecture stacks both on iOS and Android, and I think they're definitely worthy of the teams paying very close attention to them.

**[00:55:16] JM:** Given your deep knowledge from this project, and I'm sure you've done some other work before in the same area. Do you have an oblique or an optimistic perspective when it comes to cross-platform mobile development?

**[00:55:32] MA:** I think I have a cautious. Again, personally, and this is definitely necessarily a Facebook representation, but my own opinion, is that I think the extremes don't work if you try make your entire app cross-platform, including the UI. Actually, I have experience in that as well. It's just a lot more difficult. There's a lot more, and tracking the advances that Apple and Google are going to make are going to drive you crazy. They're advancing these operating systems at the slowest pace once year. It's actually really more than once a year. They have hundreds and sometimes thousands of engineers evolving these platforms. They're building tools to support them. Be careful about cross-platform UI. But the opposite is also true. Ask yourself, again, is this piece of code you're writing, what is iOS about? What is iOS about updating a table and SQLite to say that I need to mark a thread is read? Why does that have to be platform-specific?

I think I'm still – Again, in hindsight looking back, I think it was one of the principles we used that worked well for us. We empowered the team to use latest and greatest. One of the jokes on the team was don't be embarrassed to stack overflow how you do something on iOS, because we're actually just an iOS app for the UI layer. I think that is I would say caution and balance is kind of warranted in this case. Be careful about full cross-platform, but also be equally careful

about, “Well, then just let me do the whole thing in Java or let me the whole thing in Objective-C.”

Now, for a very small app, I would absolutely do it that way. But if you're at the scale of a Facebook or a Google or an Apple – Apple may not care about the Android’s side. But Facebook or Google at least or a are Microsoft and you're working on these large cross-platform efforts. I think the balance is incredibly important there. Having clarity on the principles you used to decide, “Okay, applying a razor that the team that every developer can use and it’s crystal clear. Okay, this code I'm writing. Okay, is has UI view. Boom! Objective-C. No question.” All it is doing is manipulating database. Boom! If I’m doing SQL, I’m not doing Objective-C.” That kind of thing would really help.

**[00:57:26] JM:** I know we’re up against time, but just because I’d be remised if I didn’t asked this. There was not use of React Native in this rewrite?

**[00:57:33] MA:** No. There was no React Native. there is no cross-platform user interface in project LightSpeed. The entire UI layer is UIKit Objective-C based, and everything portable was done in C and SQLite. By the way, the existing Messenger did not use React Native either. I think there was somebody online was saying we have been in React Native. We actually had never used it. We had some small features at some point the use React Native. I don't honestly remember what the feature was. Then that feature was that long before project LightSpeed. The feature was deprecated. So our need for the framework was deprecated, but Messenger, the existing Messenger was a very UIKit Objective-C app, and the current Messenger Android is very activity fragment Java app. Very kind of platform-specific app as well.

**[00:58:18] JM:** Got it. I guess just to close off. The modus operandi for not using React Native is just because the performance wasn't good enough?

**[00:58:25] MA:** It’s also to leverage the latest and greatest advances in iOS. We want to be use all the latest gesture recognizers, iOS animation. Like I mentioned, storyboards. We want to be able to do, again, layout inside of Xcode interface builder where we can – When we switched to SwiftUI, we want to be able to use the canvas and previews and all of these things. Again, it’s that principle of – It’s harder for – Again, I think cross-platform user interfaces are incredibly

useful for certain classes of applications. Messaging in particular is kind of a flagship. It's utilities and electricity of the modern mobile era we're in. These may not be the best applications to try and do cross-platform UI in. There may be a lot of other applications where it makes a ton of sense to do this in.

**[00:59:08] JM:** Mohsen, thank you so much for coming on the show. It's been great talking to you.

**[00:59:11] MA:** Same here, Jeffrey. It's really my pleasure. Thank you so much.

[END OF INTERVIEW]

**[00:59:22] JM:** If you are selling enterprise software, you want to be able to deliver that software to every kind of customer. Some enterprises are hosted on-prem. Some enterprises are on AWS. There might be a different cloud provider they use entirely, and you want to be able to deliver to all of these kinds of enterprises.

Gravity is a product for delivering software to any of these kinds of potential environments or data centers that your customers might want to run applications in. You can think of Gravity as something that you use to copy-paste entire production environments across clouds and data centers. It puts a bubble of consistency around your applications so that you can write it once and deploy it anywhere. Gravity is open source so you can look into the code and understand how it works.

Gravity is trusted by leading companies, including MuleSoft, Splunk and Anaconda. You can go to gravitational.com/sedaily to try Gravity Enterprise free for 60 days. That's gravitational.com/sedaily to find out how applications can run the way that your customers expect in their preferred data center. That's gravitational.com/sedaily.

Thanks to the team behind Gravity, the company Gravitational, for being a sponsor of Software Engineering Daily.

[END]