Cell-Based Architecture

Scaling the stock system to support over 30k RPS across all of LATAM

Luram Archanjo

Who I am

Luram Archanjo

Senior Software Architect @ Mercado Livre

MBA in Java Projects

Java and Microservices Enthusiast

Agenda

01 The problem

02 The solution

03 The outcome

04 Lessons learned

05 Q&A

Agenda

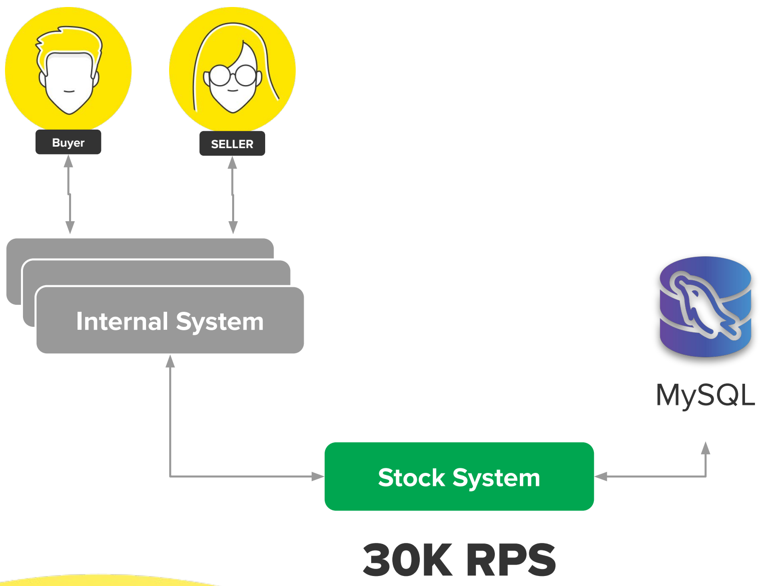
Introduction: Where we are!

Mercado Envíos operates across Latin America (LATAM), connecting millions of buyers and sellers every day in countries like Mexico, Brazil, Argentina, Chile, Uruguay, Peru and Colombia. Behind every purchase there is a complex network of fulfillment centers, cross-docks and last-mile partners that need accurate, real-time stock information to keep the promise to our customers.

All of that logistics operation relies on a single regional stock platform and one of the largest MySQL databases in the company, storing tens of terabytes of inventory data. It has successfully supported our growth for years.

FULL

Introduction: The Architecture

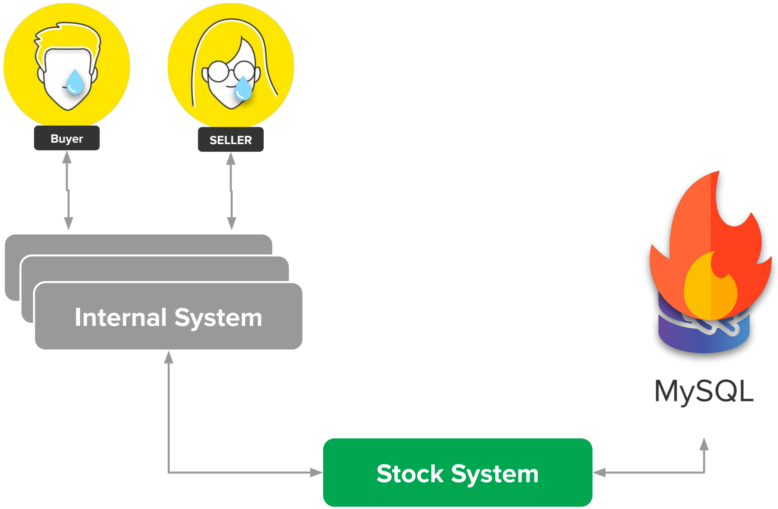


FULL

It's a Monolith: We need low latency, strong consistency and high availability

What **would be the impact** on your company if a single database failure stopped all operations in **LATAM**?

Introduction: The problem (everyday)



30K RPS

We hit our allocated resource limits and the service went down!

Let’s scale up

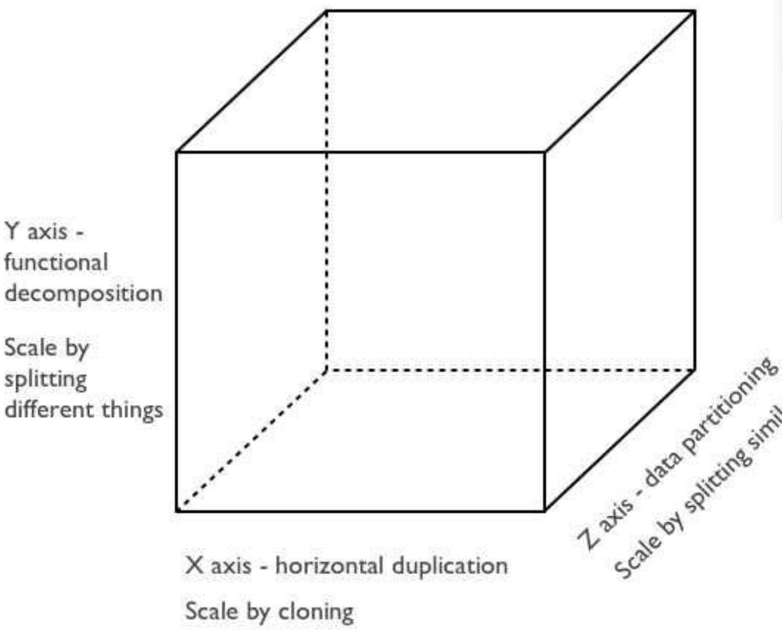
It was not possible!

There was literally no bigger instance to buy

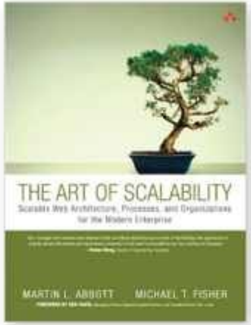
The Scale Cube

3 dimensions to scaling

Not possible



Y axis - functional decomposition
Scale by splitting different things
X axis - horizontal duplication
Scale by cloning
Z axis - data partitioning
Scale by splitting similar things



THE ART OF SCALABILITY

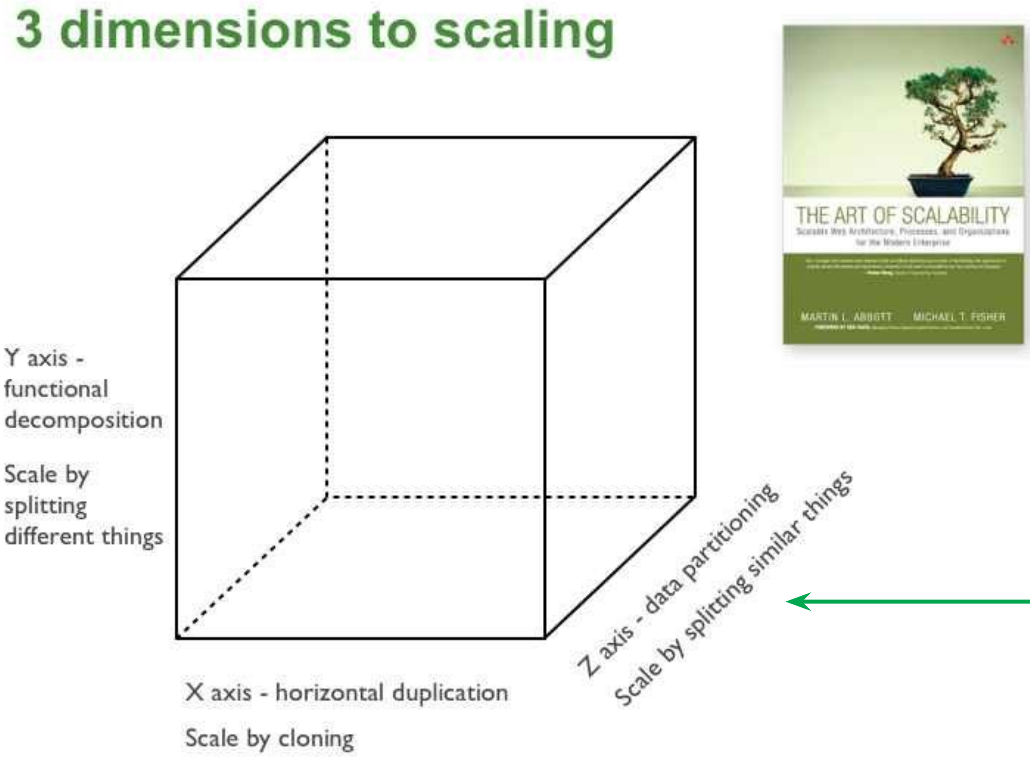
Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise

MARTIN L. ABBOTT MICHAEL T. FISHER

It's possible

What about the **Blast radius?**
If we go down, we will still impact all of LATAM

The Scale Cube


3 dimensions to scaling
Y axis - functional decomposition
Scale by splitting different things
X axis - horizontal duplication
Scale by cloning
Z axis - data partitioning
Scale by splitting similar things
THE ART OF SCALABILITY
Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise
MARTIN L. ABBOTT
MICHAEL T. FISHER

Not possible

Is it possible?

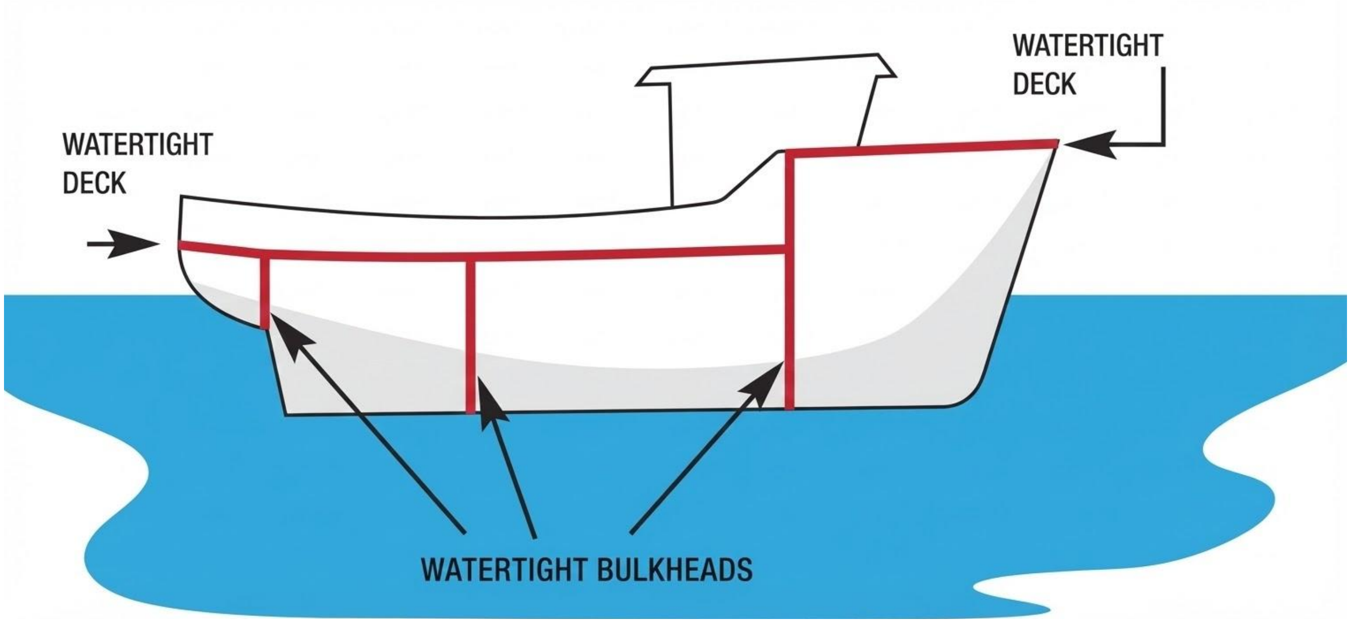
Not possible

Yes, it is, with

Cell Based Architecture

Bulkhead Architecture

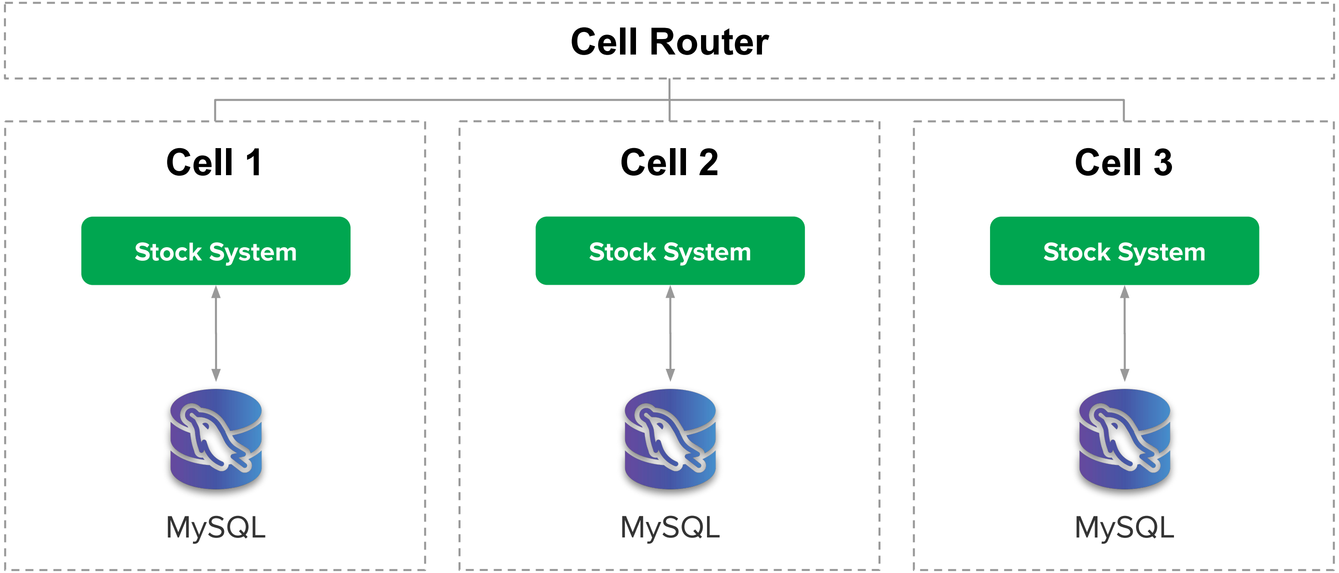
In a bulkhead architecture, elements of an application are isolated into pools so that if one fails, the others will continue to function. It's named after the sectioned partitions (bulkheads) of a ship's hull. If the hull of a ship is compromised, only the damaged section fills with water, which prevents the ship from sinking.



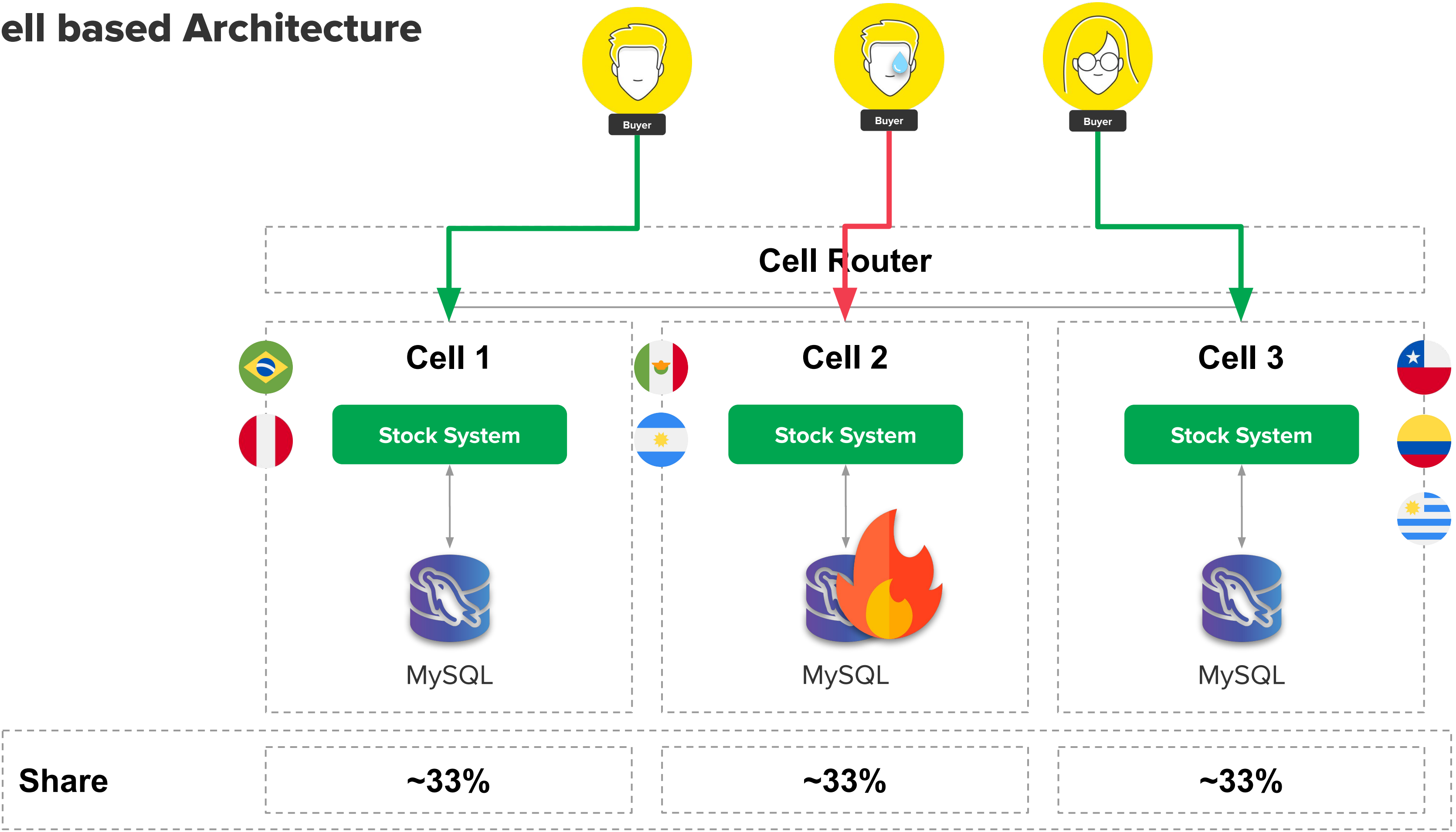
Cell based Architecture

Cell-based architecture is about splitting a large, critical system into a set of independent cells, each with its own compute, database and traffic slice.

- Each cell handles a subset of our traffic (for example, a group of countries) end-to-end.
- Cells are isolated: if one cell fails, the others keep working - we reduce the blast radius.
- We can scale and deploy each cell independently, based on its specific load and needs.



Cell based Architecture



OK, we have a plan!

Now, **let's migrate to it**

😎

Migration: Non-negotiable constraints

To migrate the system to a cell-based architecture, we had a few non-negotiable constraints:

- **Transparent** for clients and consumers
- **No visible downtime for clients** / no long maintenance window
- Safe, tested rollback plan
- Zero data loss

And these constraints turned into some hard problems we had to solve, for example:

- How can we move **tens of terabytes of data** with no long maintenance window?
- How can we **route each request to the right cell** if our existing APIs don't have a routing key?
- How can we **migrate existing records** and still keep a safe rollback path?

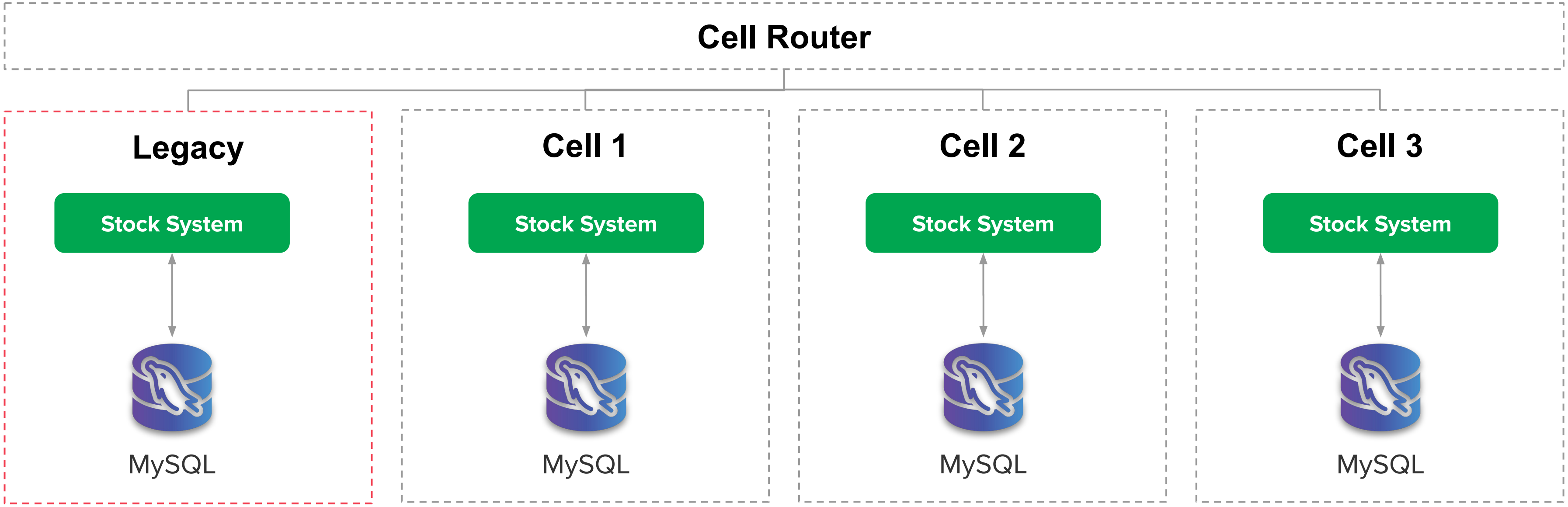
How can we **route each request to the right cell** if our existing APIs don’t have a routing key?

Migration: How can we route each request to the right cell if our existing APIs don’t have a routing key?

Many of our public APIs were **not designed for sharding**, they didn’t expose any field we could use as a routing key.

Changing all those contracts at once would mean a **huge, multi-team migration** and a lot of risk.

We introduced a **routing layer (cell router)** in front of the cells:

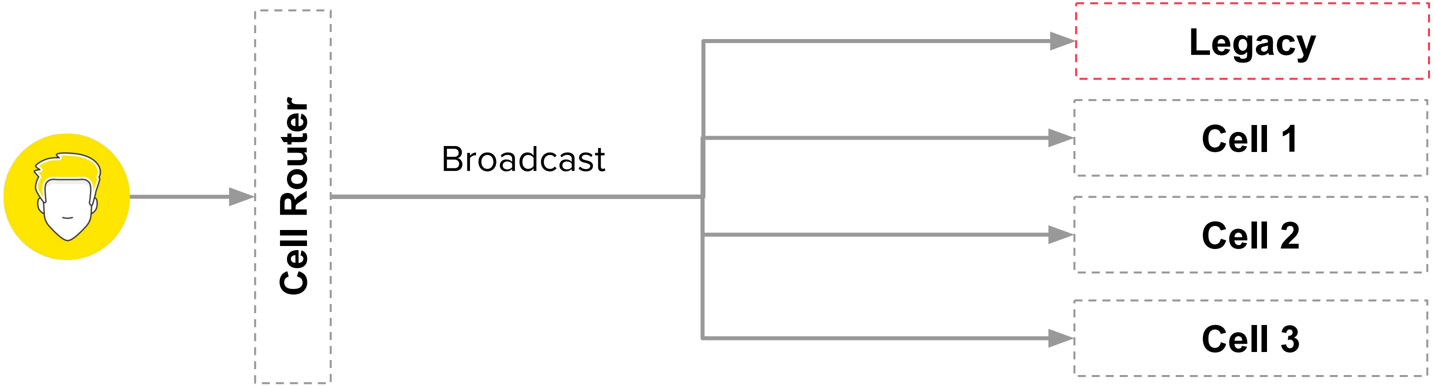


Migration: How can we route each request to the right cell if our existing APIs don’t have a routing key?

For one of our most-used APIs, there was **no way to know the cell** from the HTTP request:

- No routing key in the path, query string, headers or body
- The API was widely used, so changing the contract was very hard
- Clients should not need to know anything about cells

Because of that, our router initially had to use a **broadcast strategy**: it sent the request to all cells, and only the cell that owned the data replied successfully.



How can we move t**ens of terabytes of data** with no downtime?

Migration: How can we move tens of terabytes of data with no downtime?

The **bad news:** it was not possible to do it with zero downtime.

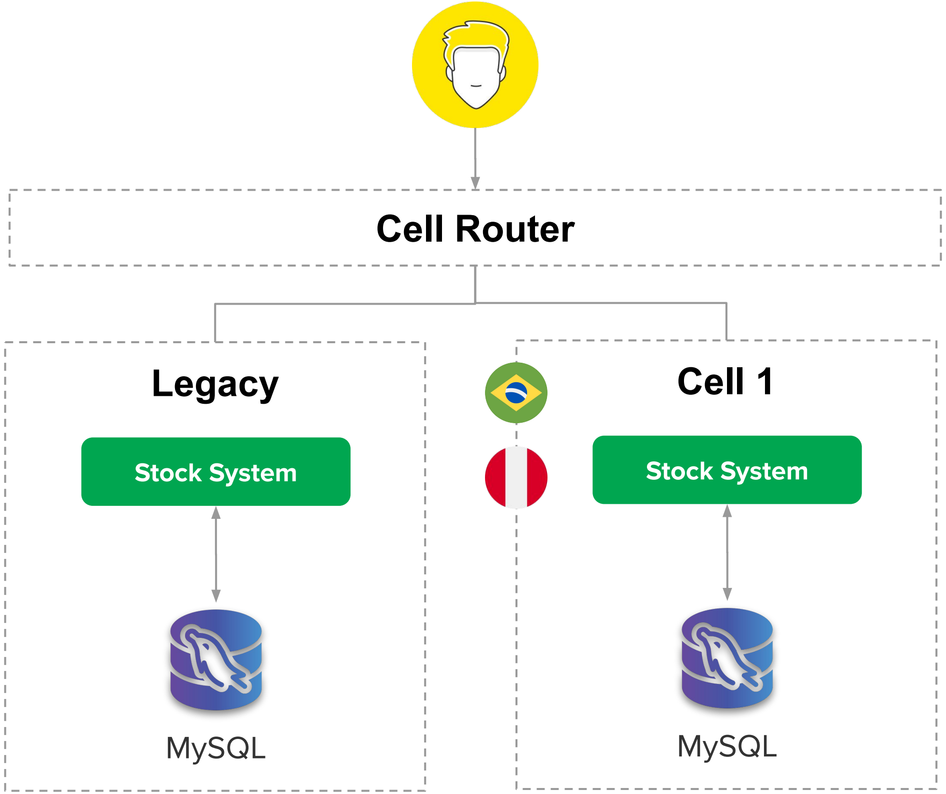
The **good news:** we minimized the write downtime to a very small, controlled window.

We had an almost **30 TB MySQL** database powering all stock operations in LATAM, and a naive migration would take many hours.

Instead, we combined data reduction with a short maintenance window, rather than doing a big-bang export.

First, we **reduced the source volume**: archived old data, cleaned up unused tables and indexes, and used the *OPTIMIZE TABLE table_name;*

Migration: How can we move tens of terabytes of data with no downtime?



1# Blocked all **write** operations to avoid data loss

2# Migrated the **remaining data** to the new databases

3# Ran validation and smoke tests

4# Re-enabled the **write** flow through the new cells

~20 minutes per cell

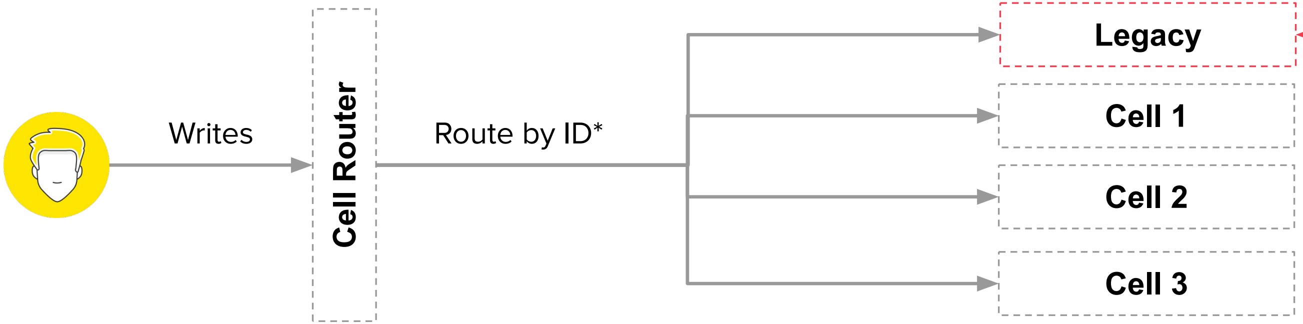
How can we **migrate existing records** and still keep a safe rollback path?

Migration: How can we migrate existing records and still keep a safe rollback path?

The **bad news:** a fully backward-compatible “dual-write + instant rollback” strategy was not feasible – the cost and complexity for this project were too high.

The **good news:** we minimized the risk and impact:

- For **existing records**, we couldn’t change IDs or API contracts – too many systems depended on them
- We kept the **legacy database as the source of truth** while we created and warmed up the cell databases
- For new records, each cell has its own **ID range**, so the ID itself becomes the routing key.
- Only after running with the cells in production for a while did we start to **deprecate the legacy path**

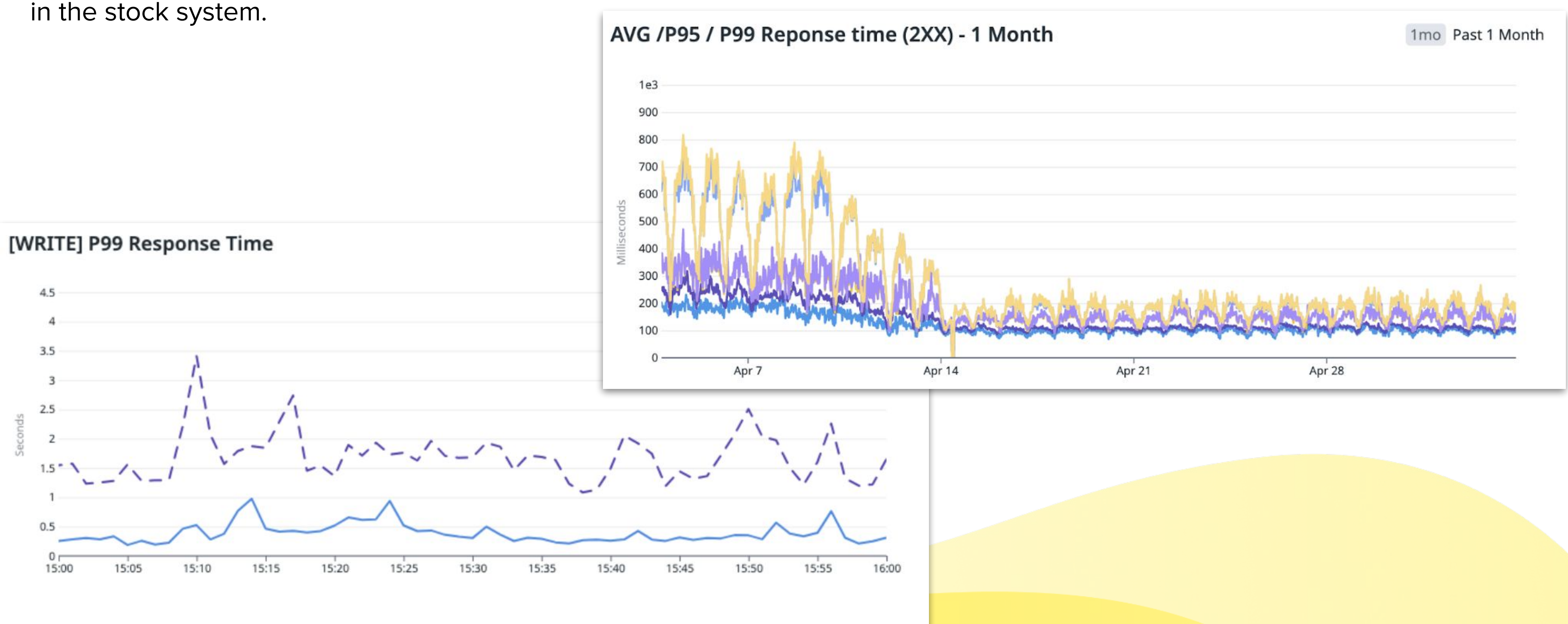


Ran for some days/months

The outcome

The outcome

We **cut p95/p99 read and write operations by about 70%** and went a **full year without a single infra-related** incident in the stock system.



AVG /P95 / P99 Reponse time (2XX) - 1 Month

[WRITE] P99 Response Time

Image 1: Response time decreases ~70% (--- before / — after)

Lessons learned: In short, scale isn’t a bigger box – it’s many smaller, safer ones.

- Scaling is not always about buying a bigger box, it’s about **redistributing responsibility**
- Always think about **blast radius**: what happens if this database or service goes down?
- A well-designed **"monolith" can still be the right choice** when you need low latency and strong consistency
- Design with a **routing key** from day one, it makes sharding and cells much easier later
- An Internal Developer Platform (Fury) turns complex patterns like cells into **reusable, operable templates** instead of one-off projects

Thanks & Questions

Luram Archanjo

/luram-archanjo