# Managing Data Chaos in The World of Microservices

Oleksii Kachaiev, @kachayev

## @me

- CTO at Attendify
- 6+ years with Clojure **in production** 🙀
- Creator of `Muse` (Clojure) & `Fn.py` (Python)
- `Aleph` & `Netty` contributor
- More: protocols, algebras, Haskell, Idris
- @kachayev on Twitter & Github

## The Landscape

- microservices are common nowadays
- mostly we talk about deployment, discovery, tracing
- rarely we talk about **protocols** and **errors handling**
- we almost **never talk** about data access 😒
- we almost never think about data access in advance

## The Landscape

- infrastructure questions are "generalizable"
- data is a pretty **peculiar phenomenon**
- number of use cases is way larger
- but we still can summarize something

## The Landscape

- service **SHOULD encapsulate** data access
- meaning, no direct access to DB, caches etc
- otherwise you have a **distributed monolith** 🙈
- ... and even more problems

## The Landscape

- data access/manipulation:
    - reads
    - writes
    - mixed transactions
- each one is a separate topic

## The Landscape

- reads
  - transactions (a.k.a "real-time", mostly API responses)
  - analysis (a.k.a "offline", mostly preprocessing)
- will talk mostly about **transaction reads**
- it's a complex topic with microservices 😱

## The Landscape

- early days: **monolith with a single storage** 😍
- (mostly) relational, (mostly) with SQL interface
- now: a **LOT** of services
  - backed by different storages
  - with different access protocols
  - with different **transactional semantic**

## Across Services...

- no "JOINS"
- no transactions
- no foreign keys
- no migrations
- no standard access protocol

## Across Services...

- ~~no~~ manual "JOINS"
- ~~no~~ manual transactions
- ~~no~~ manual foreign keys
- ~~no~~ manual migrations
- ~~no standard~~ manually crafted access protocol

## Across Services...

- "JOINS" turned to be a "glue code"
- transaction integrity is a problem, fighting with
  - dirty & non-repeatable reads
  - phantom reads
- no ideal solution for **references** integrity

## Use Case

- typical messanger application
- **users** (microservice "Users")
- chat **threads & messages** (service "Messages")
- now you need a list of unread messages with senders
- hmmm...

## JOINs: Monolith & "SQL" Storage

```
SELECT (
  m.id, m.text, m.created_at,
  u.email, u.first_name, u.last_name,
  u.photo->>'thumb_url' as photo_url
) FROM messages AS m
JOIN users AS u ON m.sender_id == u.id
WHERE m.status = UNREAD
  AND m.sent_by = :user_id
LIMIT 20
```

😍

## JOINs: Microservices

???

## JOINs: How?

- on the client side
- Falcor by Netflix
- not very popular apporach
- due to "almost" obvious problems
  - impl. complexity
  - "too much" of information on client

## JOINs: How?

- on the server side
- either put this as a new RPC to existing service
- or add new "proxy"-level functionality
- you still need to implement this...

which brings us...

## Glue Code

## Glue Code: Manual JOIN

```clojure
(defn inject-sender [{:keys [sender-id] :as message}]
  (d/chain'
   (fetch-user sender-id)
   (fn [user]
     (assoc message :sender user))))

(defn fetch-thread [thread-id]
  (d/chain'
   (fetch-last-messages thread-id 20)
   (fn [messages]
     (->> messages
          (map inject-sender)
          (apply d/zip')))))
```

🤓

## Glue Code: Manual JOIN

- it's kinda simple from the first observation
- we're all engineers, we know how to write code!
- it's super boring doing this each time
- your CI server is happy, but there're a lot of problems
- the key problem: it's messy
    - we're mixing nodes, relations, fetching etc

# Glue Code: Keep In Mind

- concurrency, scheduling
- requests deduplication
  - how many times will you fetch each user in the example?
- batches
- errors handling
- tracebility, debugability 😏

## Glue Code: Libraries

- Stitch (Scala, Twitter), 2014 (?)
- Haxl (Haskell, Facebook), 2014
- Clump (Scala, SoundCloud), 2014
- **Muse (Clojure, Attendify), 2015**
- Fetch (Scala, 47 Degrees), 2016
- ... a lot more

# Glue Code: How?

- declare data sources
- declare relations
- let the library & compiler **do the rest** of the job 🙌
    - data nodes traversal & dependencies walking
    - caching
    - parallelization

# Glue Code: Muse

```clojure
;; declare data nodes
(defrecord User [id]
  muse/DataSource
  (fetch [_] ...))

(defrecord ChatThread [id]
  muse/DataSource
  (fetch [_] (fetch-last-messages id 20)))

;; implement relations
(defn inject-sender [{:keys [sender-id] :as m}]
  (muse/fmap (partial assoc m :sender) (User. sender-id)))

(defn fetch-thread [thread-id]
  (muse/traverse inject-sender (ChatThread. thread-id)))
```

## Glue Code: How's Going?

- pros: less code & more predictability
  - separate nodes & relations
  - executor might be optimized as a library
- cons: requires a library to be adopted
- can we do more?
  - ... pair your glue code with access protocol!

## Glue Code: Being Smarter

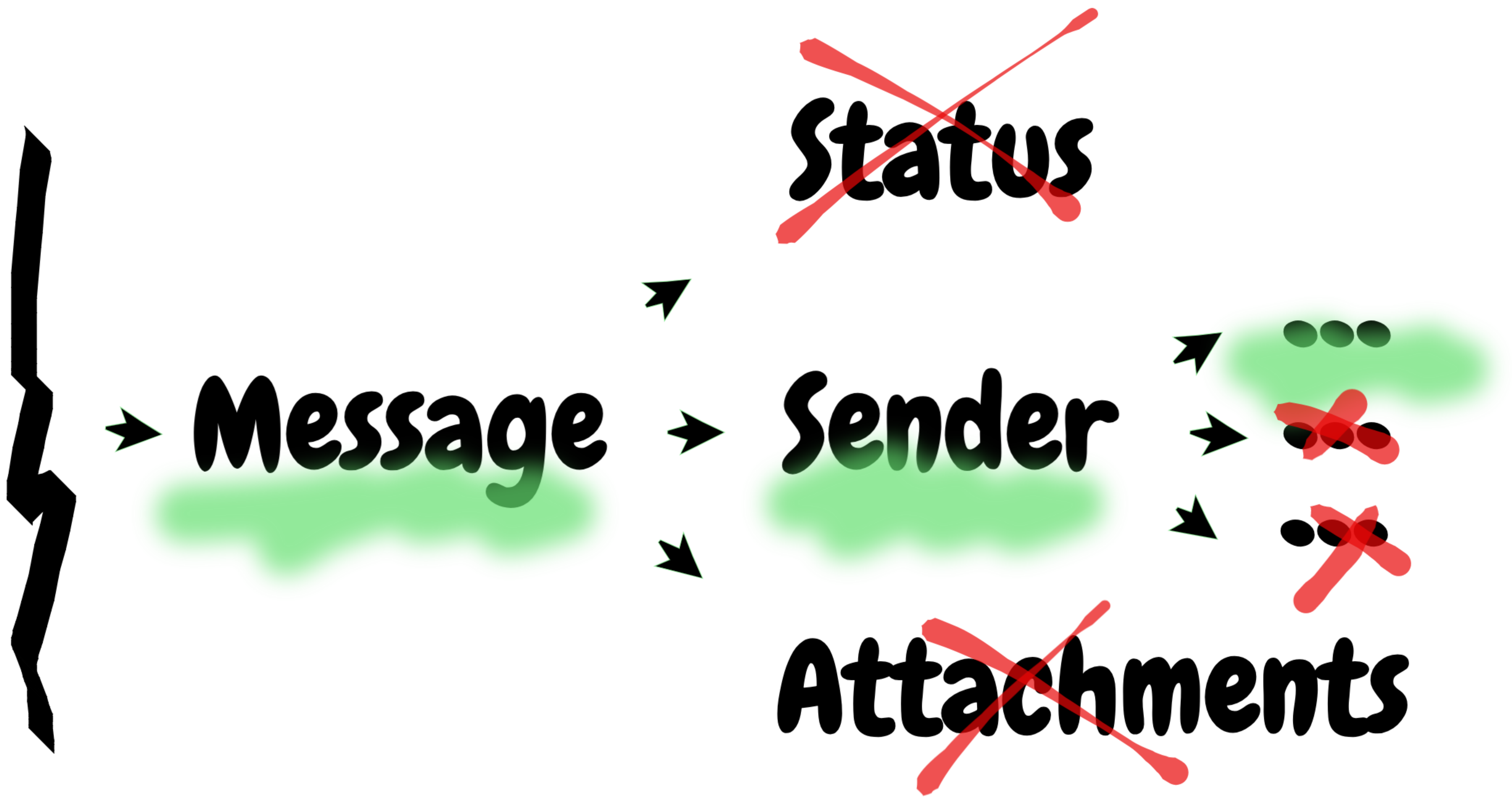
- take data nodes & relations declarations
- declare what part of the data graph we want to fetch
- make data nodes traversal **smart enough** to:
  - fetch only those relations we mentioned
  - include data fetch spec into subqueries

## Glue Code: Being Smarter

```
(defrecord ChatMessasge [id]
  DataSource
  (fetch [_]
    (d/chain'
     (fetch-message {:message-id id})
     (fn [{:keys [sender-id] :as message}]
             (assoc message
               :status (MessageDelivery. id)
               :sender (User. sender-id)
               :attachments (MessageAttachments. id))))))
```

## Glue Code: Being Smarter

```
(muse/run!! (pull (ChatMessage. "9V5x8slpS")))
;; ... everything!

(muse/run!! (pull (ChatMessage. "9V5x8slpS") [:text]))
;; {:text "Hello there!"}

(muse/run!! (pull (ChatMessage. "9V5x8slpS")
                  [:text {:sender [:firstName]}]))
;; {:text "Hello there!"
;;  :sender {:firstName "Shannon"}}
```


Message
~~Status~~
Sender
~~Attachments~~
•••
•••
•••

## Glue Code: Being Smarter

- no requirements for the downstream
- still pretty powerful
  - even though it doesn't cover 100% of use cases
- now we have query **analyzer**, query **planner** and query **executor**
  - I think we saw this before...

## Glue Code: A Few Notes

- things we don't have a perfect solution (yet?)...
- foreign keys are now managed manually
- read-level transaction guarantees are not "given" 😳
  - you have to expose them as a part of your API
  - at least through documentation

# Glue Code: Are We Good?

- `messages.fetchMessages`
- `messages.fetchMessagesWithSender`
- `messages.fetchMessagesWithoutSender`
- `messages.fetchWithSenderAndDeliveryStatus`
- 😕 🙁 ☹️
- did someone say "GraphQL"?

Protocol

Protocol?

**Protocol???**

## Protocol: GraphQL

- typical response nowadays
- the truth: it doesn't solve the problem
- it just shapes it in another form
- `GraphQL` vs `REST` is unfair comparison
    - `GraphQL` vs `SQL` is (no kidding!)

## Protocol: GraphQL

```
{
  messages(sentBy: $userId, status: "unread", lastest: 20) {
        id
        text
        createdAt
        sender {
            email
            firstName
            lastName
            photo {
                thumbUrl
            }
        }
    }
}
```

# Protocol: SQL

```sql
SELECT (
  m.id, m.text, m.created_at,
  u.email, u.first_name, u.last_name,
  u.photo->>'thumb_url' as photo_url
) FROM messages AS m
JOIN users AS u ON m.sender_id == u.id
WHERE m.status = UNREAD
  AND m.sent_by = :user_id
LIMIT 20
```

## Protocol: GraphQL, SQL

- **implicit** (GraphQL) VS **explicit** (SQL) JOINs
- **hidden** (GraphQL) VS **opaque** (SQL) underlying data structure
- **predefined** filters (GraphQL) VS **flexible** select rules (SQL)

# Protocol: GraphQL, SQL

- no silver bullet!
- `GraphQL` looks nicer for nested data
- `SQL` works better for `SELECT ... WHERE ...`
  - and `ORDER BY`, and `LIMIT` etc
- revealing how the data is structured is **not all bad**
  - ... gives you **predictability on performance**

## Protocol: What About SQL?

- you can use **SQL** as a client facing protocol
- seriously
- even if you're not a database
- why?
  - widely known
  - a lot of tools to leverage

## Protocol: How to SQL?

- Apache Calcite: define SQL engine
- Apache Avatica: run SQL server
- documentation is not perfect, look into examples
- impressive list of adopters
- do not trust "no sql" movement
    - use whatever works for you

## Protocol: How to SQL?

- working on a library on top of `Calcite`
  - hope it will be released next month
- to turn your service into a "table"
- so you can easily run SQL proxy to fetch your data
- hardest part:
  - how to convey what part of SQL is supported

## Protocol: More Protocols!

- a lot of interesting examples for inspiration
- e.g. Datomic datalog queries
- e.g. SPARQL (with data distribution in place 😊)
- ... and more!

## Migrations & Versions

## Versioning

- can I change this field "slightly"?
- this field is outdated, can I remove it?
- someone broke our API calls, I can't figure out who!

## Versioning

- sounds familiar, ah?
- API versioning * data versioning
- ... * # of your teams
- that's a lot!

## Versioning

- first step: describe everything
  - API calls
  - IO reads/writes… to files/cache/db
- second step: collect all declarations to a single place
  - no need to reinvent, **git repo** is a good start

## Versioning

- kinda obvious, but **hard to enforce** organizationally
- you don't need a "perfect solution ™"
- just start from something & **evolve** as it goes

# Versioning: Describe

- 2 specific problems/pitfalls
  - be as **precise** as you can
  - declare types **twice**

## Versioning: Refine Your Types!

- most of the time we primitives: `String`, `Float` etc
- .. and collections: `Maps`, `Arrays`, (very rarely) `Sets`
- that's not enough 😒
- came from memory management
    - doesn't work for bigger systems

## Versioning: Refine Your Types!

- you should be as precise as you can!
- type theory for the resque
- refined types in Haskell, Scala, Clojure
    - basic type + a predicate

# Versioning: Refine Your Types!

```
(def LatCoord (r/refined double (r/OpenClosedInterval -90.0 90.0)))

(def LngCoord (r/OpenClosedIntervalOf double -180.0 180.0))

(def GeoPoint {:lat LatCoord :lng LngCoord})

(def Route (r/BoundedListOf GeoPoint 2 50))

(def Route (r/refined [GeoPoint] (BoundedSize 2 50)))

(def RouteFromZurich (r/refined Route (r/First InZurich)))
```

## Versioning: Refine Your Types!

- precise types for all IO operations
- **runtime** check is **a decent start**
- **serialize** type definitions to file
  - make sure that's possible when picking a library
- you can also auto-convert storage metadata
  - `char (30)` → `(r/BoundedSizeStr 0 30)`

## Versioning: Type Twice

- **never rely** on a **single** point of view
- each request/response should be declared twice
    - by the service and the caller
- each data format (e.g. DB table)
    - by storage & by the reader
    - ... all readers

## Versioning: Type Twice

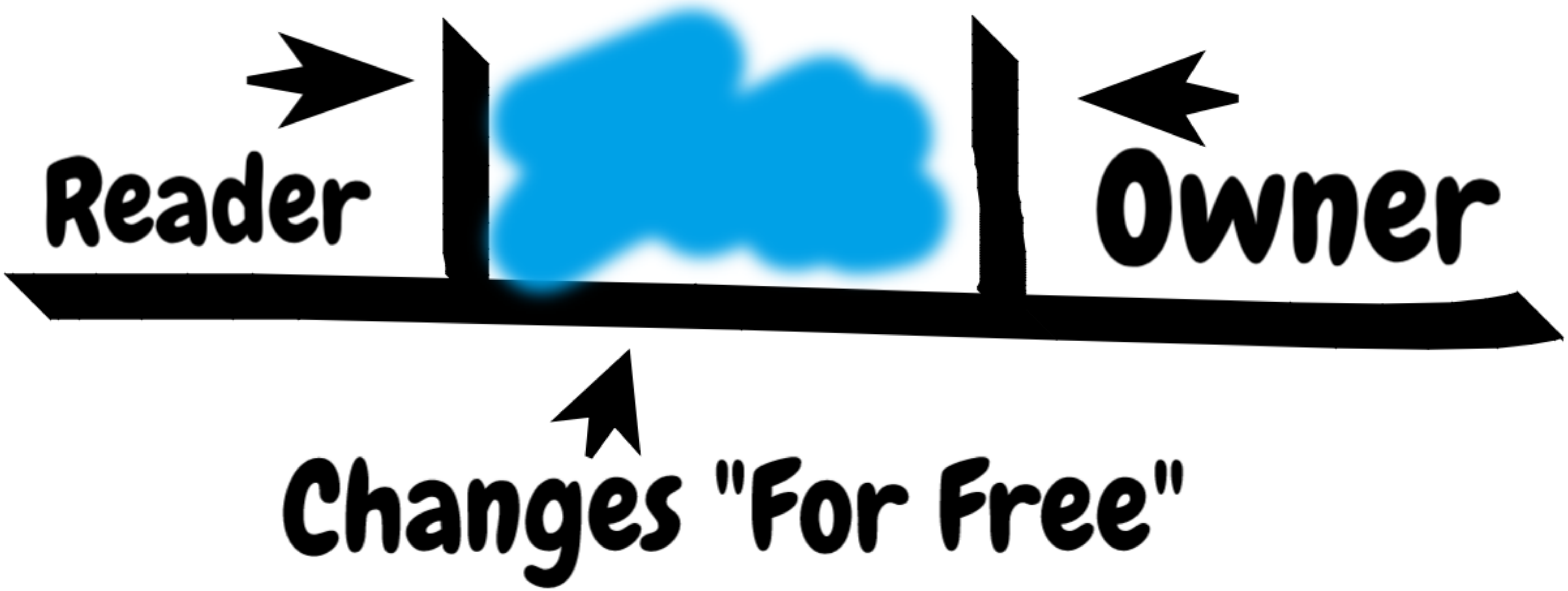
- data "owner": strongest guarantees possible
- reader/user: **relaxed** to what's (trully) necessary

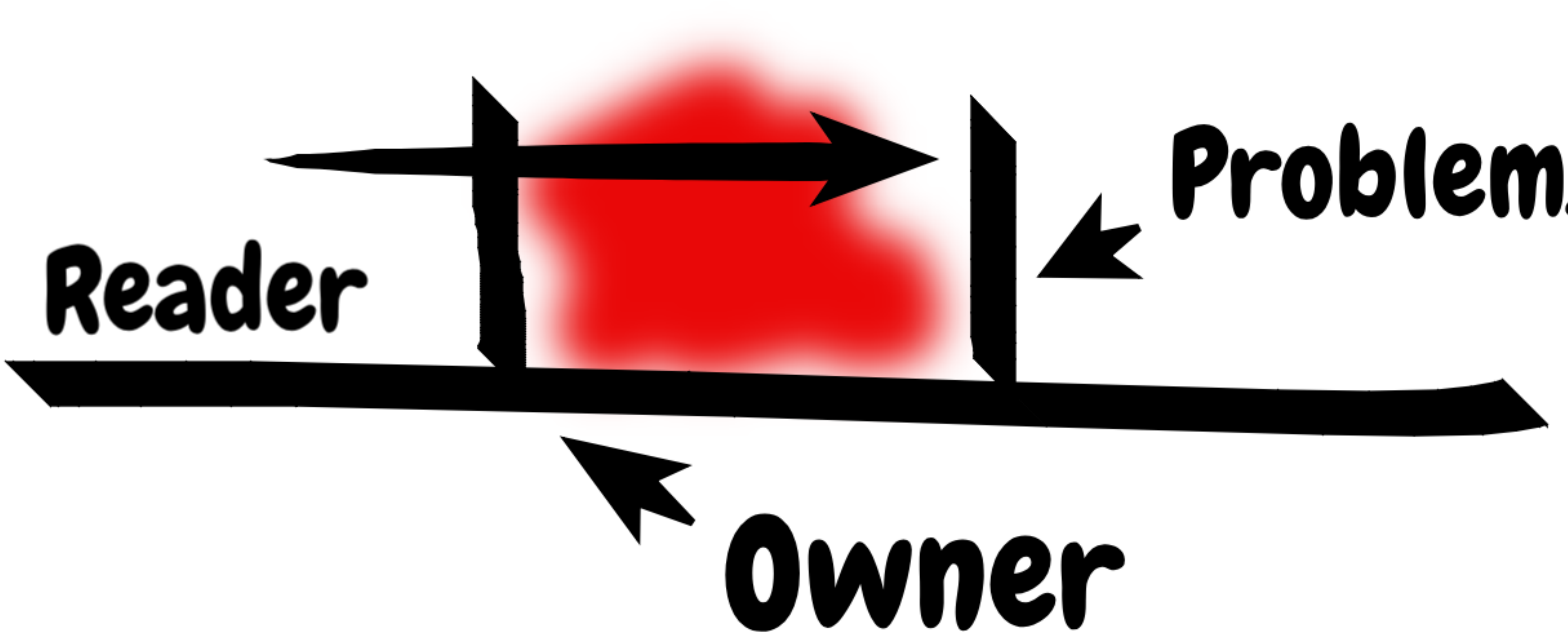
# Versioning: Type Twice

```clojure
(def EmailFromStorage
  (refined NonEmptyStr (BoundedSize _ 64) valid-email-re))

;; simply show on the screen?
(def Reader1 (refined NonEmptyStr (BoundedSize _ 64)))

;; I will truncate anyways :)
(def Reader2 NonEmptyStr)

;; I need to show "email me" button :(
(def Reader3 (refined NonEmptyStr valid-email-re))
```

## Versioning: Type Twice

- playing with predicates you're changing the scope
- scopes might intersect or be independent


Reader
Owner
Changes "For Free"


Reader
Problems
Owner

## Versioning: Type Twice

- most protocols support back- and forward- compatibility
  - `Protobuf`, `Thrift`, `FlatBuffers` & others
- rules are kinda implicit
- defined by protocol & libraries
- that's not enough 😒

## Versioning: Type Twice

- having all readers' & owners' type in a repo...
- anytime you change your types you know who's affected
  - writer guarantees >= reader expects
  - that's why you need "double definitions" 😜
- make it part of you CI cycle!

## Versioning: Refinements

- no theoretical **generic** solution (yet?)
- you can cover **a lot** of use cases "manually"
  - "if-else" driven type checker 😏
- provide "manual" proof in case of ambiguity
  - at least you have `git blame` now 😏
- advanced: run `QuickCheck` to double test that

## Summary Takeaways

## Summary

- JOINs: we did a lot, we still have a room for doing smarter
- protocol: choose wisely, don't be shy
- versioning: type your data (twice), keep types organized

Thanks!
Q&A PLS