# Case Study of *Blockchain.com*, a Fast-growing Cryptocurrency Company

Nicolás D'Cotta

***Abstract*—Blockchain is a crypto finance house that scaled fivefold in employees in less than a couple of years. This article looks into the challenges it faced scaling up human and technical resources (mostly drawing from personal experience) and reflects on what worked well and what did not.**

## I. Blockchain's Size, Business Model and Stack

I have worked at Blockchain for little under 2 years and during that time I was able to observe how the company went from about **110 employees to more than 500** at the time of writing (with many of the new joiners being engineers), transformed into a **remote-first organisation** (largely because of the coronavirus), grew an **institutional business**, and **tripled its existing retail business**.

The company started in 2011 as a free simple blockchain explorer [1] that later begun offering non-custodial online Bitcoin wallets. When I joined, it had around 40 million active wallets in several cryptocurrencies, a brokerage service (to buy and sell crypto to users), as well as a recent currency exchange (where users trade with each other, not unlike Coinbase [2]).

This is implemented with a coarse-grained microservices architecture, with some services being much larger than others.

## II. Scaling Up Systems

We call the larger services *core services*, and they are further split into modules.

This results in a hybrid of microservices with a couple monoliths. When adding a new feature, depending on its scope and the data it needs, it is either built into a microservice or simply inside an existing core service. If, further down the line, it is decided it needs to grow further, it can get moved out of a core service into a new microservice.

Zero-Downtime deployability
Organizational autonomy

## III. Scaling Up People

### A. Microservices to Restructure Teams

Moving a module inside a core service out to its own microservice can happen as the team that owns the monolith grows and needs to be partitioned. This is the case when a feature becomes large enough to deserve its own engineer owners.

This happened with a 'crypto brokerage' service which got split off from the retail core service. Aside from partitioning the retail team in order to from a new brokerage team, the other goal of this migration was to achieve more fine-grained deployability (brokerage was getting releases more often than the core, and we wanted to be able to roll back one without rolling back the other – this allowed the two teams to not step on each other's toes). This shows Blockchain trying to leverage Conway's Law [3].

### B. Ubiquitous Language

Another paradigm that facilitated Blockchian's fast growth was the idea of having a *Ubiquitous Language* (or *UL*). The goal is to have some form of documentation that describes the functioning of a 'flow' (like the process for withdrawing crypto from the exchange).

These are represented as Finite State Machine (FSM) diagrams, designed to have well-named states and state transitions depending on each interaction with the user, other microservices, and external services. Forcing some backend teams to think of flows as Finite State Machines allows implementing them as actors which go through state transitions. Events that trigger these transitions are just messages passed around between actors.

The ultimate purpose of introducing UL was to be able to transmit and persist knowledge effectively, because Blockchain had what we called a strong 'bus factor' [4] – meaning that if a few, knowledgeable team members got hit by a bus, then the company would be in serious trouble. The reason for this was probably a mix of high rotation of employees due to varying competitive salaries in the crypto industry (so team members that remained became senior very quickly) and complex critical legacy code that only some understood.

The idea of Ubiquitous Language was successful, to some extent. Drawing Finite State Machine diagrams do make it very easy to communicate a flow or a feature while accounting for every single edge case and interaction. But having to design a state machine can be tedious for simple cases, and for complex ones it can be too time-consuming (some FSMs have 'sub-FSMs' when they are too large to put in a single document!).

Additionally, most developers are very used to classical object-oriented programming and some struggle to write in an actor model at first – it is also harder to find new hires that know the paradigm well. Furthermore, having to represent state transitions for every single side effect and call (and the states resulting from their failures) require large amounts fo boilerplate – code that traditionally would have been a simple *try-catch* in Kotlin or Java.

## IV. Growing in a Financial Regulated Environment

As a company operating in the finance industry, which is heavily regulated, and dealing with large monetary amounts; consistency, reliability, and accountability are at the top of Blockchain's priorities.

Adopting FSMs means we write actor-oriented code. This makes reasoning about capturing events easy (an event is simply a message!) and allows to implementing Event Sourcing [5] on critical monetary services (like the ledger).

Event Sourcing allows resiliency and consistency without having to resort to transactionality or retries: messages are persisted and replayed on service restarts or crashes. It also enables a very transparent audit trail of everything happening inside services – one can just look at the persisted events.

While in theory Event Sourcing provides many benefits, in practice it proved very costly to implement. Very critical old code had to be re-written in an actor-oriented fashion, persisting messages adds runtime overhead, replaying them makes service startups much slower, and serialisation has to be implemented for events. Serialisation in particular involves a lot of boilerplate code for each message class.

To add to the complexity, modifying a state machine often involves changing the messages actors pass around. This means that then we must deal with different serialisation versions and worry about backwards compatibility between them.

## References


[1] Blockchain. "About." (Feb. 2022), [Online]. Available: https://www.blockchain.com/about.

[2] Coinbase. "Coinbase Pro Exchange." (Feb. 2022), [Online]. Available: https://pro.coinbase.com/ (visited on 02/12/2022).

[3] M. E. Conway. "How do committees invent." (Apr. 1968), [Online]. Available: http://www.melconway.com/Home/Committees_Paper.html (visited on 02/12/2022).

[4] M. Bowler, "Truck factor," in *Agile Advice*. May 15, 2005. [Online]. Available: https://www.agileadvice.com/2005/05/15/agilemanagement/truck-factor/ (visited on 02/12/2022).

[5] M. Fowler. "Event sourcing." (Dec. 12, 2005), [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html (visited on 02/12/2022).