Transcript

Edward Wible: Good morning, can you hear me okay? I'm Ed, this is Rafael, and we have been building Nubank since day one, the initial lines of code, together; we started as a credit card issuer in Brazil and we expanded to a full digital bank from there.

Growing Quickly in a Complex Domain

So, if you want a sense of scale, we are at 2.6 million customers, and that is a lot when you consider the complexity of the domain model. There's a reason why core banking systems run on main frames in most places, it is a large domain. I'm going to try to illustrate that for you a little bit to the extent that I can.

And, you know, we have hundreds of millions of purchases, purchases coming from people visiting about every country on earth. But we are not a bank. We are selling financial services, but it feels like a tech company, we are a tech company, we deploy 20 times per day, have 100 microservices, and crossed 100 engineers on the team. So what we will talk about is a credit to the engineers in Sao Paulo.

Inmutable Themes from Our Stack

And so on our stack, we wrote credit card processing from scratch, we used modern technology. We love Clojure, it is an opinionated functional language. It does not mean you cannot write object-oriented patterns, it keeps us lean and helps us to scale.

And we use Datomic for most transactional workloads, it is like Git for your data. You never lose anything, you made a fact true, and then later not true, but you never do an update in place and lose that history. We will talk about interesting properties of that.

We are heavy users of Kafka, our patterns are by Kafka, we like that concept of immutable log, it helps us with logical and temporal decoupling. And the financial services industry is legendary for big batch jobs, we are starting to understand why, but one of the things that we do is we distribute batch jobs as a stream of messages over Kafka and we can model everything in banking as stream processing.

And lastly, this is obvious, but we are cloud native, we architect everything on native VS from the beginning, four years ago, and infrastructure is code. We do blue green deploys for new versions of services and for the new versions of the entire architecture. So we will spend an entire version of Nubank and kill the old when we re-route DNS, to avoid mutating things in production and help to scale.

Functional Benefits

And lastly, on the intro, we have seen a number of benefits from taking -- a functional programming philosophy from the start of the company. On hiring, we see it as positive self-selection, people that have the courage to try out and try a technical exercise in a language they have never used before are generally good people to hire. We have seen that with over a hundred services and millions of customers, the complexity of the domain is under control. We have things that are difficult to deal with, but we always are able to untangle those things. And that's the architecture of a bunch of small pure functions. And consistency; Clojure is a language that encourages a bunch of small language features into big things, and people move between teams and every service is similar, it is a consistent architecture. And these are our headquarters in Sao Paulo.

Core Banking + Credit Card Architecture

Our initial architecture for credit card- I will put a table so it makes sense, but it is complex. This is the MVP. So in order to get a card passing in a machine, right, we had to build card origination, logistics, shipping, billing, payments, charge back, a whole back office tooling, etc. The next thing we built was KYC, credit scoring, and credit management; this is the first place that we put machine learning into production. And after that, we started to take ourselves seriously as a system of record. We rely on ourselves to know how much you owe us, that's where the general ledger system comes in, we will talk about that in more detail.

And more recently, we brought in card authorization in-house, we built a Mastercard authorizer from scratch and we built an ETL, and we launched rewards, and we launched a bank account. This is not the traditional path, we came from credit card and went to core banking. The key features are earning money on your savings and using the same account like a checking account for peer to-peer transfers. And by American standards, it is a good investment, we are paying 7 percent per year risk-free in Brazil. It is a good place to put your money.

Banking adds a few other components, including real-time transfer, and infrastructure, we will talk about that in Nubank. We cannot cover all the modules, but we will go over the highlighted ones to give folks a sense of how things work, starting with purchase authorization.

Purchase Authorization Value Chain

Rafael Ferreira: Good morning, guys. So, I thought we would start with maybe the most representative flow of our first product and, in many ways, our core product, the credit card. So let's talk about what happens when you make a purchase. So, when you do that, you are actually setting in motion a whole set of participants of the payments chain. So you are interacting with the merchant, and the terminal where you swipe your card is usually managed by another company, a credit card acquirer. The acquirer is the company that maintains the relationship with the merchants, it connects merchants to the broader payments chain.

The acquirer will then move forward through that transactioning information message to the credit card network or brand. We are Using MasterCard, Visa, American Express, etc. And the brand will send the transaction request to us, the issuer, on our end.

Issuer Authorization

And that's our role as a payment institution, we're the issuer. That means we maintain the relationship with the card holder, the customer is a banking customer, and this is an important part of talking about transaction authorization- we take risk for the customer. If the customer fails to make do on payments, we are on the hook for that account. So that's why we have the final say on whether a transaction was authorized or declined.

And now, technically, how does that work? On a data center that we control, the credit card network will place several devices, connecting to the world-wide network. And, one of our servers, we run a service we are calling an authorizer, which you connect to that -- to the brand devices, the specific device that we connect to and calling it here the MasterCard interface device, it is an Edge device that accepts TCP connections. Once we connect, we start receiving authorization requests.

And a couple of interesting points to note here is that we are a network client, not a network server, and that means to achieve concurrency of the transactional details, we have that

connection. And when we connect to the real live payments network, we receive authorization requests immediately, which is a bit odd since we have not issued any credit cards at that point. We are receiving attacks, either in brute force for credit card numbers, or in people using the data from the large credit card breaches in the past to try to attack us. And the payments -- they are constantly under attack, that is something that you need to know about before you undertake these kinds of projects.

So the protocol? The protocol we use to connect to the branding device is the same that flows throughout the payments chain, it is based on an international standard, ISO-8553, it is not like HTTP where you can view the spec and get the information, we had to get it from MasterCard, we had to consult a lot of documentation for the compliance implementation, and sometimes that was not enough. At one point in the project, we had to pass a signature verification algorithm, and one of the engineers had to build a tool to brute force of possible combinations to understand what we had to build.

And you can also see in the slide, we are featuring a hardware security module, that's a device that stores a primary key on secure hardware, and every other key used for the encryption parts of the protocol are derived from that key on the HSM. And one of the ways that we use the HSM, when you do a chip transaction, that microprocessor, the chip, we generate a cryptogram, a small amount of data that flows through the network to the services, and we pass it through the HSM for validation.

So in the bank, Edward was talking about how much we love Clojure; most of our code is in Clojure. And we thought, for parsing that binary protocol, we can take advantage of a library, S Codec, and we can achieve a lot of things by building on it. It is easy to build a parser, and the system helps to ensure that all the parts are fitting together, we get earlier detection, and a cool thing is that it makes it easier to evolve that parser over time.

When we are doing one of our early tests, for instance, we thought some of the fields were going to be encoded as S-key strengths. When we were doing the live tests, we figured out we are receiving the old main frame protocol strings. We had to change a couple of references in the parser three and we were able to complete the test live while it was running in the test window. So, this kind of library for parsing protocols is something that was very helpful for us.

Issuer Authorization: Requirements

And now, stepping back a little bit and talking broadly about the authorizer project, there were two main requirements that drove many of our architecture decisions. One is the availability requirements. We, of course, on our cloud services, we make sure that they are available, that we have observability measures, we have measures in place to bring back service when it fails. But when we are talking about a project like an authorizer where the customer can be trying to make a purchase in a gas station at midnight with only our card in his pocket, it is very critical that every transaction goes through. So availability was critical.

And the second requirement that is important to us is that we are building a physical infrastructure. As mentioned in the beginning, most of our experience has been on cloud services, most of our services we are building up on the Amazon public cloud, we take a lot of advantage from automation that the public cloud affords us. And then we have to face the prospect of building something on physical servers, we had to build all of that automation ourselves. We could not just to it manually, it doesn't work like that.

Authorizer Service Layou

And when you put the two requirements together, we decided to try to build a minimal structure, infrastructure, within those physical data centers, composed of a small set of highly-available services; they are redundant in the data center, and we have multiple live data centers accepting transaction requests.

In addition to being minimal, the other important part is that it is isolated, we can authorize transactions that are coming into the payments network, only with this small set of services running on our physical data centers. We don't need any cloud communication in order to authorize the transaction, that was important. We were worried that if one of the links goes down, or there's a problem with Amazon, this would affect availability. And the, the other thing to note here is that we are using Thrift and Finagle for inter-service communication.

Kafka as the Bridge between Environments

So I was making the point about how isolated the inner loop for transactional authorization was, there are so many communications required. The ultimate truth about data for all customers lies on all of our cloud services. They maintain the databases that are the ultimate owners of that information. And so there's -- they need to know when the transaction was authorized. At the same time, there was information that we only learn about on the cloud side. For instance, if a customer makes a payment and that entailed a change in the credit limit, we need to send the authorization back to the authorizer. We are using Kafka for that, that was mentioned earlier, that is the default communication technology.

Kafka-Based Log/Snapshot

And the cool thing about using Kafka for this project is that we are not using it only for communication. One interesting property that Kafka has that other message brokers usually don't, is that messages are durable; a message does not disappear immediately after it is consumed. We are using that to build a log snapshot style data platform on the authorizer project. We did that because we didn't want to have a database running on the physical infrastructure as part of those requirements of being minimal and being highly available.

And so, we are starting everything in memory. The data for a credit card transaction is not very extensive, it can be stored in memory in the altogether -- authorizer services. And to ensure that every authorizer replica is consistent, we piggy-back on Kafka's log. The Amazon is running a message, and it is consumed by our infrastructure. And at the same time, a second service is consuming the messages. This snap shot or services accumulate the memory state and it generates a snap shot to disk and to S3.

And when a new authorizer instance starts, let's say we are deploying a new version, that authorizer instance fetches the snapshot from disk, loads it into memory, and now it can accept new transaction authorization requests, and it fetches from the snapshot on the Kafka log where the message was generated so we don't lose any data here.

Dramatic Improvements in Reliability and Fraud

And these are the metrics that we keep track of- you can see how it improved the standing rates, that's the rates -- that's the percentage of transactions we cannot authorize, the brand does so on our behalf, we are going to minimize that, and after we deploy our own authorizer, we will go around one percent of standing to below one basis point.

And what I want to convey to you is a point about the process. With this project, we decided that we wanted to go live as soon as possible to pass the MasterCard tests as soon as possible so we can learn from real data using controlled experiments and a set of test cards, and learn what actually was coming into the wire in the real life. Because, we knew that the documentation could not be relied on for complete implementation. That was very helpful for us, after those test cards, we rolled out larger and larger sets of cards until we were able to move all of our customer base to the authorizer. And next, Ed is going to talk about accounting.

Double Entry Accounting: Business Logic Depends on Data across Many Service

Edward Wible: Yes, doing the wonderful thing about touching everybody accounting this morning, which I'm sure you are excited about. I will try to do this efficiently.

So a problem that we face is that, in the example that was given on authorization, in order to say yes or no to a given purchase, you don't need a lot of data, but the data you need depends on a lot of other data. We need to know if you have an available limit, that is a cumulative function of everything we know about you: purchases, payments, credit card changes, all of that comes together to give us the answer of what you are open to buy, what is your available limit.

Double Entry: The Model

What we do to manage this complexity, we treat a lot of the core financial logic in the state full cumulative functions in a service we call double entry. So the model is pretty simple. It is an entry, it has an amount, and a pair of book or ledger accounts, the credit account and the debit account, and the balance at any given time is a cumulative function over the debits and credits for the book account, the cumulative sum. And the customer's balance sheet is just the collective picture of all their book accounts.

Double Entry: The Rulebook

And so, what we've done that is different from what most people do is model the double entry accounting system as an operational system, this is not an analytical system with finance, this is in real-time with customer, and we are relating business events, like a purchase or payment, to a series of entries that map to the domain model. This is a relationship between the business event and the debits and credits, and collectively it is a movement. This is a movement, you have two entries, the unsettled for the transactions that come in that have not settled and the current limit.

Double Entry: Challenges

This system is very powerful for us, it is the best way that we know to show you, the customers, what your balance sheet is. We have a limit bar in the app, and we need to know what's your limit, what's your current bill, etc. And so we use double entry for this. And there are some challenges. For one, ordering matters. So, as an example, if you're late and you make a payment for more than you owe, you don't have a negative late balance. You have a pre-paid balance. If the events happen in a different order, things are different. And so, to make this even worse, you have late-arriving events, you can make a payment that we receive that should be credited on Friday, we need to time travel back, re-play the events and figuring out the balance that does not invalidate the invariants we have, that is fixing invariants.

This is a property-based test that we use, and given that the interleaving of events creates a combining explosion, you cannot write-up unit tests that capture the entire space. We create a

randomized initial state with adjustments and purchases, etc., and then we create a loss event, and we generate thousands of examples of this and, every time we verify that our invariants, the properties, still hold. This allowed us to catch very real bugs in production.

And the last problem here is something that will be discussed next, right throughput, this is where we hit the limits of where we write to a single database, and entry is highest pressure database, because most businesses generate debits and credits for the banks. You have writes on the architecture causing writes on the entry system. The next item is infrastructure, and we will discuss how we dealt with that problem, among others.

Sharded, Fault Tolerant Infrastructure: Scaling Bottlenecks

Rafael Ferreira: And we began this talk by showing this curve, this is the customer growth curve. We are happy with that, it is good to have customers, a market number of customers. And, as engineers, we need to look at that with skepticism. And, with scale comes scalability problems. And, of course, we are no exception. The first couple of bottlenecks that started to affect us initially were the database throughput problem that Edward was talking about, we are using Datomic, it is a great database, but when we reach high write throughput levels, we reach certain limits, we have to throttle message conception that led to the writes in order to maintain service stability initially.

And another bottleneck that reared up its head in our infrastructure was the batch jobs. Some batch jobs that used to take just a few minutes were now taking many, many hours, even more than one day. And that, as you can imagine, can have a pretty serious impact for our customers.

Scaling Plan

And so, the first thing to do when you are at scale, you try to optimize. And we tried to optimize a few of our core flows, so they are faster. But that has a limit. At some point, you need to find a way to partition your workload so you can safely handle that workload in parallel and in an isolated manner. And we had to do that, and one interesting property of our domain that was helpful in that is that the interactions between customers are minimal. That is different from other domains; for instance, we look at social networks. In our case, most of the data and the business logic that we run pertains to customers at a time. And so we could use a partitioning of our customer base as a proxy for partitioning the workload.

Option #1: Partition Service Databases

And the first option that we considered when we were thinking about how to partition our workloads to scale is to partition at the database level. So here, you have a beck and service, and it is to write to a single database, and it reached certain limits, we would then partition the database. You have database charts, and that service on every write and on every query would need to route that query to the correct database. And that cannot work, a lot of companies do that, but we saw some problems with this approach. One is that it takes an enormous effort to go through every service that was facing scalability problems, and update every single query, update every single transaction, and the quality of the code base, after we changed all services to route, to multiple databases, might deteriorate a bit.

But that's not the main problem. The main problem, in our eyes, is that when you partition the database, you are addressing scalability problems with the database. There can be other scalability problems, we can have a long-running batch jobs, we can have the operational

overhead of scaling our Kafka clusters, there's a lot of problems that this option would not address.

Option #2: Scalability Units

So we considered a different model, the scalability unit model. When you are doing scalability units, your shards are not database shards, they are actually copies of your infrastructure. So let's look at Nubank's infrastructure. We have 120-plus services living on a cloud, their databases, Kafka clusters, Zookeeper clusters, networking. So when we go to the scalability units model, we build clones of the infrastructure and we assign different partitions of our customer base to different clones, those are our shards.

And that's what we ended up doing, and it didn't really help. We cannot rest multiple scalability bottlenecks at the same time. But, of course, some routing has to happen. When an external event happens, we need to be able to route those external events to the right chart.

So on the transaction authorization use case that we are just talking about, a customer makes a purchase. This is run on our physical infrastructure. Our cloud has to know about the purchase, and it has to be routed to the right shard. So it builds a section of infrastructure, we called it global, and the global section is not sharded, and the services that it leaves on the global section, they are done mostly to maintain mappings from externally-known identifiers to customer IDs and their shards. So when a purchase comes in, we look at certain identifying information, find the shard, and route it there. The same thing happens when we know about deposits, payments, and other events.

And there's a -- there's a slightly different use case for interactive use. When our customers are interacting with the app to talk to our services, we are able to route that kind of flow only at the very beginning when the customer logs in. And the way this happens, you are looking at here, at customer logging in. There's a global service, we will validate the customer's credentials to say that the person is who he or she says he or she is, and then it will respond to the app with hyperlinks, with hypermedia links, where the app will fetch more information. And so here, the app -- it will fetch information about the account from a different service. And this next interaction is already directly connected to the shard where that customer lives.

And a new -- new links will further the communication between the apps and the services and it will funnel to a series of hyperlinks, and then we can ensure that the app talks to the correct shard at all times, and we don't need to build an enormous routing layer that would intercept every single request from the outside. We are able to do that only at the beginning, at that log-in, which is helpful, of course, for managing our load.

Scaling Lessons Learned

And some -- this is going to some lessons learned from this process. The first lesson- it is maybe obvious- it is working for us and we are much less concerned about the impact of the growth of our customer base on our production services, or production quality. But there are some caveats. One is that this -- if you go to the scalability and its route, versus sharding each database route, it is harder to roll out incrementally. You basically have to be able to route, know how to route, every single externally-coming event to the right shard before you are able to send even a single customer to a second chard. This means that it is a long project that only bears fruit at the end, and it is important to pay attention to how your customer base is growing while that project is under way.

Fault Tolerance Patterns

And another point regarding infrastructure I want to make here is that even if you have a solution in place to deal with scale, problems happen. Services might try to interact with third party providers that are flying, services might run out of memory, a bug might be introduced and services will start running exceptions. We apply a couple of highly interesting tolerance patters to address those problems.

The first I will talk about is dead letters, a simple pattern that produces enormous gains for operations. A message is produced, it is consumed, and at which point an exception happens. And the libraries for the message researcher, we forward the original message along with extra metadata about the error to that letter topic. And another service consumes from the dead letter's topic and stores it in the database somewhere. And then, offline, an engineer will triage those errors, of those letters, to fire that service; we called it mortician for obvious reasons. If the engineer is confident that the bug was fixed and the problem healed, he can republish that message back to the original topic. It helps, if you are a financial institution, data loss is the worst thing that can happen. So we can recover from possible and eventual data loss using dead letters.

And another pattern I will talk about- circuit breakers, it is a common pattern if you are doing microservices. The interesting thing is how a circuit breaker interacts with messaging. And so here, we are consuming messages, and an out bound call fails. If that happens enough times, a circuit breaker will trip. What you do then is we pause the consumer, we stop consuming messages. And that allows what could be a barrage of exceptions of events happening to become simply in a lag that accumulates on a Kafka topic, when the problem is fixed, we start consuming it again. All right.

ETL + The Analytical Environment

Edward Wible: What did not scale well was using Datomic and our operational transactional infrastructure for aggregations, even for simple things, like how many customers we have. Our CEO asked this many times and it was increasingly difficult to answer. It is a simple question, what is going on? And even, with sharding and fragmenting the operational systems so that they scale better, you are making analysis harder and you are making aggregates and analytical work harder. So what we did is the traditional approach of making an ETL, which stands for extract, Transform, and Load.

Datomic Primer

And before we get to the ETL, I will give a primer on how Datomic works. It basically works mapping business events to get commit-style transactions in the database; so if we think about a customer joining Nubank and getting a credit limit, making a purchase, getting a credit limit increase, and blocking a card. It is a common sequence of events for our business. This is how they look in a Datomic database. So you have assertions of facts, this is a fact, this is a fact. And sometimes, like, in the case of the limit changing, you have something that used to be a fact that is no longer a fact; it is no longer 3,000, it is 5,000, that's the limit. And on the right side, you have a monotomically increasing transaction because the transactions are a first-class citizen, and you can use that to deal with log tailing.

Extract, Transform, Load

So for the ETL, we have an always on log tailor that tails the logs we have, and pumps them into the data lake, doing chunking and format conversion and things like that. And it is the same thing that we do for certain Kafka topics where the data is relevant and it is not in Datomic; similar log abstraction, we consume the topics and put them into the data lake. And we can paper over sharding; an analyst never wants to think about where the data is, shard one, two, or three. We put that back together into a contract. A contract is basically reconstructing a single logical table for an entity. And then we build, as functions of contracts and other data sets, we build a directed asynchronous graph for machine learning models and policies that translate the scores into business decisions. One example is putting a risk score to a credit limit for you, to do proactive credit limit increases.

ETL Example: Contribution Margin

And this is an example from our business where it is very valuable to have the analytical environment. The green line is the revenue, that comes from double entry, the operational system we talked about. And the purple line is from our ERP, that is cost. That's not a real time system and it is never going to be in production in a real-time environment, but in the analytical environment, we can combine those things to get the answer of, are we making money? Our business is complex, it is not clear if we are making money on every customer all the time. So having every tool you can to understand contribution margin per customer per day actually makes analytics much easier.

And so this is also the environment where we generate reports for regulators to keep them informed and happy, this is something that makes the business complex, but it does not hurt us because we have good tooling to make that happen. It is a great place to run machine learning models without having all of the constraints of production.

Real-Time Transfers

And last, but not least, I wanted to talk about something really recent that we launched- real-time transfers. And these are screenshots of an app, so the concept of the product is that you scan someone's QR code, or you get the bank account, you put in the money to transfer and the money is there in real-time. It is simple. The way it works, the transfer request comes in but, as I said, you are earning 7 percent per year, this money is invested. So what we do, instead of the traditional -- there is no database transaction here, because people are in different shards. So any form of in the database transactional semantics are not going to work. And we need to liquidate your investment and make sure that you have the money and it is ready to transfer, and so we do that and only then we initiate a transfer out. And if this transfer out for whatever reason doesn't work, we do a compensating transaction, re-invest the money and roll it back, because it is a very rare event and we wanted to optimize for the most scalable way to run this. So we are real-time gross settlement of transfers here.

And, at that point, we have a global service that consumes from every shard to get all of the global transfers into a single stateful service, so we can maintain idempotence. And at the same time, the double entry service is observing the events and updating the debits and credits, so we have an up-to-date an -- analytical view. And if it is to the right customer, we will update the shard and complete the transaction.

And with Brazil, if it is to an external bank account in the country, that is also fine and in real-time. We can translate Kafka messages into a soap approach, and back and forth, and get that

into a centralized hub-and-spoke payments model for Brazil that is connected to other Brazilian banks that we did not have to integrate with, one by one, thankfully.

Brazilian Payment System

And so I find this really interesting, kind of as an American, coming from the ACH system. This is pretty much space-age technology. And, it is kind of born out of a legacy of high inflation rates in a Brazil. People didn't want to wait until the end of the month to, kind of, build up counter-party risk, and then that settles, even if it is easier on the systems, the value is on real-time systems. And so around $100 billion is transferred per day, and every connected has IBM Q series hardware, it is very sophisticated and you can read about it at this link.

Domain Summary Model

And I think the overall big-picture summary is that the financial domain is large, there's a lot of pieces you need to have an MVP, a minimum viable product when you are dealing with people's money tends to not look minimal, it is large and there's a lot of interactions between the bounded context. You put something in charge back, and somehow that affects whether you are authorizing a purchase or not. And these things get highly coupled when placed inside a main frame, and when you allow the database intersections and storage procedures to couple them together. We had to work hard to decouple them and we use Kafka and asynchronous messaging as the life blood under that.

We are hiring in Sao Paulo and Berlin, and thank you, we will open it up for questions.

Can I run the mic around? Right up front.

When you talk about the authorizing components, when you pull up the messages from the Kafka streams, how do you respond to the client that this is the coder? Because Kafka is just one way, right?

For our authorizer services, they can produce messages to Kafka. Do you know if this one is HTTP or Kafka when a new transaction comes?

Kafka.

For us, we use Kafka to synchronize state into the authorizer so you can buy in real-time, and then we push it back into the cloud and also back out, importantly, into the other authorizer instances so that they all get an updated version of your open to buy. Otherwise, you risk, you know, speed attacks and things where, if something is routed to a stale authorizer, you can spend the same money twice. That's the critical, the double-spend problem is the critical thing we have to avoid.

[Editor's note: speaker far from mic].

Yeah, we use -- so Kafka has a synchronous semantics, and it basically operates in real-time for us. We think about it as synchronous, we don't think of it as something that is going to take a long time, we think about it in real-time, even though it is asynchronous, and it becomes synchronous when lags build up in response to some kind of issue.

And we actually monitor Q lags and we ensure that, in regular operation, most of the time, all Qs have a zero lag.

Um, first of all, I think that the architecture probably, like the reason is for so many decoupled parts. So using Kafka, when you are doing any kind of aggregations in Kafka, or because -- I think that Kafka is an interesting choice. You have a chance of duplications, right? And, I mean, the -- we talk about exactly the semantics, and at the end of the day, it is really hard, with so many events messages coming in.

And so, how do you do that? Maybe a little bit on that. And I mean, with so many moving parts, and so many decoupled parts, how difficult was it in general to, I think, what was the road to that complete architecture, and how difficult was it not to just do it, but convince everyone around it that it is going to work?

That's a great question. So we don't rely on Kafka exactly on semantics for our business, we make sure that any message that goes on Kafka has to be idempotent because you can consume the same message more than once, we use transactioning messages to maintain that. When you consume a message, we have a correlate ID to know if you have seen that, and when that goes into the database, we use transaction functions so we don't have something that is double-counted, even in the presence of multiple messages. We use acid transactions for that, we don't use the stateful Kafka streams.

Is it real-time, or do you do a lambda architecture?

It is real-time for everything in the architecture; we do lambda operation for the machine learning models, we run an expensive model over night to precompute something and, for a credit limit increase, we can respond in real-time, even if it was in the batch last night. We do a little bit of merging but not much. And how do we end up with this decoupled models, are we smart? No, we built a massive service, this is infamous in Nubank. We spent months splitting and building that. Why not put them in the same database? It is very tempting. We had to split services, with Datomic, it is not very fun. You do log replays, and it is a tough thing. And, you know, that said, it is very clear that we need to get to massive scale or we will not be a successful commercial retail bank. So it is easy to justify those sorts of investments when you have a long-term vision for where you need to get to, cutting the corner does not make sense in that context.

I have a couple of questions. And so, the first one would be, choosing somehow an extraordinary stack. Would you have some hind sights about that, or what would you change if you would have an opportunity to change it, or so would you change anything about the architecture?

Do you want to take that one?

Yeah, I have one big thing I would like to change, if I could, if I could have thought of it in the beginning is that, one interesting feature of Datomic that we have not mentioned yet is that transactions in Datomic are a first-class concept, and we attach that with the Git version of the service, we attach the credentials of the user that is making that transaction, and several other things. And one thing that we do not add and I would have liked to add, a customer identifier. Because, if every transaction had an identifier that could point to the actual customer that owns that data, things like splitting databases for sharding would have been much, much easier. That's a small detail, but something I would like to get better in the beginning.

And I think we also made some mistakes with how we used Datomic, it is really good for high-value business data. It doesn't work that well for fire hose writes, or long strings, or things where

you want to use different sorts of data stores. So we use Datomic as the default database for everything, and you are lulled into the false sense of security, I will start a service with Datomic, when the answer should be S3 or something slightly more used. No regrets. And something I would do differently is think about event sourcing a little bit earlier. We have it down streamed from Datomic, but not up streamed. We don't see every request that happens before the database write and the events are effectively lost. That is something I would have thought harder about and taught Kafka as a persistent thing, rather than an ephemeral message queue.

So debugging hundreds of microservices sounds like a lot of fun, especially when you have customers complaining they didn't receive any money. How are you attacking that problem, you are using a distributed tracer?

Yeah, it is surprising how non-patient people can be when they don't see their money. It is not a very forgiving domain. We don't use anything like Zipkin yet, distributed tracing is interesting. But we use Splunk. We use Splunk, and we use a correlation ID, where every service appends a segment, behaving like a tree. So for every request, you can figure out the fan of events that happened on HTTP and Kafka in the system. Does that make it easy? No. But it it is traceable, you can establish what happened and why.

Yeah, a related comment is that we use that feature of pinning metadata to the Datomic transactions as well, so we correlate the log aggregation data with the actual database, and what happened in the database at that moment. So it is much easier to debug than if you have a system that can lose data where you don't know the history of what happened. So this has been very helpful for us.