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.

A. Dealing With Large Services

Something the team noticed about the core services around 2019 is that when they were large enough, compilation and testing times were no longer negligible: from a commit in a feature branch being pushed, it took 50 minutes for the retail core service to compile and test (after which the branch could be merged into `master`). After the merge, CI took another 50 minutes to test and build again (the new merge commit may not be building the exact same artifact as the previous test run) to produce the docker image that could then be deployed.

Developers also need to get two approving reviews in order to merge, so overall we would expect around two hours from the moment we pushed a commit to the moment we were able to deploy it. This is, of course, assuming the commit was good straight away: if the CI runs reported test failures, at least another 50m had to be accounted for. The causes of the problem were mainly large codebases in Kotlin (which did not compile very fast in version 1.3) and the fact that code generation was involved in the building process.

It was in order to mitigate this that modules were introduced in the core services, making them as small as possible. The modules are dynamically linked (due to the nature of the JVM) and so compilation artifacts can now be cached for each module in CI persistent storage, which is hosted by a cloud provider and scaled dynamically with the number of concurrent builds.

Therefore, when modifying a module, developers only need to wait for that module's compilation, its tests, and the integration tests – binaries and test results from other modules are reused. Additionally, developers' build tools (in this case, gradle) are given remote read access to the build cache, so their local compilation times also massively improved.

After the modularisation of the core services and the implementation of the build cache, builds improved dramatically (most commonly down to 10 minutes per build) both locally and on CI, and core services became much easier to break up. This shows Blockchain leveraging cloud to scale tooling (not servers in production) that speeds up iteration – and therefore shortens the feedback loop for a feature.

B. Zero-downtime

Unlike more traditional markets and trading platforms, crypto markets are usually always open. Therefore, downtime translates to losses for the business, so zero-downtime deployability is very desirable.

While there is no company-wide policy, the platform team tries to have always 2 instances of every service running at all times. This allows for the following:

- **Blue/Green delpoyments** [3] where one instance of the service at a time is updated to the new version. Once an instance has been serving traffic for a short while, it is marked as 'healthy', and the next instance is updated. This means that even during a new release, there is always an instance serving traffic.
- **Higher Availability** – if an instance crashes, the load balancer can direct traffic to the other (hopefully still healthy) instance while the unhealthy one is restarted.
- **Stateless Services** – developers are aware that whatever new service they are writing, it will most likely be running two instances from the start. This encourages (but does not force) engineers to write stateless services, so they will

not have to bother with mechanisms like leader election. [1]

In practice, always running more than a single instance of a service from the start is a mostly successful policy, but there are some downsides.

Firstly, it is costly – plenty of microservices could cope with load on a single instance just fine, but still run two. This is mitigated with containers, as opposed to using a machine for each instance.

Secondly, some services cannot easily be made stateless (examples include polling services and the ledger), so we must choose between a single instance or several instances with leader election mechanisms. For the sake of less downtime, the latter is usually preferred; which means engineers need to reason about leadership when developing these services.

C. Even-Driven Services

`TODO do only if covered in course?`

For reasons described in III-B, lots of services are modeled after Finite State Machines. This encourages developers to think of their applications as things that 'wait for events' rather than things that 'are called', allowing for an *event-driven* design [5].

This provide some benefits, like *Event-Sourcing* (discussed in IV).

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 [6].

B. Ubiquitous Language

Another paradigm that facilitated Blockchain's fast growth was the idea of having a *Ubiquitous Language* (or *UL*) [7], [8]. 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). This from of communication should be understood by engineers as well as the product team.

Flows 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' [9] – 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.

While the idea of UL is nothing new ([7] is from 2004) I personally think Blockchain took an innovative approach in that they took it to the extreme by using FSMs and diagrams and exposing product to them, rather than simply conforming to *'a common [...] language between developers and users'* (as described by Martin Fowler in [8]).

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) requires 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 implementing Event Sourcing [10] 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,

[1] **Leader Election** is a mechanism where a cluster of servers agree on a 'leader' among them to run tasks that cannot be parallelized [4].

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.

V. In Retrospective

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] HashiCorp Learn. "Blue/green & canary deployments, Nomad." (2022), [Online]. Available: https://learn.hashicorp.com/tutorials/nomad/job-blue-green-and-canary-deployments (visited on 02/15/2022).

[4] H. Attiya and J. Welch, *Distributed computing: fundamentals, simulations, and advanced topics*. John Wiley & Sons, 2004, vol. 19.

[5] M. Fowler. "What do you mean by "event-driven"?" (Feb. 7, 2017), [Online]. Available: https://martinfowler.com/articles/201701-event-driven.html (visited on 02/15/2022).

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

[7] EricEvans, "Ubiquitous language," in *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004, ISBN: 978-0321125217.

[8] M. Fowler. "Uniquitous language." (Oct. 31, 2006), [Online]. Available: https://www.martinfowler.com/bliki/UbiquitousLanguage.html (visited on 02/16/2022).

[9] 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).

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