**EPISODE 1483**

[INTRODUCTION]

**[00:00:00] SPEAKER:** This episode is hosted by Lee Atchison. Lee Atchison is a software architect, author, and thought leader on cloud computing and application modernization. His most recent book, *Architecting for Scale* is an essential resource for technical teams looking to maintain high availability and manage risk in their cloud environments.

Lee is the host of his podcast, *Modern Digital Business*, an engaging and informative podcast produced for people looking to build and grow their digital business with the help of modern applications and processes developed for today's fast-moving business environment. Subscribe at mdb.fm and follow Lee at leeatchison.com.

Blockchains are distributed ledger technology underlying Bitcoin and other cryptocurrencies. More broadly, a blockchain is a mechanism for updating truth states and distributed network computing, producing consensus trust and serving as a new form of general computational substrate. Lewis Tuff is the VP of engineering blockchain.com and joins the show to discuss the engineering behind the applications that involve buying and selling cryptocurrencies.

[INTERVIEW]

**[00:01:15] JM:** Lewis, welcome to the show.

**[00:01:16] LT:** Thanks. It's great to be here.

**[00:01:19] JM:** You work blockchain.com and you have some background in other financial industry applications. Now, I've seen some of the mechanics behind a traditional order book of buying and selling equities, securities. How does the engineering behind an order book for buying and selling cryptocurrencies compared to that of traditional equities?

**[00:01:49] LT:** Great question, I think in a lot of cases, we've actually modeled the implementation on Wall Street, to ensure that, one, we're able to build a system that competes with or at least matches

warship annotation, and two, can reach to kind of scale throughput latencies that have been achieved over kind of many decades of iterative work.

So, in terms of how you model cryptocurrencies, we basically looked at it and treated it like any other asset. On the back end, in our matching engine, we're able to model kind of an arbitrary contract with any kind of defined parameters and that could be a civic token asset. That could be some derivative, sounds like we support right now, but potentially, in the future. And the matching really is been focused and built around being extremely fast execution, extremely high throughput and scale, and product agnostic for the most part. So, all the complexity and lifecycle and state on the product specifics, in this case, kind of tokens, assets, derivatives, or any other permutation is all built outside of that. So, we really kind of take a first principles approach, and keep each component of the system as tightly constrained as possible, and that realized you to scale and grow and continue to evolve the platform effectively.

**[00:03:17] JM:** Can you walk me through the lifecycle of a trade being executed?

**[00:03:22] LT:** Yeah, of course. So, we have a few different entry points into our active trading venue, or blockchain.com exchange. First one is interactive trading, going through the UI, and so that's backed by a combination of both WebSocket and REST API's. You can go through our public gateways. So, these are our REST or WebSocket API's directly, building trading bots, or some automated or systematic trading effort. The final way is actually going through codification. So, we offer codification for clients that wish to invest in that, and we have a binary gateway or a DMA gateway, that gives you direct access into the matching engine. And that's using a schema known as Fix, which is kind of the industry standard.

In the space, analyze for both traditional and new clients to connect through a common interface, and so in terms of the lifecycle, what happens is that a client connecting through one of these gateways will construct a new order, and if you're doing that for API, you're obviously construction that, we have an open socket connection for the WebSocket. So, send the command down the WebSocket channel, which is a new order single, which is parameterized with the market you wish to trade. So, the pair of assets was trade, how much you wish to buy ourselves, the direction and the amount, and that then goes through the gateway, which does initial kind of status check and validation, that the command is

as expected matches the schema parameters aren't at a range, et cetera, and then hits a risk engine. And the risk engine component is responsible for all the accounting.

So, making sure that you have sufficient balances in a token you're wishing to trade. If it takes, say, the Bitcoin USD market, if you're trying to buy bitcoin and sell dollars, please make sure you actually have the sufficient dollars available on your account to be able to then buy the Bitcoin that you've specified. So, risk hedging does those sync checks and the kind of the accounting and does some final validation on the message schema and structure, and it can do an immediate reject then, if balances are insufficient, or there's some other issue with the parameters of the event. If everything passes through, it will then be sent into the matching engine. And then at that point, depending on the order type, if it's a market order, it will just sweep the current order book. So, we'll look at each the price levels and the amounts and try to match off immediately the amount you wish to buy. And it was a resting order, I set our limit price, it will then sit in the book until that order is crossed, i.e. an opposing order comes in.

So, in this case, a sell for Bitcoin and $5 that will then get matched off within the match engine itself. And then the matching engine will then send out an execution report, which is an event to confirm either the orders been placed or the trade has been executed, and that gets passed back through the stack. So, back to the risk engine to amend the in-memory accounting imbalances of the user, and then back to the gateway to publish this to the client, either the interface or a programmatic client. Does that make sense?

**[00:06:50] JM:** Absolutely. The lifecycle of that trade, what parts of it have to be serialized or atomized? Are there places where race conditions can create problematic circumstances?

**[00:07:07] LT:** Oh, 100%. So, if we're talking about, say to the public gateways or cloud infrastructure, so we have the REST and WebSocket gateways and the front end sitting in front of those, they are stateless, for the most part. There's some caching, but for the most part, are stateless services. So, that means we can horizontally scale them based on demand and client connections. As I mentioned, for every client, that client's republic gateway, they're going to have an open socket connection. And so, we need to make sure that we are load balancing those over an appropriate number of instances of that service. The service itself is stateless. We use kind of some shared distributed cache, for kind of live order updates and events, but nothing critical to accounting or balances. And then the components

behind the gateway, so the risk engine and matching engine are single threaded, and execute things like all of the commands and personal events sequentially.

Everything gets stamped with an identifier, the sequence number, and is processed in the order that is received. That's super important to ensure the synchronization of the overall platform and system, and as you say, to make sure we don't hit any kind of contention or threading issues that you would see in, say, a multi-threaded environment.

**[00:08:35] JM:** What do you use for distributed caching infrastructure?

**[00:08:38] LT:** So, on the public gateways, we are using Redis. There's a key value store for storing mainly informational data that can be used in the front end or pass back to clients connecting, because the public networks are non-deterministic, and not always the most reliable when it's outside of our control, right? So, we want to make sure that if a client temporarily disconnects for a few milliseconds, or in a few seconds, that when they reconnect, they've not lost all of their state, because maybe they didn't get a chance in the shoe molded events. Same on the front end, we offer mobile clients and browser-based clients for the exchange.

You can imagine, if you're in a car or moving around in the city, you're moving between masks, your connection is constantly going to be moving with it, and traversing that network in a slightly different way as the packets are being passed to our infrastructure. So, for us, yeah, we use Redis as that kind of caching layer for this informational metadata that helps with understanding where you are in terms of your open orders, any activity that happened during a disconnect, and a few other kind of key pieces.

**[00:09:49] JM:** I'd like to talk about wallets. Operating your own wallet is not a fun experience. But of course, it maximizes security. There's this tradeoff between security and ease of use for wallet design, and this is particularly acute for the exchanges, who have a lot of non-technical users. What's the ideal wallet design for this mix of technical and non-technical users?

**[00:10:21] LT:** That is a large question. So, I think, how to answer those in a few parts. One is that we do have a very diverse and broad user base, and there are extremely early adopters that have been with us on this journey over the last 10 years, and really understand the nuances and mechanics of what's going on, and are what you deem a kind of a power user. In that case, they want maximum

transparency, maximum control, and visibility into the transactions are being instructed, how they're being broadcast on chain, and access to all of the different applications that are going to build on top of these protocols.

And then, as you mentioned, we have exchange users that maybe are coming to the crypto industry and domains the first time, and really what they want to do is just speculate and buy and sell. So, for those users, they're really looking for an interface that is familiar, and very much reflects what they're used to in terms of financial services that they use on daily basis. That really means I'm looking at brokerage accounts, traditional brokerage accounts, and he's looking at challenger banks, or your banking services apps, or other kind of light investment platforms. So, when we think about a wallet, that's kind of the broad spectrum, right? It's the self-sovereign individuals that want to manage and own their keys, and never deposit or send their funds to us or any third party. And then you have the group of innovators on the other side, who really just want to access some part of the ecosystem, or get some exposure in a very light touch unfamiliar way, and that's a fully custodial offering that we offer on the other side.

So, what you have to do there is be very intentional about the user, the persona of that user, what their behavior is, and expectations are, and what is the outcome they're trying to achieve. Instead of trying to kind of conflate all these different user profiles into one wallet context, in one app experience, what we're actually doing is delineating between, okay, what is your intention, and what is the outcome you want to achieve here? Do you just want to buy the underlying asset and then sell it at some point later, and you do not want to vote on a DAO or buy NFT, or access on decks or some other kind of crypto native experience? Okay, in that case, we can take you down as tater flow, or we recommend the custodial offering, and everything that you touch and see and feel from first, signup will be very familiar, because will simplify and abstract away the complexity. The beauty of that is that then later, you can introduce these new concepts piece by piece, right? Rather than overwhelming the user, the first time they come to your product.

That's our current strategy is like, we add a lot of products and features, and now we're really thinking about clearly defining these intent driven flows, to tailor the whole product experience end to end to a power user versus a completely new user in the space that wants to just understand what their friends have told them about recently, and dive a little bit deeper.

**[00:13:36] JM:** Is there a lot of off the shelf wallet infrastructure that you can use? Or do you have to spend a lot of time rolling your own security systems for protecting wallets?

**[00:13:47] LT:** So, we are one of the largest non-custodial or defy wallets on the market. So, we have invested a decade or more of time and effort and energy to build out that infrastructure, and provide that abstraction on top of the underlying protocols to technology, and the security concerns and constraints. So, all of our wallets are open source across the mobile clients and the browser, and we built all that in house to answer your questions to succinctly. We've built out the noncustodial element, which is the path to self-custody, your assets, as well as the custodial element in our custody system in house from scratch.

**[00:14:30] JM:** Coming back to the trading. I wonder about risk of holding assets, because if you consider yourself a store where people can go and buy crypto, those assets can obviously fluctuate in value. The more you can match trades between people and just have arbitrage there, it's obviously more ideal, but you probably have to hold some crypto unless you buy crypto on the fly from other exchanges when an order comes in. Can you tell me about the mechanics of what crypto you hold, and just how you think about the risk of price fluctuations?

**[00:15:13] LT:** Yeah, I think, just one thing to clarify, first of all, before I ask the full question is like, we have two main consumer products for buying, selling, swapping and holding. One of them is the brokerage product that sits within the wallet context and app. One of them is the exchange, the active trading venue. The brokerage product really is your one-click checkout experience. So, these are for users where it's their first time entering a space, they don't necessarily have any kind of financial markets, knowledge or experience, they want to just get exposure to that underlying asset.

In that case, we are familiar on ramps, so with card, bank, and other aerators. You can buy with fiat and hold a crypto asset. And if you do that with us, yes, we will be holding and costing those crypto assets on your behalf until which time you wish to sell them, or withdraw them from your wallet. In that case, that sits in our custody system, which has gone through kind of security audits and reviews, and it's kind of multi-tiered, to ensure that only a very small percentage are ever available in our hot wallet, everything else is in cold storage or offline.

So, with that model, the user themselves are buying the exposure to a particular asset, and if that goes off, or goes down, that's on the user to decide when they wish to sell and how long they wish to hold. There's no kind of exposure or risk on the company. It's just on the market generally in terms of dynamics of how the asset performs. If you think about the exchange, in that case, we're not on the other side of any of these transactions. So, this is a two-sided marketplace where we're bringing together buyers and sellers of assets, and they are placing these resting orders, limit orders into the book, or placing market orders to immediately fulfill and we're never stepping in as a counterparty in the middle. The matching engine is fair, it's looking at press time priority of these orders coming in which were sequenced, and only if there is enough of the particular asset on the opposing side available, will it be matched off otherwise, the partial order remaining, or the full order if it wasn't matched off with any of the counter currency at that price, we just sit resting until which time there is an order that is suitable to be matched.

So, in that case, the exchange is a custody system where costing the assets, but again, there's no effects risk there to the company, or to the individual, right? They're just holding the assets that they deposited on the exchange initially. So, whether that's fiat or that's crypto, that's up to them. Again, it comes down to market dynamics rather than inherent effects risk in the system on either side of transaction. In the brokerage, the company is fulfilling the other side of the transaction and you're buying and selling from blockchain.com. On the exchange, you're actually buying and selling from our counterparties in the market, over uses of our exchange, and then maybe institutional and then maybe retail.

**[00:18:26] JM:** Are there any other places where the company is subject to price fluctuations or risks of dramatic market downturns, things like that?

**[00:18:38] LT:** Sure. We offer a very broad range of products on the institutional and on the consumer side, and we have a very successful lending business. There, you are both borrowing from clients as well as lending to clients. And there we have a team of individuals that are managing the credit risk and managing the lending book. So, there is very much going to be around focusing on the price of any assets any one time, what collateral has been used to service that loan, and then what are the margin requirements on a specific loan.

**[00:19:17] JM:** Can you tell me more about the engineering behind a lending business?

**[00:19:20] LT:** Well, we have a rewards product in our wallet where users are able to deposit funds using the same platform. One of the things that we've built within blockchain.com's platform is to make sure that we're building the core layers that are product agnostic and offer maximum leverage across the entire stack. So, what I mean by that is like for every product or feature that's being developed, we're not building out redundant components. We have a single costing system that has been security audited and iterated on, and is now the core place for all custody assets across all of our different product lines, whether that's on the traditional side or on the retail side. Same with our ledger. Same with authentication, authorization. We've built out these kind of platform layers and we have teams focused on maintaining those and securing them.

On top of that, we're able to build out these different permutations of product experiences that we can then offer consumers and institutions. So, the rewards product takes advantage of using our costing system, using our ledger, using our payment rails, on and off ramps, it ties all those components together to build out his new product experience, or from a user's perspective, all they have to do is decide how much of their assets they want to deposit, and then we'll start generating return was posted assets, calculating that daily and paying that back week. Behind the scenes, those assets are then available to our lending desk, which are using any kind of borrowed assets to generate returns and we're passing that directly on to our consumers.

**[00:21:00] JM:** I think it's worth talking here about programming languages. We've talked about a variety of sensitive transactions involving lots of money. So, safety is a big concern here. I can see a lot of candidate languages for these different systems. Obviously, Java, so widely known for trading infrastructure. Go Lang. What's your language of choice for some of these financial execution platforms?

**[00:21:30] LT:** Yeah, great question. So, we have a wide variety of skill sets and engineers within a team. But the two most common platforms and stacks are based around C++, and around the JVM, so Java and Kotlin. And anything that's on kind of the critical path of trading, so placing or purchasing orders, updating kind of the accounting in the risk engine, or doing the matching is all built in C++, and that really then gives us the most control over the execution flow, as well as the underlying hardware to make sure we're kind of optimizing our execution path for the underlying architecture.

Yeah, as I mentioned before, we have colocation for the exchange offering, and so as part of our system is on prem, it's not all in the cloud, and we've done that to be able to maximize throughput and minimize latencies. So, there are two primary languages that we source for, and hire into those roles here. Aside from that, there's a number of other languages that are used in data science, these Python and our SRE teams for automation. They use Go Lang, as you mentioned, but I think in terms of the financial service components and infrastructure, it's more and more being dominated by C++, and periphery components are built into JVM, Kotlin, and Java.

**[00:22:54] JM:** Can you help me understand the delineation between JVM platforms and C++, because I can obviously imagine, a C++ for the low level mechanics of trade execution, like maybe if you have some really big pool of orders of buy and sell orders, and you have to pair them, you have to do some matching, that's really latency sensitive, lots of filtering and sorting operations that you would want to get really quick. Then on the other hand, the JVM side of things I could see just for calling into that lower level C++ infrastructure, and you can take advantage of the greater safety, higher level of abstraction for actually calling into those low-level functions. But maybe you could tell me more about the interface between those two platforms.

**[00:23:50] LT:** Yeah, sure. I mean, you have described it pretty accurately. I think that, for us, the delineation is around kind of public gateways that are stateless, and need to scale fast and service both open source technologies and platforms. We mentioned Redis. We use Kafka for event-driven architecture and message passing. So, like these libraries, and integrations are common practice and readily available in tried and tested in production environments. If you go to that C++, there's less of these available. So, you either have to build them yourself, or find some niche library that there may not be battle tested.

Generally, for a service where we need to spin up public facing API, whether that's WebSocket or REST, or maybe even internal services for gRPC, which we have, in some cases. There, it's much easier to do that and quicker to do that in these abstracted languages, where this is kind of common practice and happens all the time. And then, in terms of the interface between them, as mentioned, we are using Kafka to basically message paths between the JVM world and then consume it into C++ world. That's really kind of the interface, that primary interface we're using between the two.

**[00:25:15] JM:** Got you. How else are you utilizing Kafka?

**[00:25:20] LT:** we mainly use it as an event log for message passing, and then offloading state to periphery components for reporting analytics, for persistent state onto disk. So, anything that's kind of in memory and decisions remaining fast, we then need to synchronize and persist at the disk. So, we have transactional log, which is already being used for those primary cases. Message passing between services, and acting as a bridge there and then offloading events that need to be persisted for archiving an event log.

**[00:25:53] JM:** Are you utilizing any of the Kafka native features like Kafka streams, or KSQL to build layers of transactions on top of Kafka?

**[00:26:08] LT:** No, we've kind of veered away from doing that, especially in the trading system. We're very much using it in a deterministic high throughput way. So, we want to maximize our ability to write and read back events in the same order as quickly as possible, and we haven't yet kind of gone down at least in the trading system, partitioning route and adding any transactional logic outside of the application itself. There are definitely other applications in our broader platform where we're using partitioning and sequencing doesn't matter as much, and so we do them paralyze transactional activity and assumption, and we're able to kind of have single producer and multiple consumers. In that case, you've got different architectural model. I mean, trading system, it's very much sequential, single threaded, and everything is processed in sequence.

**[00:27:05] JM:** Is there any significant downside to using Kafka for message passing in this kind of low latency infrastructure? Does Kafka serve effectively? I've seen obviously, a variety of message buses, but I guess Kafka is so resilient and fast at this point, it's probably up to snuff for this kind of application.

**[00:27:29] LT:** Well, it's a great topic. It's something that we discussed a lot and when we first launched exchange, like three years ago now, Kafka was kind of the go to, to get something into the market quickly and using segments battle tested, robust, resilient, and has a set of best practices around how you scale it. For us, what we've done over time, is actually remove Kafka from any of the critical paths of trading decisions, and really just using it to say offload events for third-party processing, for downstream processing, and not using it to execute any order or trade decisions. So, that's really the pathway we're going on now. It's definitely great in a system where you're not sensitive to latency, or

sequencing, I think more crucially, but in a system where you care about sequencing, latency and throughput, yeah, Kafka is definitely not the appropriate choice.

**[00:28:32] JM:** I'd like to know more about some of the higher-level building blocks that you use to build this platform to get a pretty good picture of it at this point. The lowest level of trade execution, you have C++, you have Java serving as kind of a middleware layer, Kafka for message passing between different services, Redis for in-memory caching. There are a few other points we haven't really talked about, transactional database, as well as deployment medium for how you're deploying your services. So, maybe you could talk through some of those other infrastructure decisions.

**[00:29:16] LT:** Yeah, sure. So, on the database front, we've predominantly rolled out a Postgres across any kind of relational database, and/or object database. Postgres is extremely versatile, fast, and if tuned appropriately for your workload, serves a lot of different interesting use cases. So, we're using Postgres for a lot of our persistent state. And, yeah, where, again, throughput and latency is needed, we've done various optimizations to make sure that the database is not on the critical path of those decisions. But in a web-based application, Postgres serves a great purpose.

We do have some SQL, but that's more for some legacy components, and say we've kind of converged on Postgres moving forwards. We run that in a cloud environment and leverage services to remove a lot of operational burden around managing 50, 100 different databases across many services. And then in terms of our database systems, we use a few different, very bespoke implementations for kind of fast in memory, execution and computation, which is a memory map to files and offloaded. But yeah, in terms of like, traditionally, data is Postgres are the primary two. And then in terms of other items around deployment, as a kind of alluded to, we do operate most of our stack in the cloud. So, we built a containerized architecture and deployment system, where every service no matter what the underlying platform or language, whether it's built in Java, Kotlin, Python, Go Lang, C++ is being wrapped in a Docker container.

We define the Docker images and build all those in-house ourselves and assign them and they become the base images for the doc files. And then we have any kind of bespoke logic or customization for a particular microservice. And then we have some internal tooling to bootstrap the creation initiation of the services in our development, or staging or production environments. And we use kind of Nomad under the hood, from HashiCorp to actually do the orchestration itself. So, we have to basically wrap the

Nomad API's, enrich it with some internal tooling, for metrics, learning, monitoring, so that every engineer gets a minimum set of visibility and observability into their component, as long as they follow this schema and manifest for that particular component.

And then yeah, if you don't have any specific constraints on your service, that you just want to deploy a new micro service that is going to be integrated with some other system internally, then you can do that very simply and easily. If you want to build a micro service that exposes some public routes, or some additional configuration to expose those for security reasons. But again, it's kind of a simple process. But all of that stems back to a common CI/CD process. Everything is defined in Docker files, containerized, and we have a centralized system for managing access controls, team permissions, et cetera, et cetera.

**[00:32:41] JM:** You mentioned Nomad. Why would you choose Nomad for your orchestration?

**[00:32:47] LT:** Well, it's a powerful orchestration tool that actually lives you to span both on premise and multiple cloud environments. We're using this to actually manage the lifecycle of our binaries, or the role binaries that we want to deploy to on prem boxes, or the containers that we want to instantiate, within our cloud-based environment. So, we've actually invested a lot in working with the system to take full advantage, and we're doing everything from optimizing it for our training systems. Implementing kind of core pinning, and CPU isolation, and making sure we're aligning components that require themselves to be NUMA aware on the VMs, appropriately, to also then just running as I say, a vanilla container that is just a JVM based service that exposes a few endpoints that needs to be consumed by a public or a private component.

So, for us, it's served well. We've been using it for a long time and we've both contributed back to the source project and spoke with the team on some of our kind of more interesting workloads, which do kind of come down to this optimizing for low latency and high throughput and our trading infrastructure and crossing the boundary between both a cloud-based deployment and an on premise.

**[00:34:17] JM:** Do you miss anything by not getting exposed to the pace of change and the ecosystem of the Kubernetes community?

**[00:34:28] LT:** I think that Kubernetes is a powerful tool and it does have significant community behind it, and I've actually been involved in Kubernetes community for a long time and was playing around with it during the early beta days and on the channel when there's only a few thousand people on the Slack channel. It was great to be there seeing the evolution of the product, and how engaged the engineering community were in making this a success and contributing back and improving the whole experience. I do think that Kubernetes is not a one size fits all. So, you have to think carefully about what you're giving up by leveraging Kubernetes, which provides a lot of abstractions, and does take care of a lot of the wiring of how you deploy and connect services, how you manage firewall rules, how you manage an app configuration. It's very specific in defining that state, the schema and the lifecycle, which is great if you have simple services, and needs to do something quickly.

I think if you want to do things at our scale, where we have kind of 600 plus containers running, we have five and a half to 6,000 cores running – I mean, it's a significant footprint. I think they're kind of putting all of the control of the management of your platform stack into software that you didn't write or don't understand fully the mechanics off, I think it's problematic. So, unless I had a team that was deeply familiar with the internals of Kubernetes, I wouldn't want just going to switch this overnight for the whole platform. I think having specific projects that could benefit from it, for example, more elastic dynamic projects, where you're not sure on the amount of compute you may need. For example, data science is a good example of this. It would make a lot of sense to have a dedicated Kubernetes cluster, using GKE and GCP, so you don't have to manage all of the underlying infrastructure to let the key value store and the state, you kind of pay a small premium, and I source that, and then they can be very elastic on, okay, we want to run a bunch of research jobs, that are going to do a ton of batch processing, but they're going to be done in four hours, they want to set it like that. And then you kind of get the real benefits of having this kind of very dynamic infrastructure, where you really just want to use compute on demand, you don't really care too much about how that is done, as long as you get your own goal.

Whereas a lot of our workloads across the stack, whether it's on chain activity, whether it's building these systems that are optimized for speed and throughput, there you really do care about how your workload is executed, and in which environment it is being executed. And there, you really need to control the stack end to end both from the network level, as well as the underlying architecture of the hardware, which, again, is why we went on prem for our trading system, our exchange, and didn't go into the cloud, because there you have control over every component the system should you need it.

**[00:37:46] JM:** That's an interesting infrastructure decision made, essentially because of scale. We didn't talk that much about the transactional database layer. That was another thing I want to talk to you about. As far as the transactional layer. Okay, so Postgres, there's obviously been some newer databases, that have come out since Postgres that have challenged it as the OLTP database of choice. And notably, CockroachDB or Spanner. And then I guess you could say PlanetScale, the PlanetScale is, I believe, mostly scalable, Postgres. But can you tell me more about your database choice and maybe the constraints on your database? And yeah, just take me inside the database world of blockchain.com.

**[00:38:34] LT:** Sure. Yeah, this is an evolving space and topic, for sure. I think there are a number of abstractions on top of Postgres that have been built for specific use cases, like Time Series DB is another one that we're looking to use, and some of the team are very familiar with, which out of the box gives you a time series database built on top of the kind of solid foundations of Postgres, and all of the integrations, tooling and plugins you get with that. So, that's pretty powerful, and so we are looking at kind of different flavors of Postgres, and how you can kind of take advantage of them.

I think that in terms of the decision making, with all our infrastructure and at our scale, there's a few kind of kind of parameters that go into the heuristic for how to make these decisions. Security is number one. Are we managing or interacting with payloads or data rescues that are highly sensitive in some manner, shape or form? Second one is do we believe that the underlying service or system or software we choose will be able to meet our demands in scale, both in terms of throughput and as well as persistent data on disk and then the challenges of managing double digit terabytes of data at scale. How do you do that effectively? Kind of partition management in offloading those partitions and how you can do that effectively? Is there a huge operational load on the S3 team to manage that? Are there services or tooling that available to with your choice?

The third thing is consistency. Do you need eventual consistency, immediate consistency, like strong consistency? What are the constraints on the system or on the data that you're storing up? So actually, we did do an experiment and we did roll out a production set of services for our Ethereum ingestion. So, ingesting the Ethereum chain, and then presenting that back off a set of API's, both publicly and internally to support our products in the wallet and elsewhere, using Spanner so that we have immediate consistency on a global scale. I think that we tried Spanner, we've also rolled out with our

Bitcoin ingestion RocksDB, and using more of this in memory, memory mapped database, that we then synchronize ourselves to the application layer.

So, there's a number of ways of doing this, and then you have to think about consensus and how you reach that, and do you kind of build that yourself. So, we have gone through a number of these topics, and I think that in the most general state and sense, Postgres is a good solution to a lot of our problems, where you just want to capture an event and persist it on disk, and access that easily through both programmatic and third-party products. I.e., you want to serve as data off the product managers, so they can interrogate the behavior and performance of their feature or product.

In that case, Postgres is a good fit, right? It solves all those problems. We then have built libraries in house and ways to kind of interact with database layer in a lightweight way without having to kind of reinvent the wheel for every micro service that needs access to a persistent store. That abstraction is helpful, because you get leverage internally, as more and more engineers come on board. And we use kind of lightweight frameworks like Juke, which is a Java query language, like a fluent interface on top of SQL, that gives you kind of type safety around your queries. Using these lightweight frameworks, building out the internal libraries, means your engineers can move forward faster.

So, really, like coming back to the original decision making, it's like, does it require heightened levels of security? It wouldn't match and meet our scale, and doesn't meet any consistency requirements, we go through that and then understand, okay, do we have a product or service that already fits that mold? Or is there something unique about the problem we're solving that requires us to rethink our approach? For us, yeah, we are a fast-growing team. But we're still pretty lean across a lot of different key areas. And we're focused on kind of hiring in high performance engineers and not just hiring in kind of hundreds or thousands of additional engineers. So, I think, small focused, cross functional engineering teams that are high performing, really drive maximum value. That means being commercial, product focused, and really thinking about, okay, yeah, there is this new shiny tool. But is it really providing any additional value? Does it solve a new problem that we couldn't solve before? I think in most cases, the answer is usually no.

**[00:43:30] JM:** So, when you look at decentralized trading infrastructure, most notably, the ones built on the Ethereum blockchain. Are there ways that you can utilize decentralized trading infrastructure to underpin your own trading infrastructure? Or does it make sense to just stick to your current platform?

**[00:43:56] LT:** Yeah, we're actually looking to build a both crypto native experiences, and so, we're launching NFT marketplace. We're looking at supporting new protocols and decentralized apps on top of those protocols and the ecosystem around them. We have like rolls out right now to hire in Solana engineers, for example, to help kind of contribute and build out the system there. And in terms of decks, specifically, I think there's a few different product choices you made. It's like, do we want to build an aggregator where we source liquidity from multiple places, and then present them as a unified product experience to the user, so they can maximize the ways that they can access liquidity, but don't have to jump through all these hoops do that manually.

So, I really see this as a complementary product to a centralized exchange in our book. And at some point, maybe we want to bridge the two, but right now, yeah, we see them as complementary and we want to build out and provide a gateway into these crypto native experiences for those that are interested in diving deeper, as well as the custodial experiences, which are familiar products and features for people day to day.

**[00:45:03] JM:** Cool. Well, thank you so much for coming on the show. It's been a real pleasure talking to you.

**[00:45:06] LT:** Amazing. Thanks a lot, Jeffrey. Appreciate your time.

[END]