The Kappa Architecture

Aris Kyriakos Koliopoulos

Drivetribe

- The world’s biggest motoring community.
- A social network for petrolheads.
- By Clarkson, Hammond and May.

Drivetribe

- Built upon the concept of a “tribe”, which is a group of people sharing a common interest.
- Those tribes have their own feeds, discussion boards and chat channels.
- This model lends itself quite well to fragmented verticals such as motoring.

POSTS DISCUSSIONS 72

he should have said his line I'm going to crash

1 month ago · **Reply** (1)

Jeremy Clarkson

21 Jun · 198.5K Views

My work is done. That's one hell of a car

COMMENTS (145) **REPOST** **BUMPS** (1.9K)

Jeremy Clarkson

20 Jun · 242.5K Views

Now that's what I call a speed-o-meter

COMMENTS (98) **REPOST** **BUMPS** (1.9K)

Joe odero

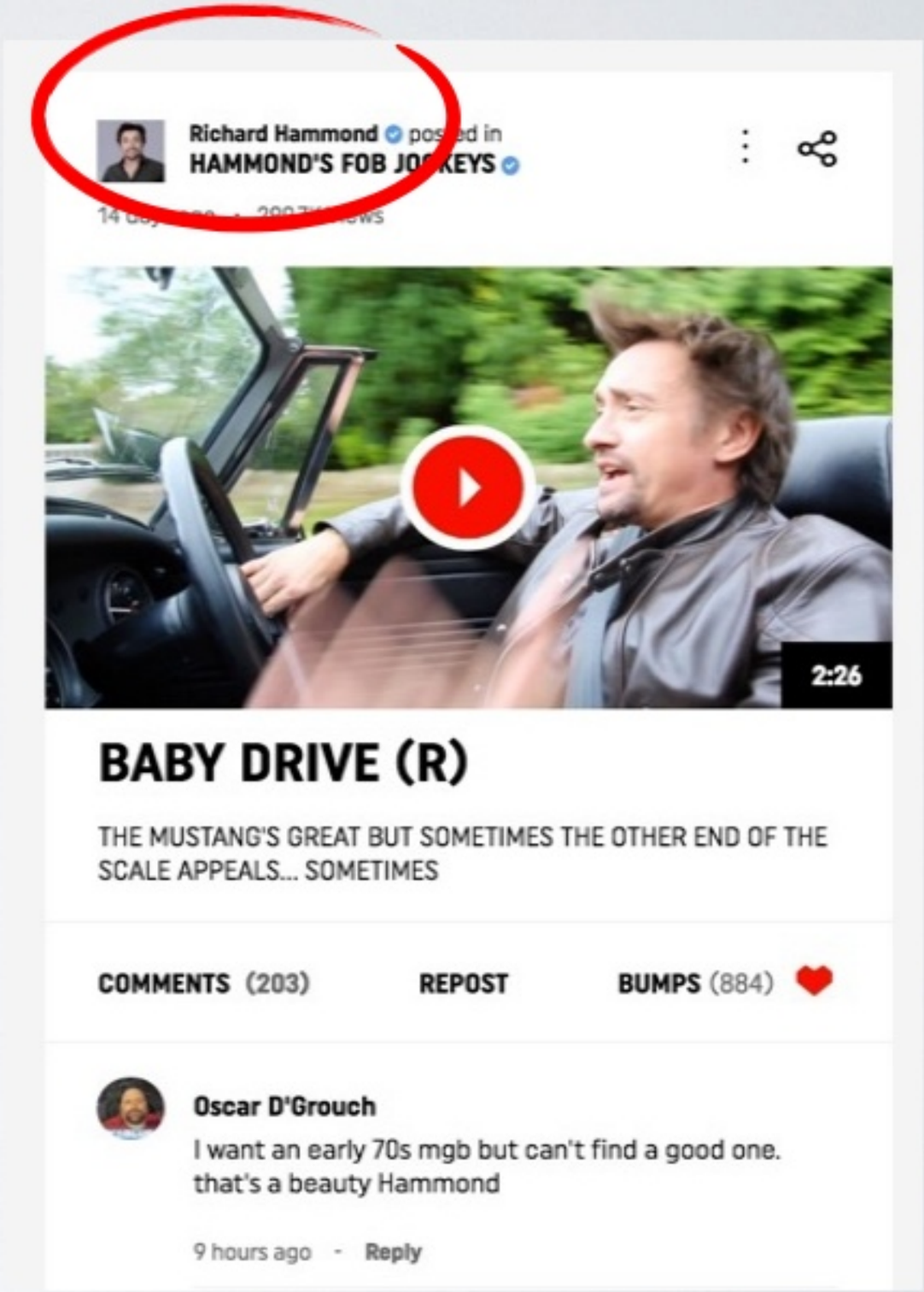
black and white

15 days ago · **Reply**

Let’s zoom in a little bit.

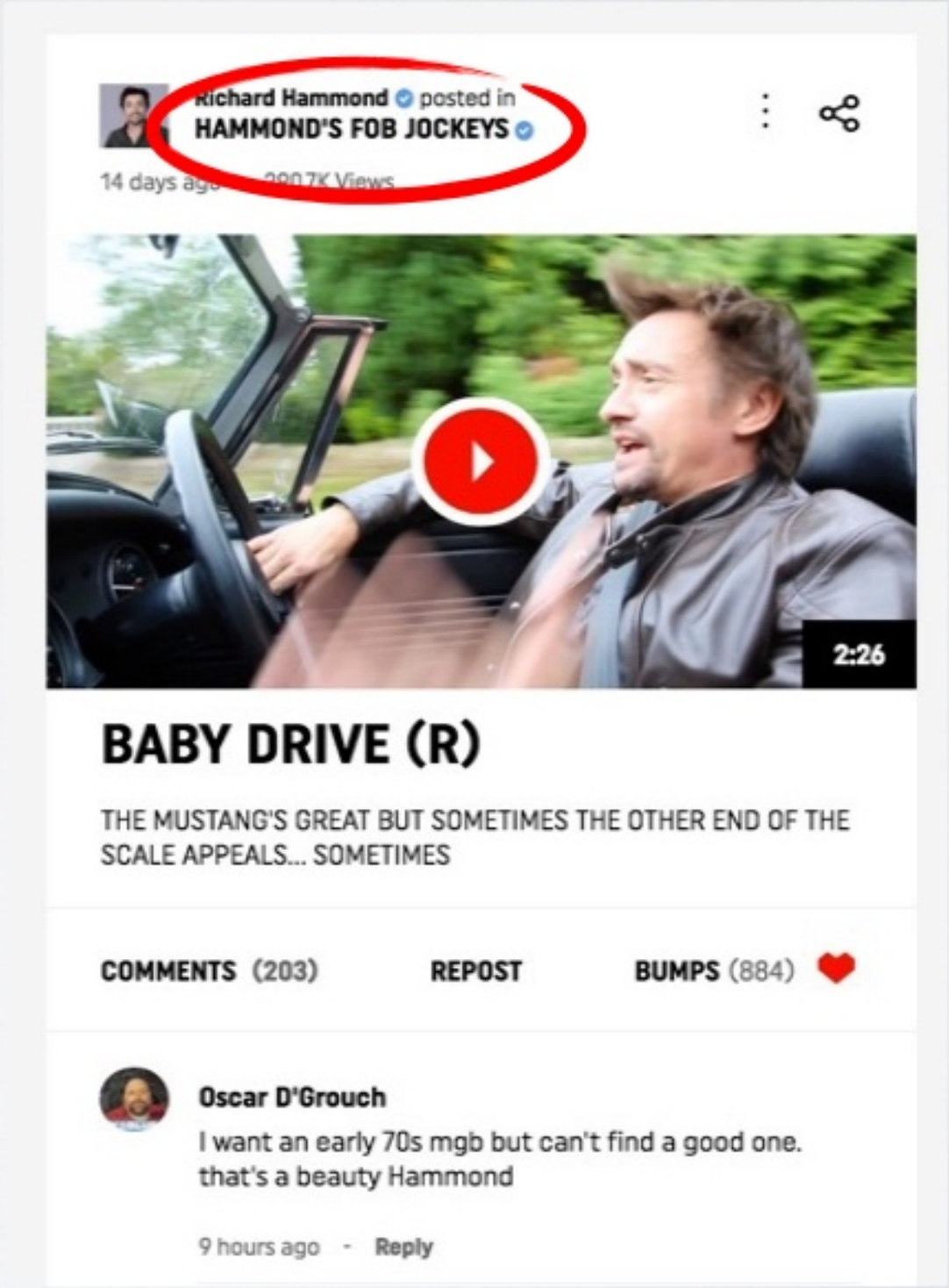
Computing a feed item

- **getUser(userId: Id[User])**



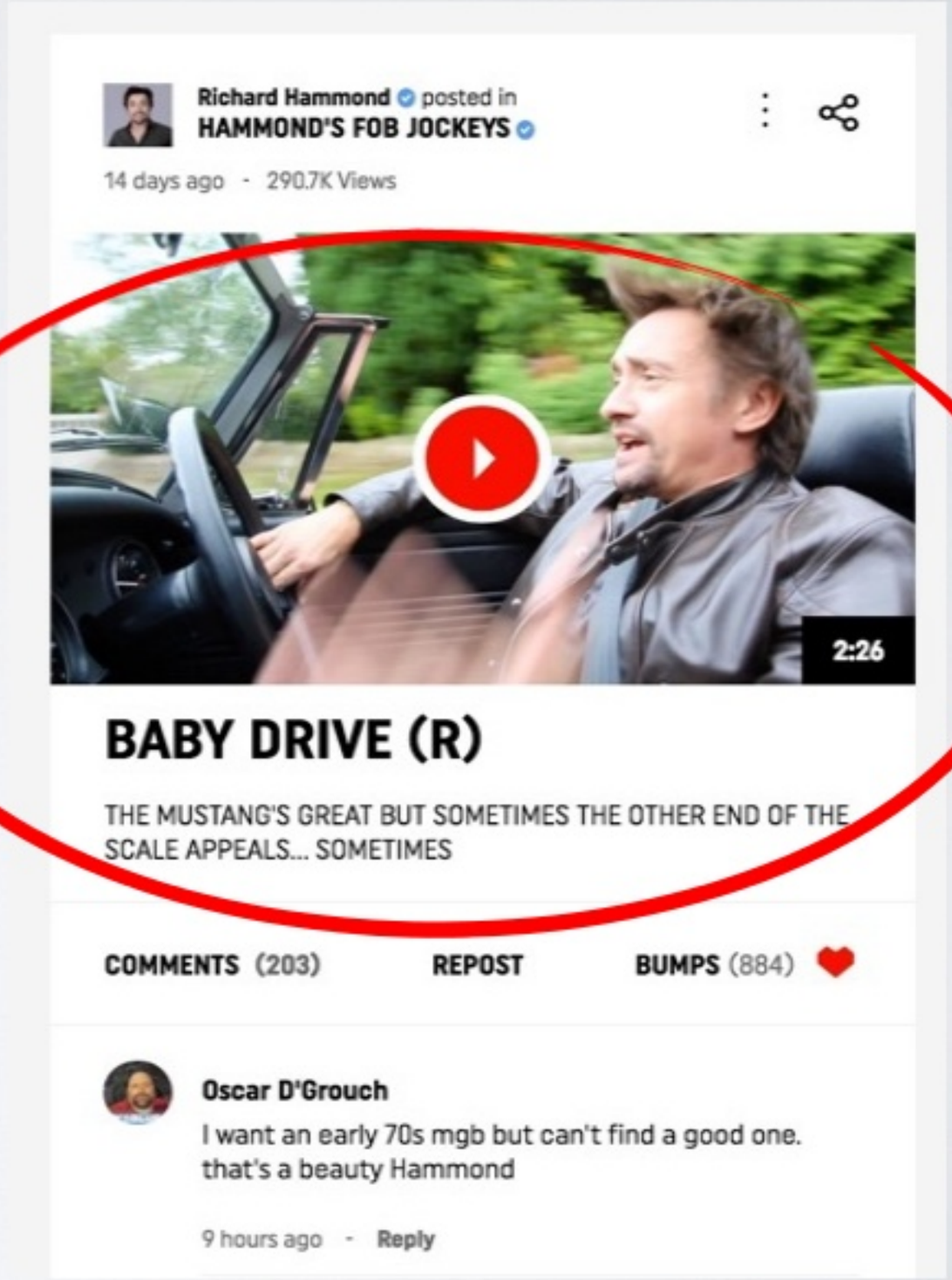
Computing a feed item

- getUser(userId: Id[User])
- **getTribe(tribeId: Id[Tribe])**



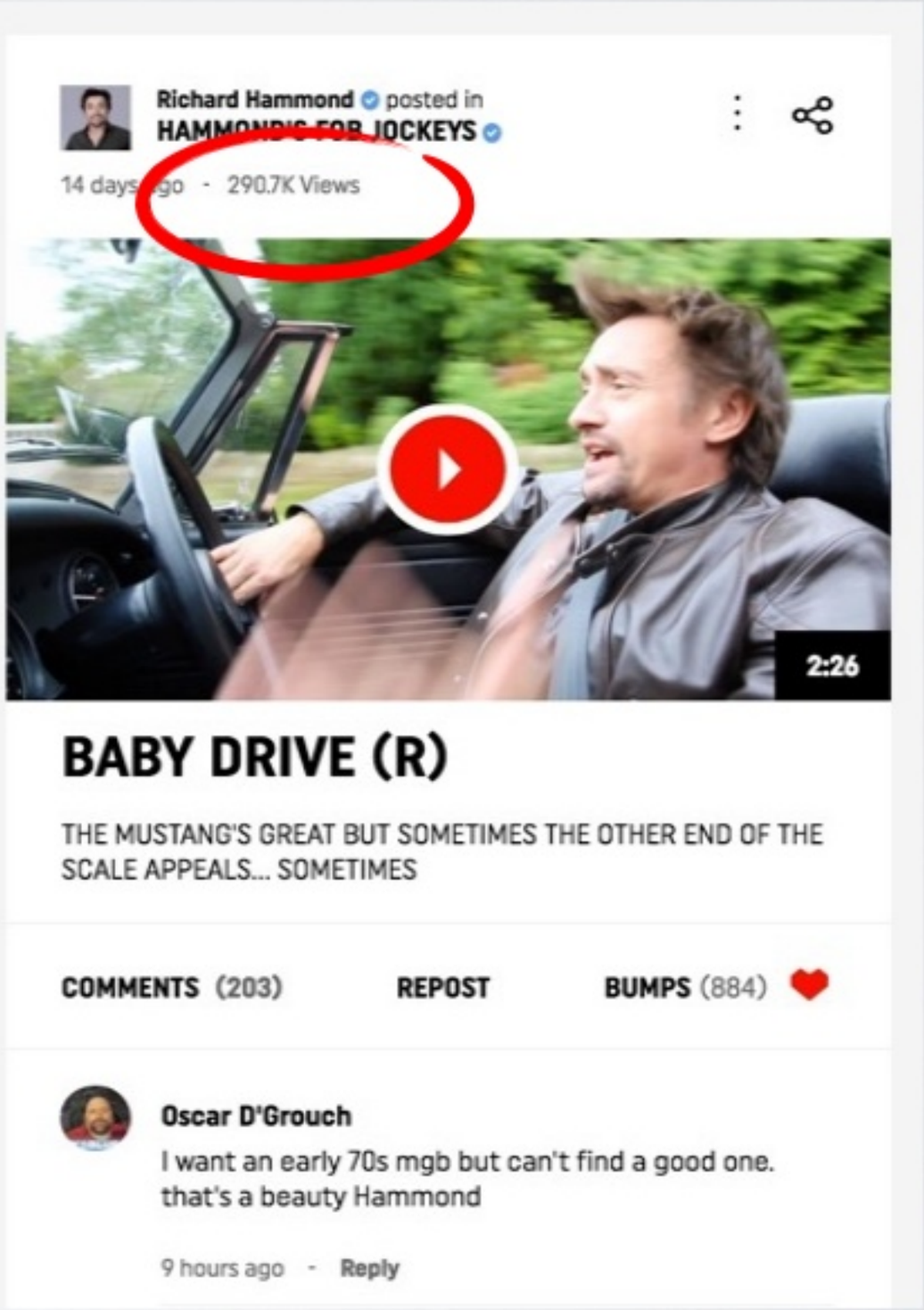
Computing a feed item

- getUser(userId: Id[User])
- getTribe(tribeId: Id[Tribe])
- **getPost(postId: Id[Post])**



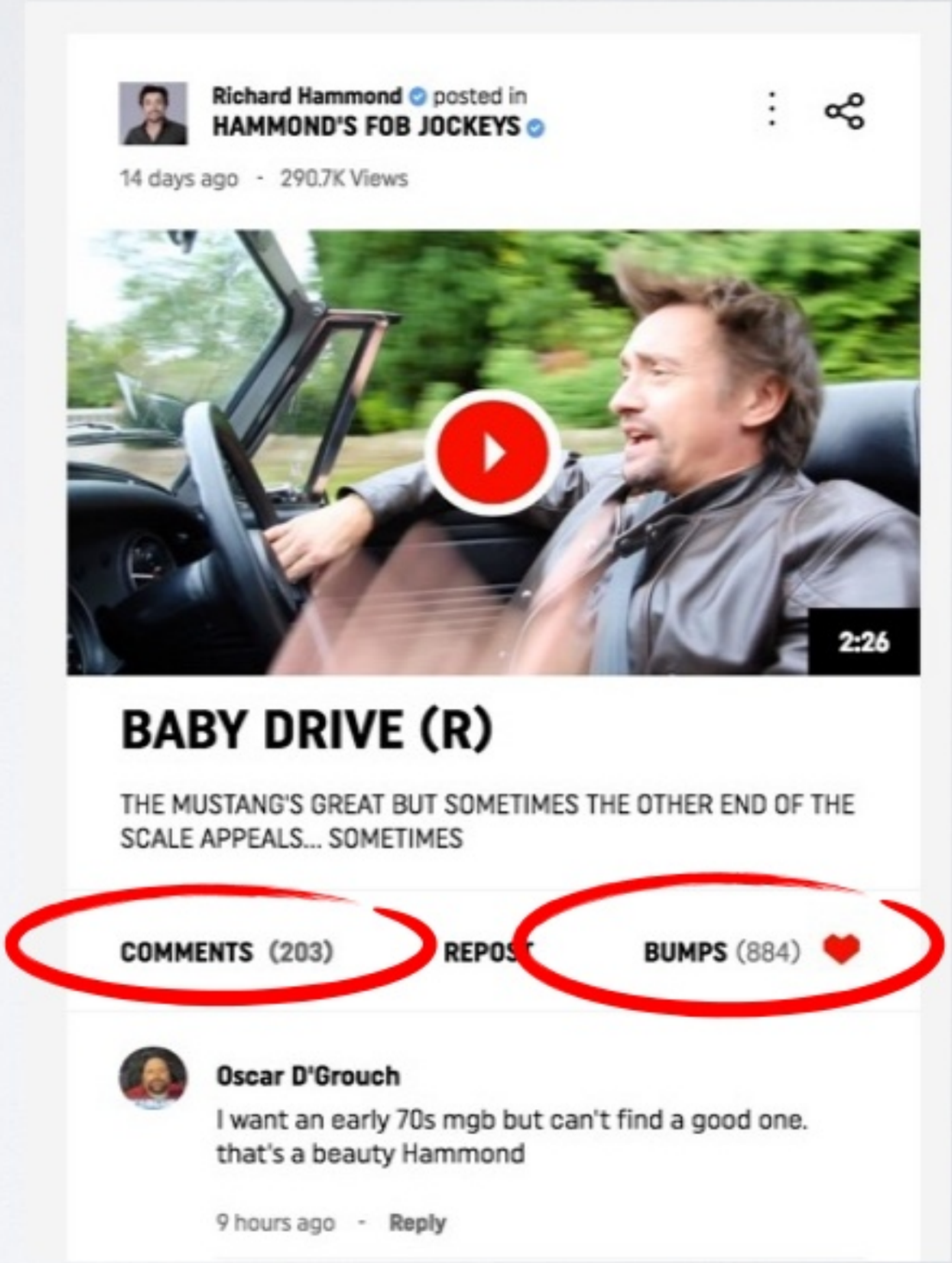
Computing a feed item

- getUser(userId: Id[User])
- getTribe(tribeId: Id[Tribe])
- getPost(postId: Id[Post])
- **countViews(postId: Id[Post])**



Computing a feed item

- getUser(userId: Id[User])
- getTribe(tribeId: Id[Tribe])
- getPost(postId: Id[Post])
- countViews(postId: Id[Post])
- **countLikes(postId: Id[Post])**
- **countComments(postId: Id[Post])**
- Maybe a few others.



Richard Hammond posted in HAMMOND'S FOB JOCKEYS

1 days ago - 290.7K Views

2:26

BABY DRIVE (R)

THE MUSTANG'S GREAT BUT SOMETIMES THE OTHER END OF THE SCALE APPEALS... SOMETIMES

COMMENTS (203) REPOST BUMPS (884)

Oscar D'Grouch

I want an early 70s Mach but can't find a good one. that's a beauty Hammond

9 hours ago - Reply

Andrew Lopez

Old or New Mustang?

1 day ago · Reply

Richard Hammond posted in HAMMOND'S FOB JOCKEYS

22 days ago - 449.3K Views

1:08

BACK BEHIND THE WHEEL!

TODAY I TOOK MY FIRST DRIVE SINCE MY RECENT VISIT TO SHUNTSVILLE

COMMENTS (343) REPOST BUMPS (2K)

Heather Marsters

lmao too bad I'm in Texas, otherwise I'd go chase that ghost with you

1 day ago - Reply

Jon Rabuck

The Hamster is officially out of the cage!!! BTW...Sweet car!! It doesn't get much better than that!

Problem summary

- Social feeds containing complex aggregates.
- Potentially algorithmically ranked and personalised.
- Potentially for thousands of concurrent users.

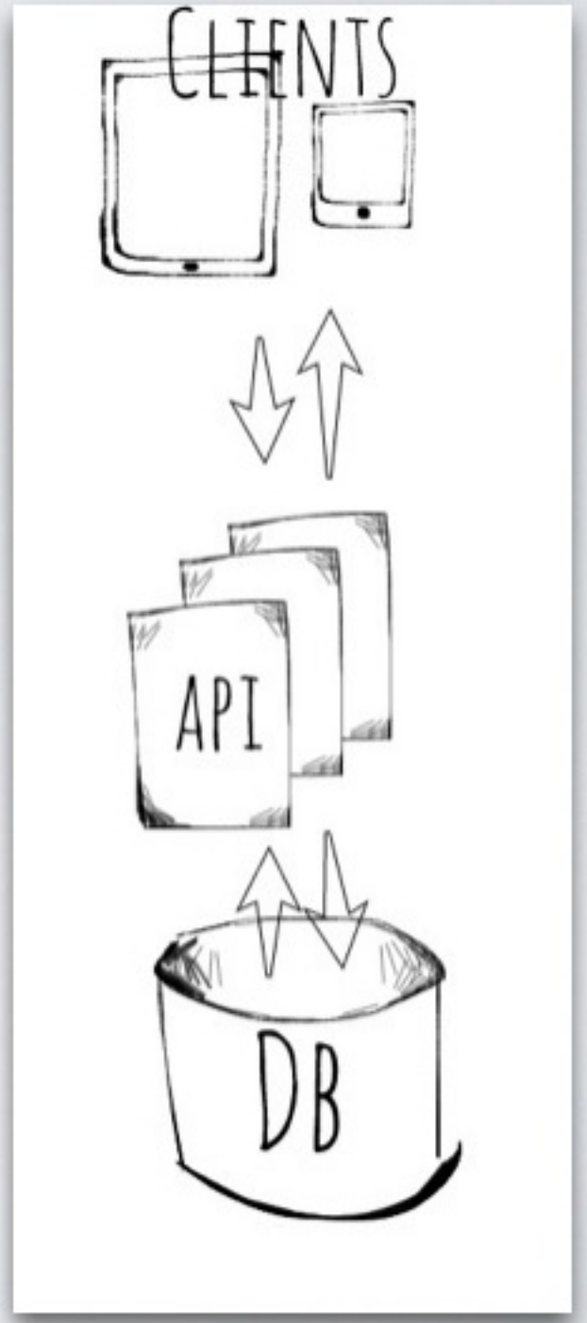
How can we solve this problem?

Principles

- **Scalable**. CHM have tens of millions of social media followers.
- **Performant**. Low latency is key. Networks are already adding a lot of it.
- **Extensible**. Young start-ups tend to iterate a lot.
- **Maintainable**. Spaghetti code becomes expensive over time.

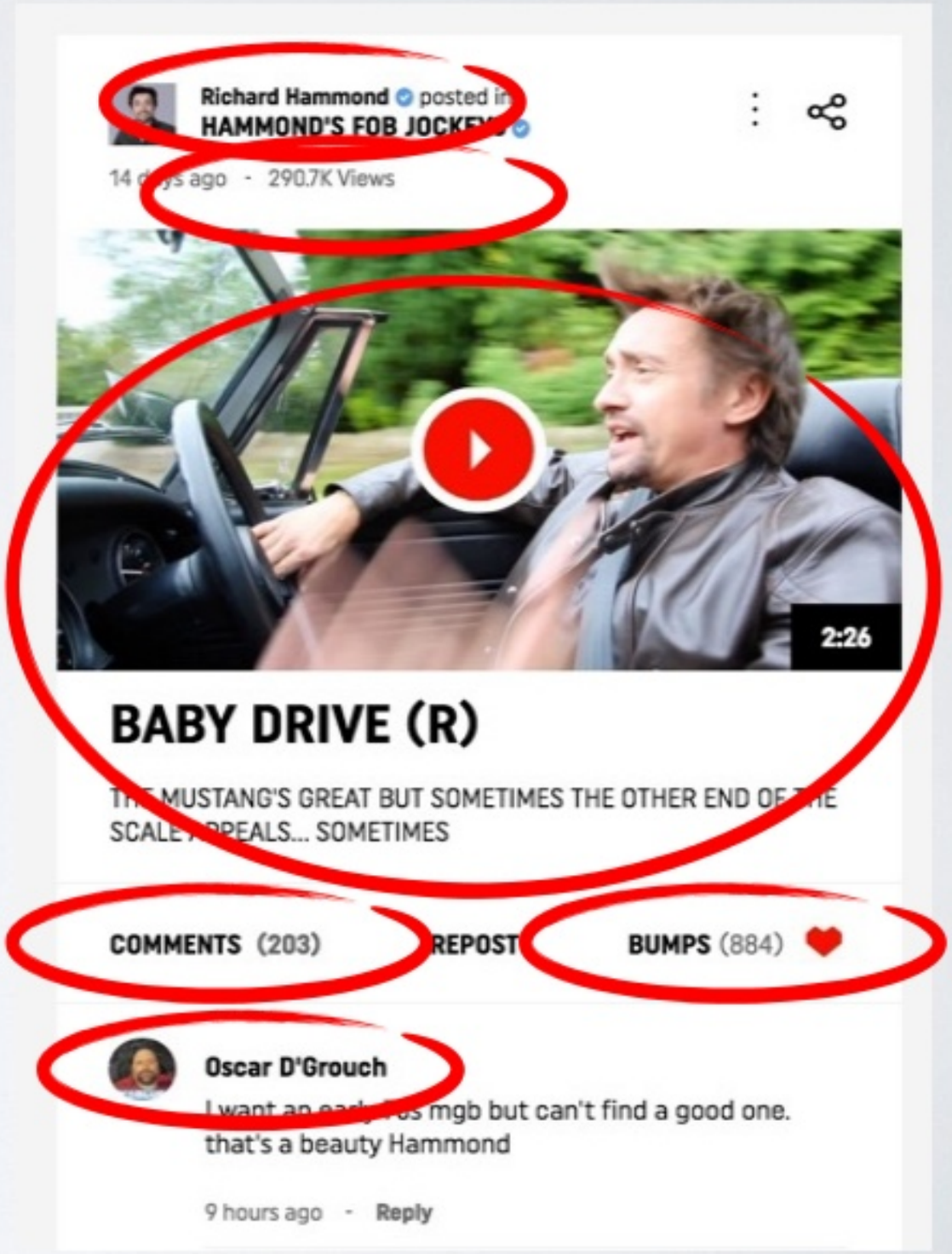
Naive approach

- Clients interact with a fleet of stateless servers (aka “API” servers or “Backend”) via HTTP (which is stateless).
- Global shared mutable state (aka the Database).
- Starting simple: Store data in a DB.
- Starting simple: Compute the aggregated views on the fly.



Naive approach

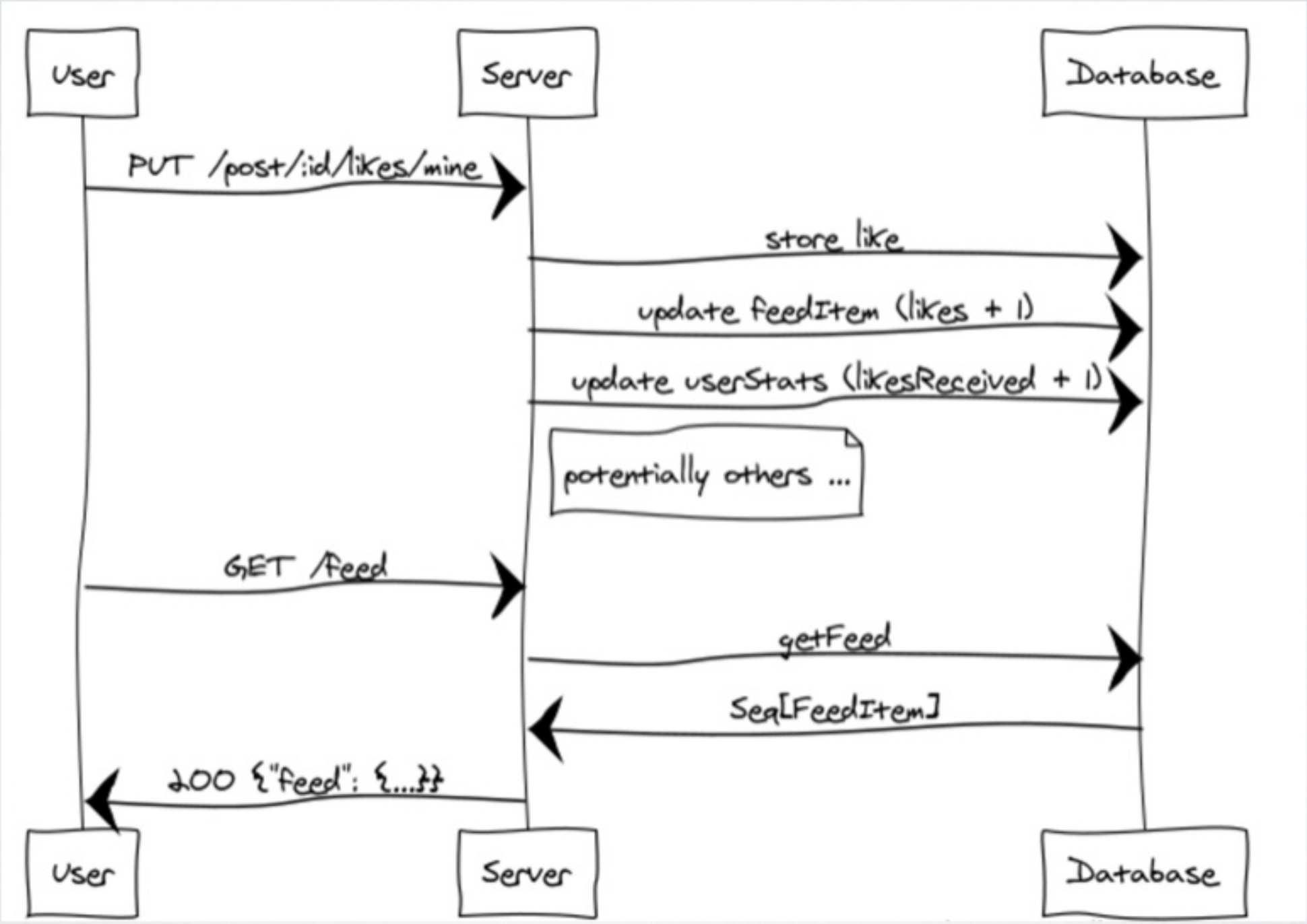
- (~10 queries per Item) x (items per page)
- Count queries are generally expensive and become slower over time.
- Slow. Not really Performant.
- Not really Scalable either.



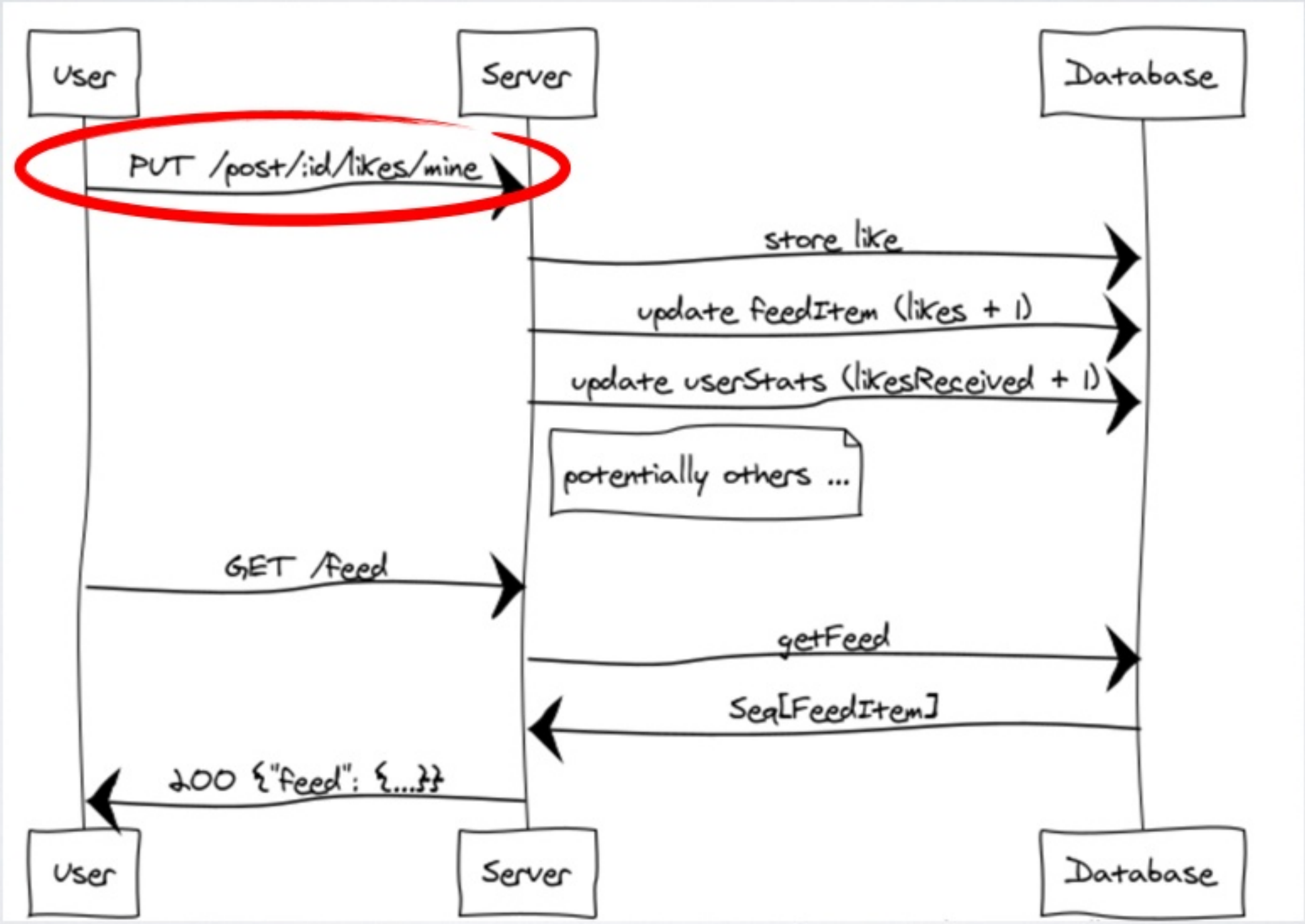
Can we make this scale?

- **Aggregate at write time**. Do the heavy lifting when the user is **not** waiting.
- Then sorting and fetching needs a single query.
- This is **Performant**.

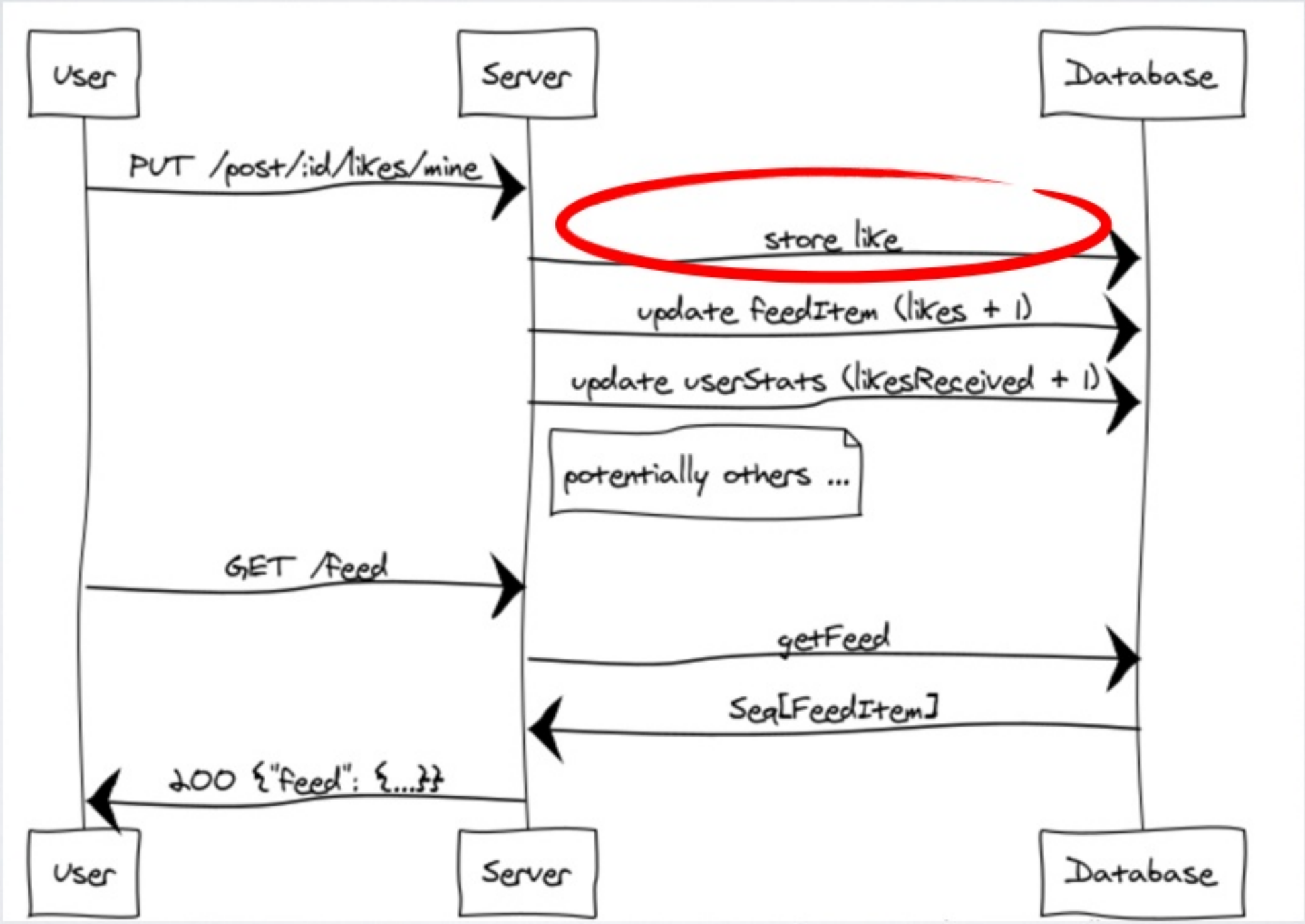
Write time aggregation



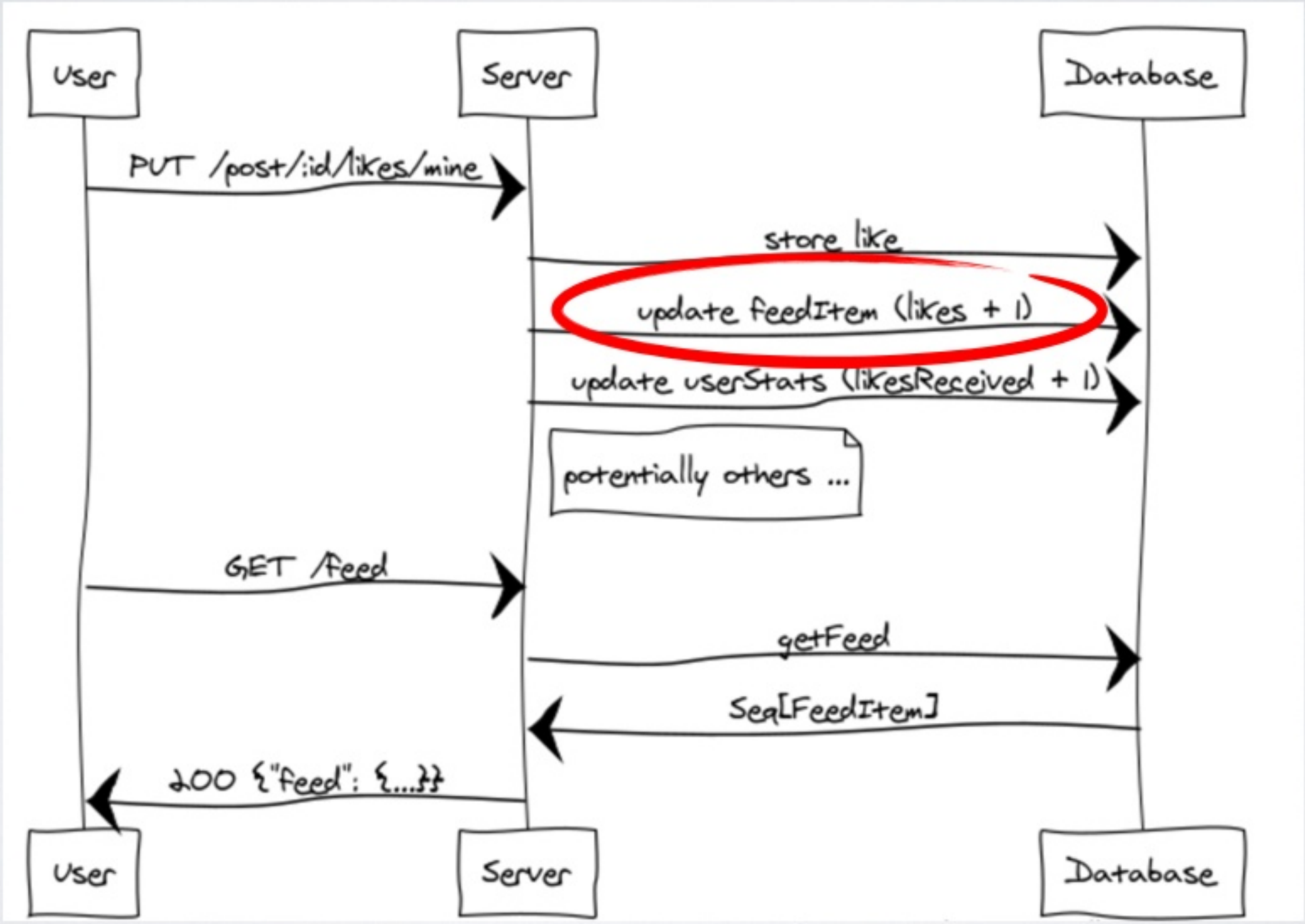
Write time aggregation



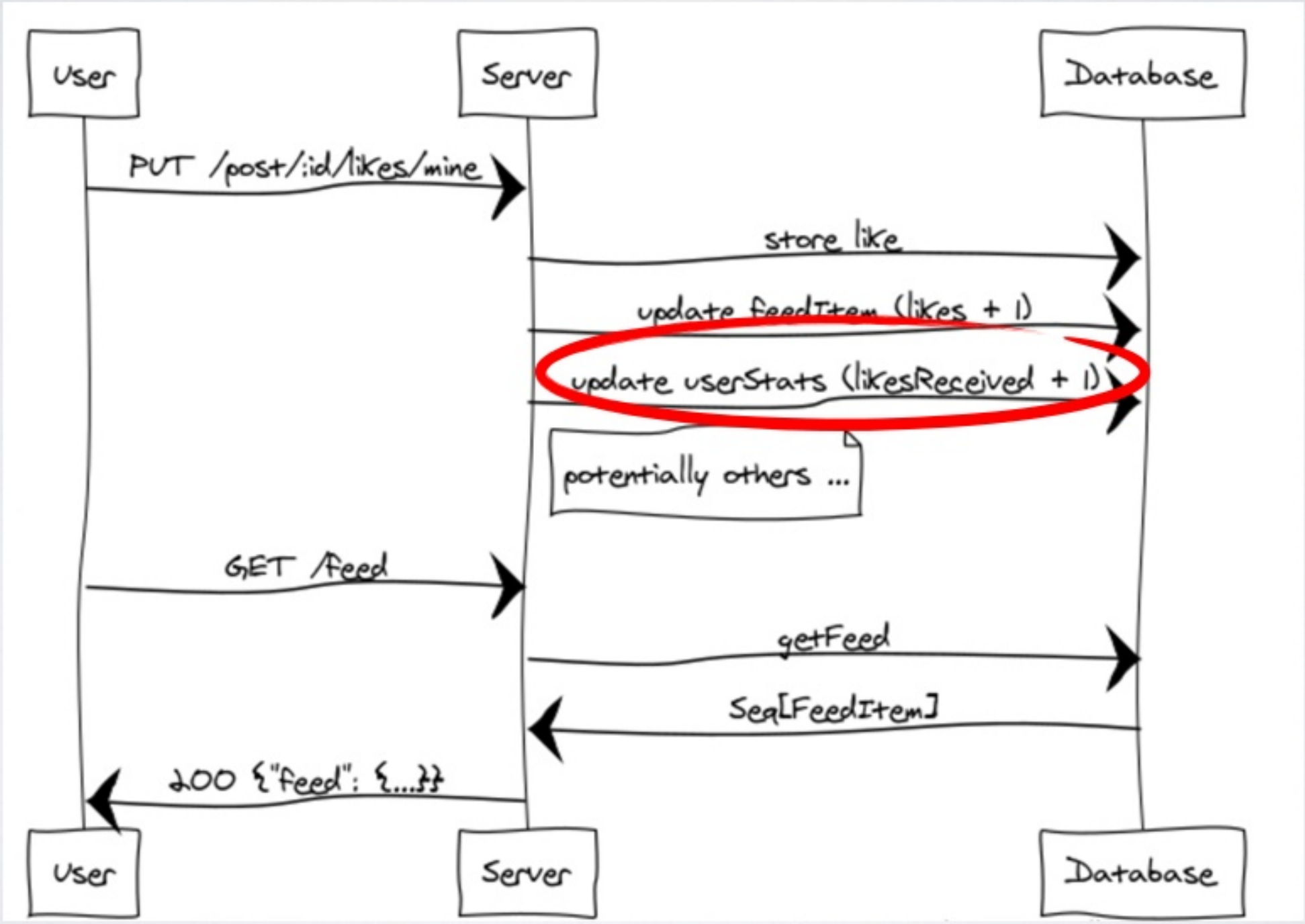
Write time aggregation



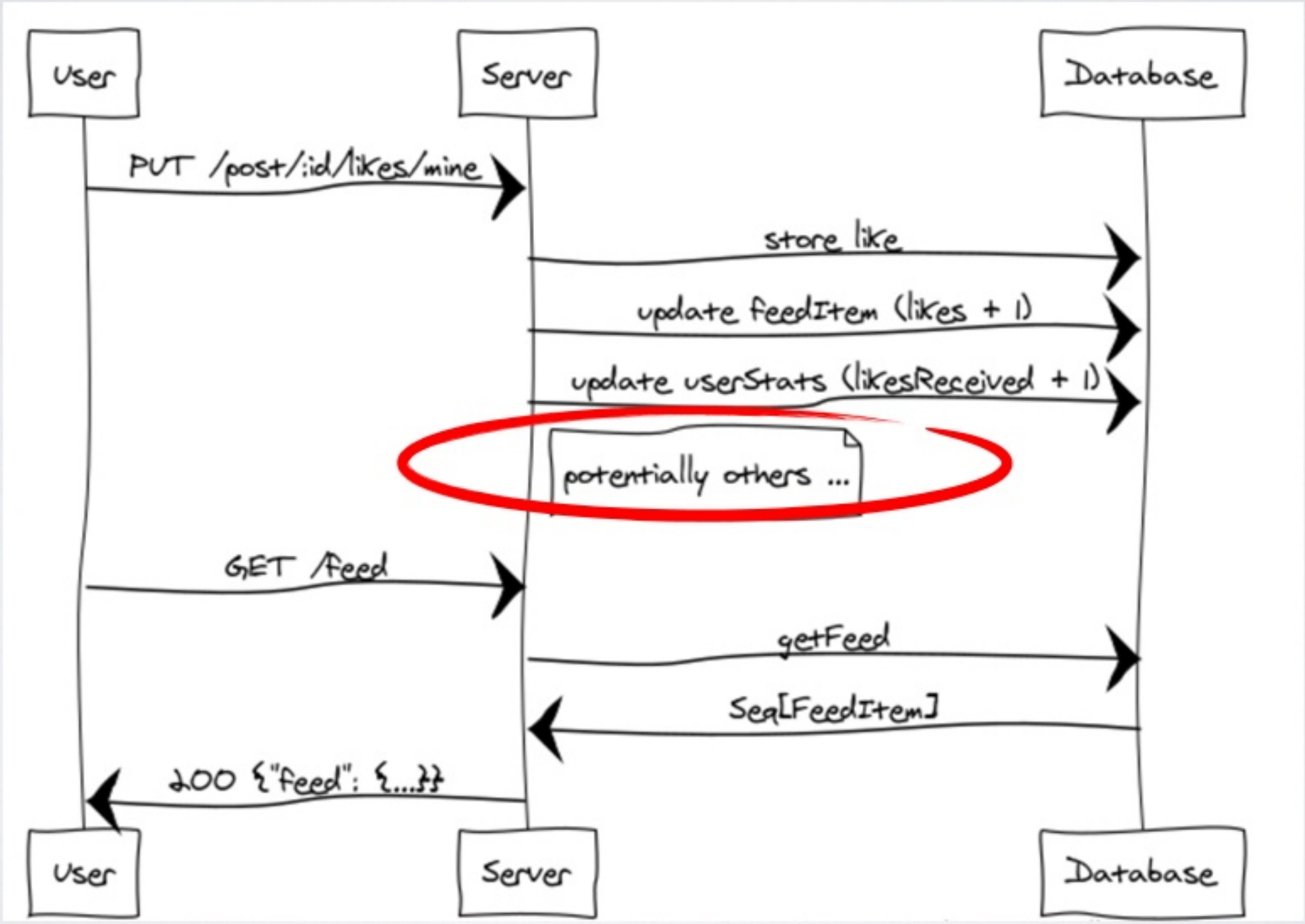
Write time aggregation



Write time aggregation



Write time aggregation



Sounds good, but...

Write-time aggregation - Shortcomings

- Concurrent mutations on a global shared state entail race conditions.
- Locking mechanisms limit throughput. The write path cannot scale for hot keys.
- State mutations are destructive and can not be (easily) undone.
- A bug can corrupt the data permanently.

Write-time aggregation - Shortcomings

- Model evolution becomes difficult. Reads and writes are tightly coupled.
- Migrations are scary.
- This is neither **Extensible** nor **Maintainable**.

Let’s take a step back...

Facts

- Clients send facts to the API: “John liked Jeremy’s post”, “Maria updated her profile”
- Facts are **immutable.** They capture a user action that happened at some point in time.

- Logging those facts would yield an **append-only log** of events.

- We don’t just capture the state. We can also capture state transitions.

- Immutability ensures we can reprocess the log as required **without side-effects**.

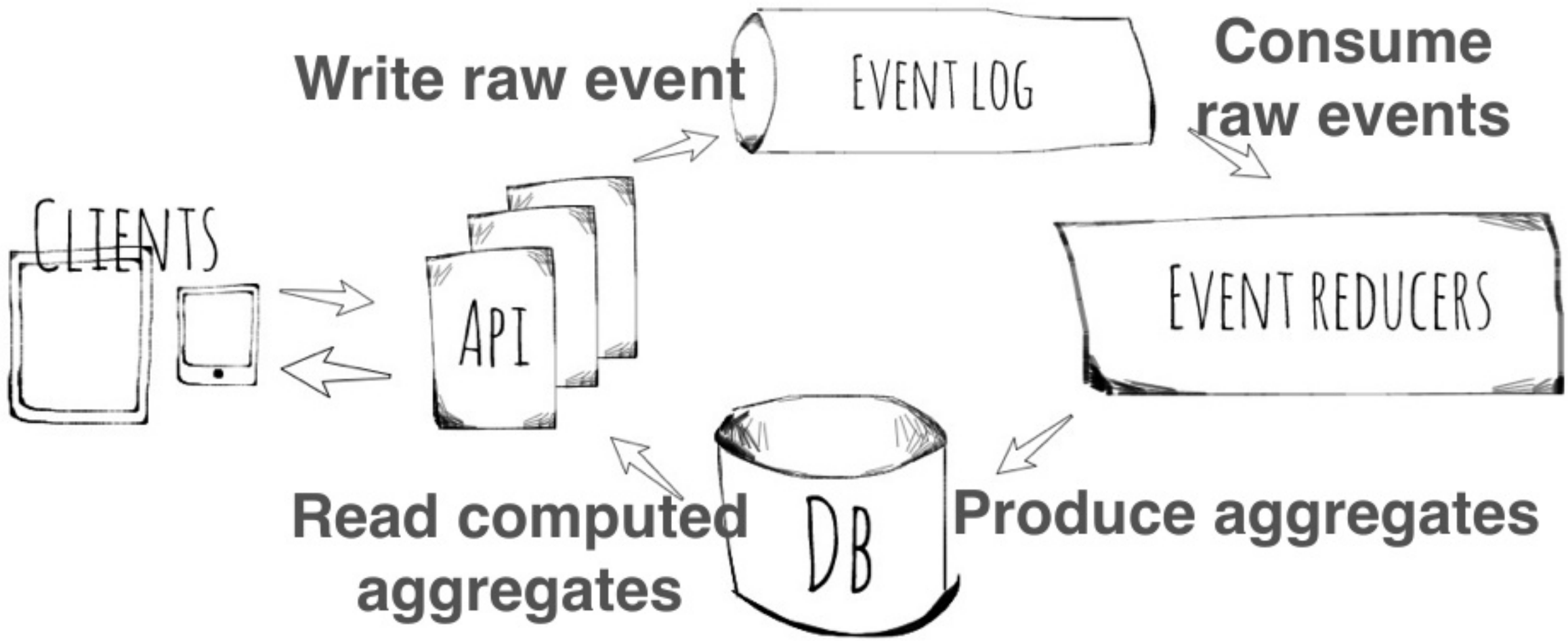
- This is **event sourcing**.

- Event consumers can process the log and produce read-time optimised aggregates.
- Essentially read-time and write-time models are decoupled. They can be implemented, scaled and troubleshot independently.
- This is **CQRS** (Command-Query-Responsibility-Segregation).

Event sourcing & CQRS

- Immutability makes reasoning about the system easier. State transitions can be traced and reproduced.
- Separating reads and writes provides better flexibility in system design.
- The read model is just a projection that can be easily changed.
- Some people call it the Kappa approach.

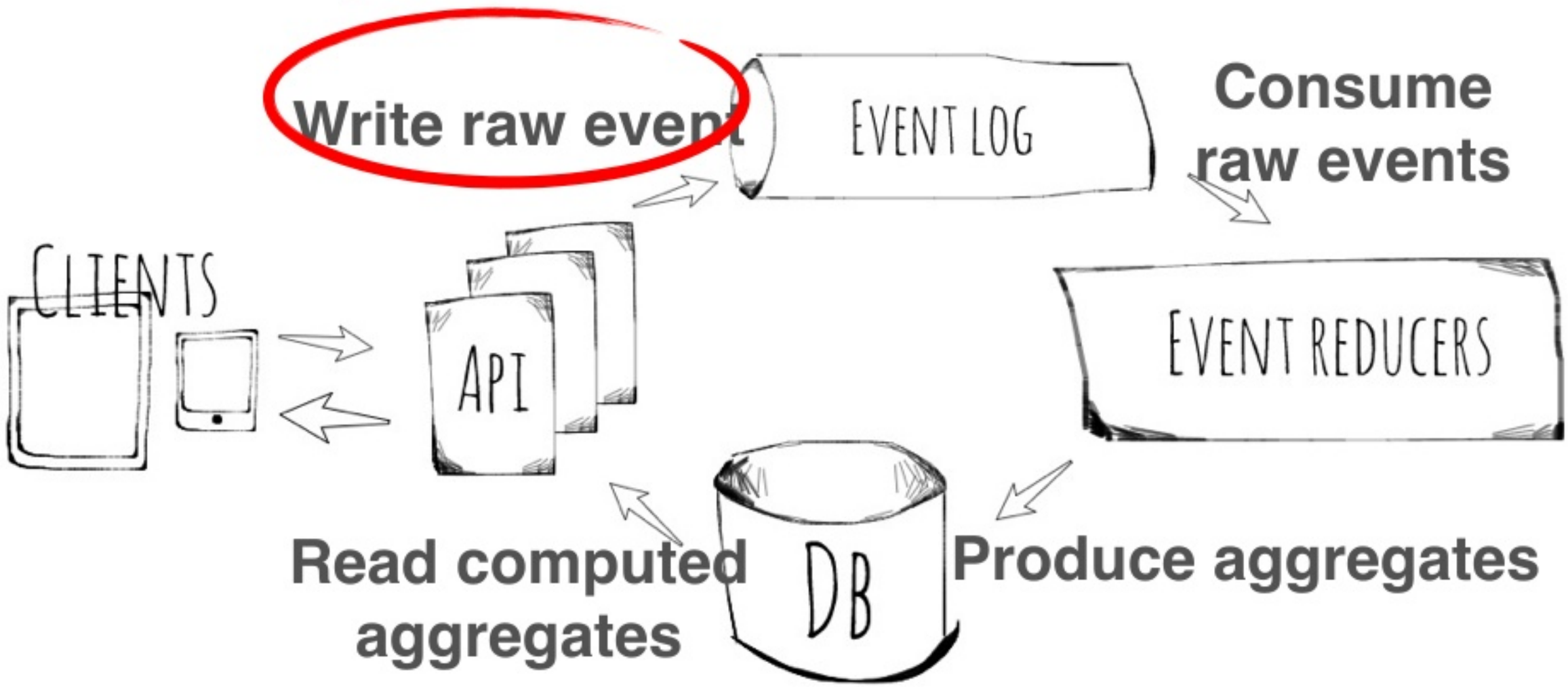
Kappa approach



Kappa approach

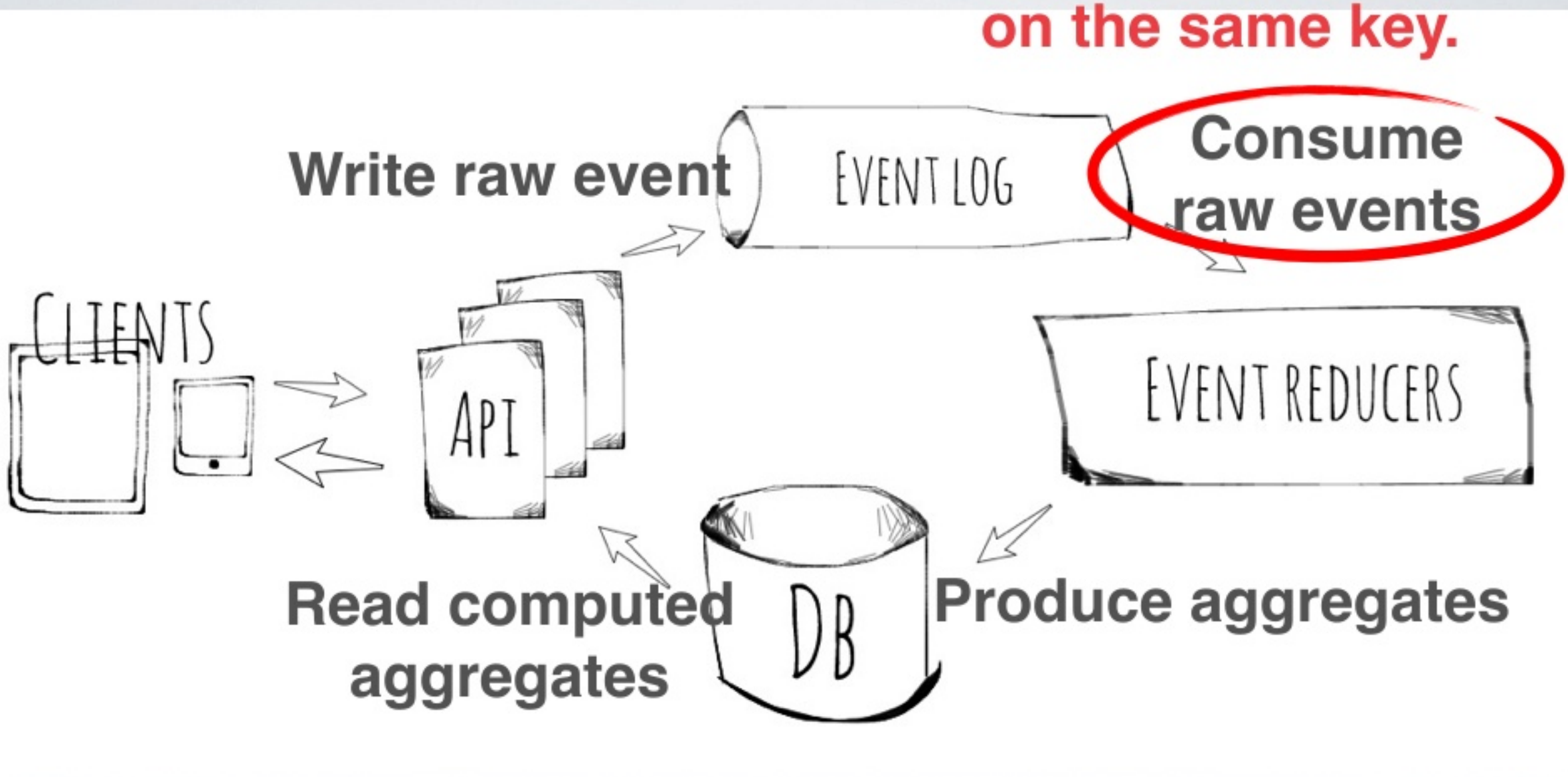
pending to a log is sequential I/O.

Generally very fast.

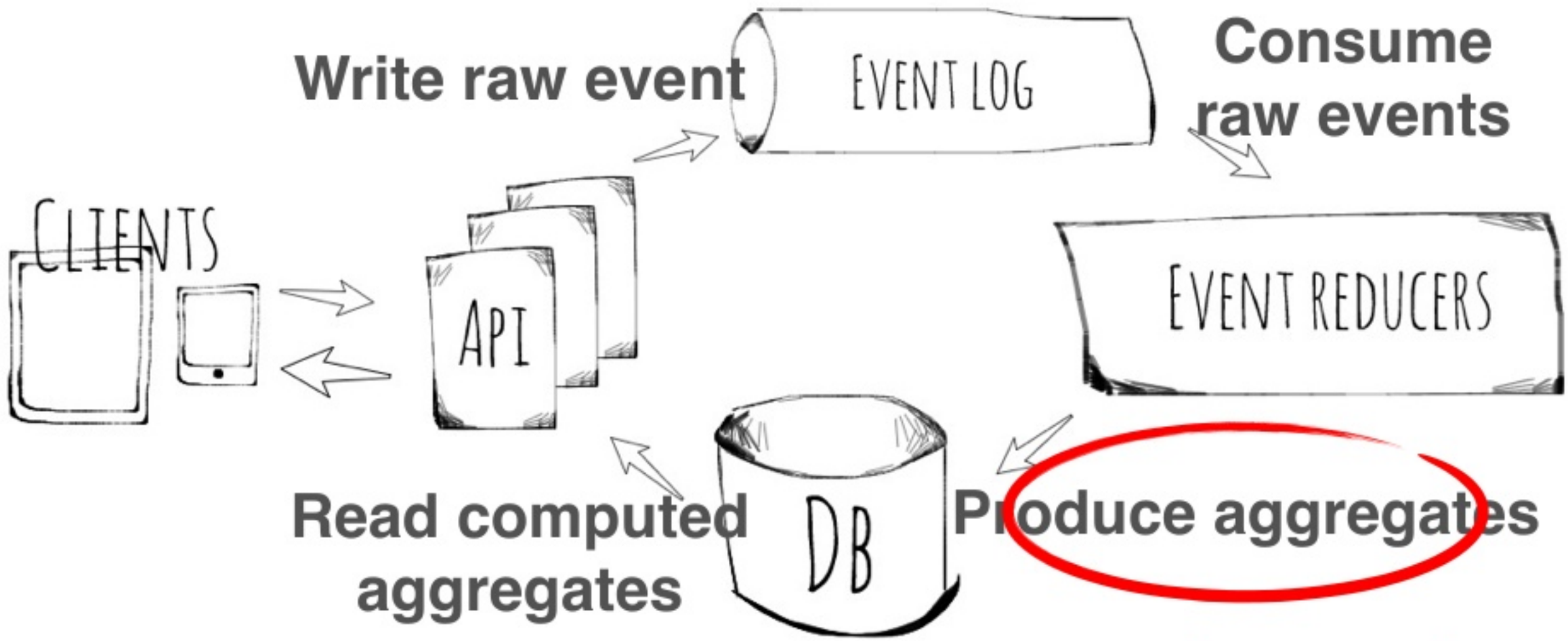


Kappa approach

No need for concurrent update on the same key.



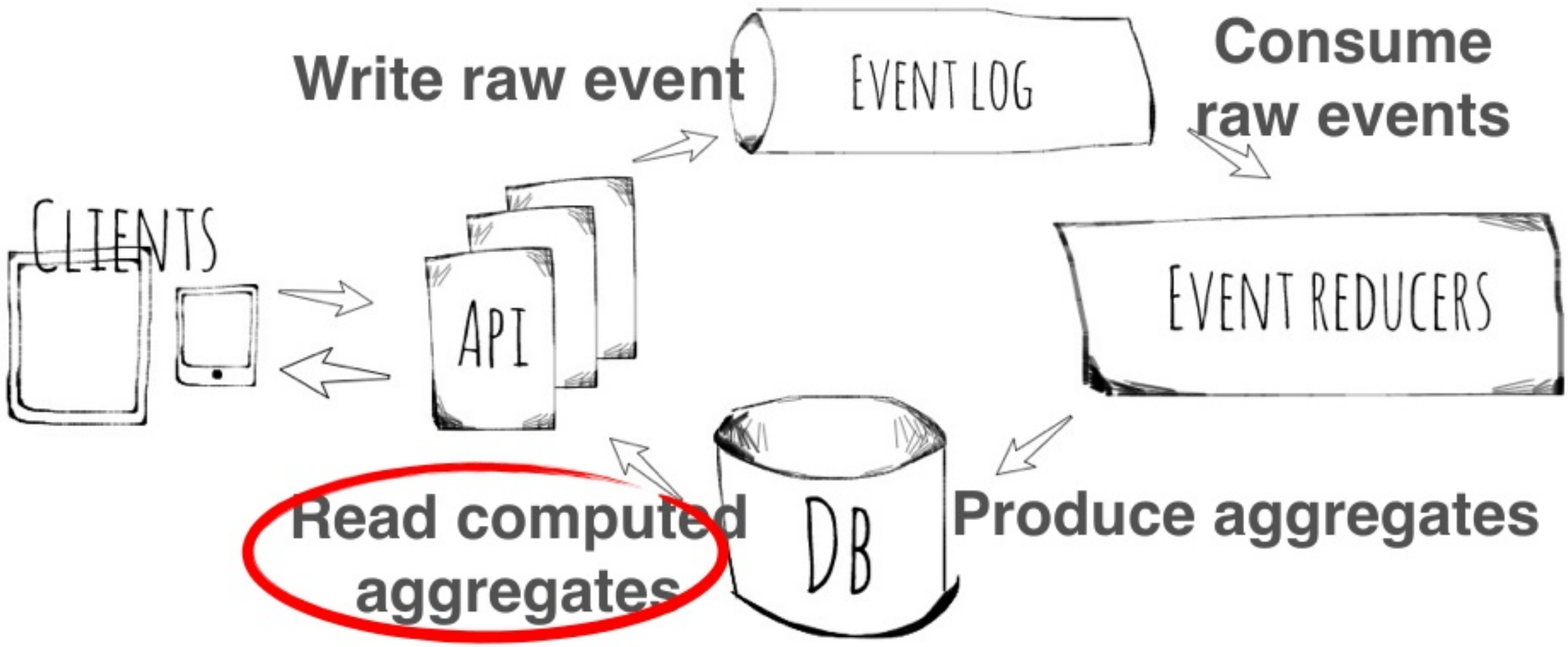
Kappa approach



Very easy to extend/change/debug.

Just update the reducer code and replay th

Kappa approach



Efficient, read time optimised aggregates.

Kappa approach

- Reads and writes are decoupled and optimised independently. **Scalability** ✔ **Performance** ✔
- The write model is just a projection. It can be easily extended/changed without painful migrations. **Extensibility** ✔ **Maintainability** ✔

Kappa approach - trade offs

- Consistency is eventual. Not guaranteed to read your own writes.
- There is an operational overhead. More moving parts.
- Failure scenarios get more complicated.

How can this be implemented?

1. We need a log.

Apache Kafka

- Distributed, fault-tolerant, durable and fast append-only log.
- Each business event type can be stored in its own topic.

2. We need consumers.

Apache Flink

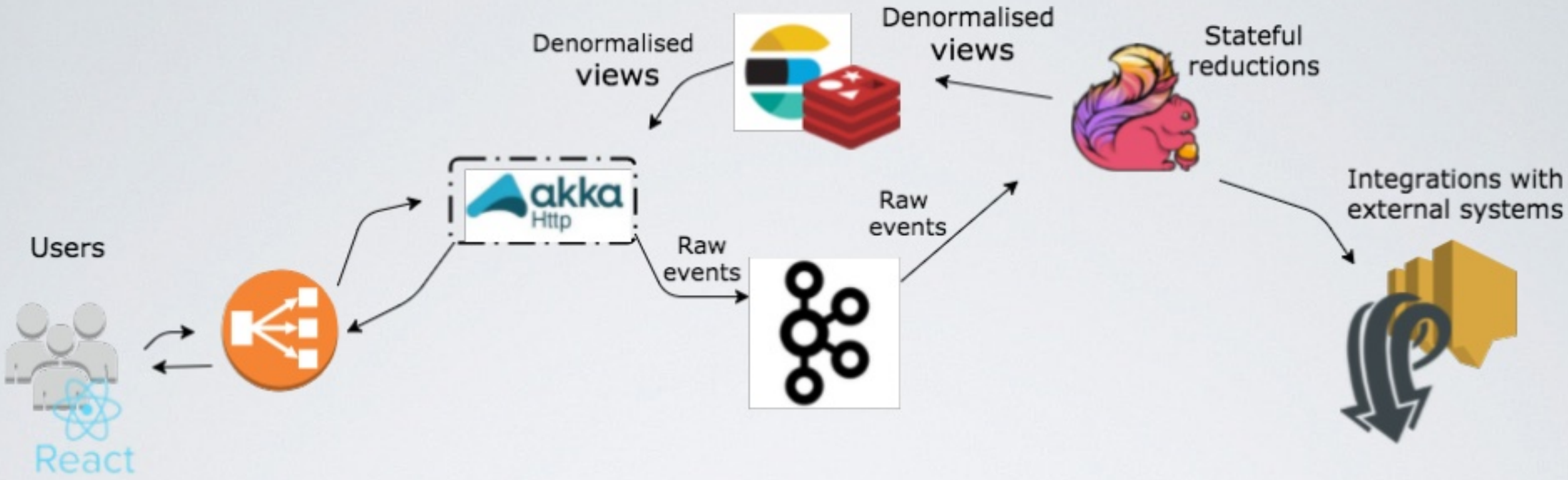
- Scalable, performant, mature.
- Elegant high level APIs in Scala.
- Powerful low level APIs for advanced tuning.
- Multiple battle-tested integrations.
- Very nice and active community.

3. And a serving layer.

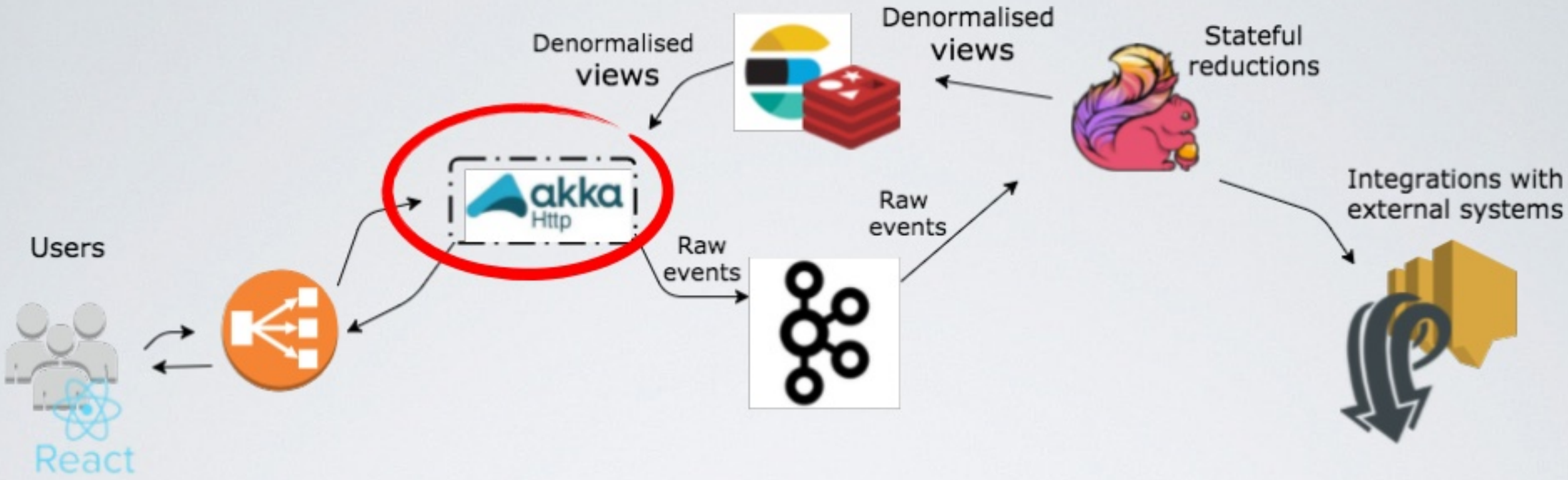
Elasticsearch and Redis

- Elasticsearch as a queryable document store.
- Redis to store relationships for quick access.
- Dispensable. Those can come and go as required.

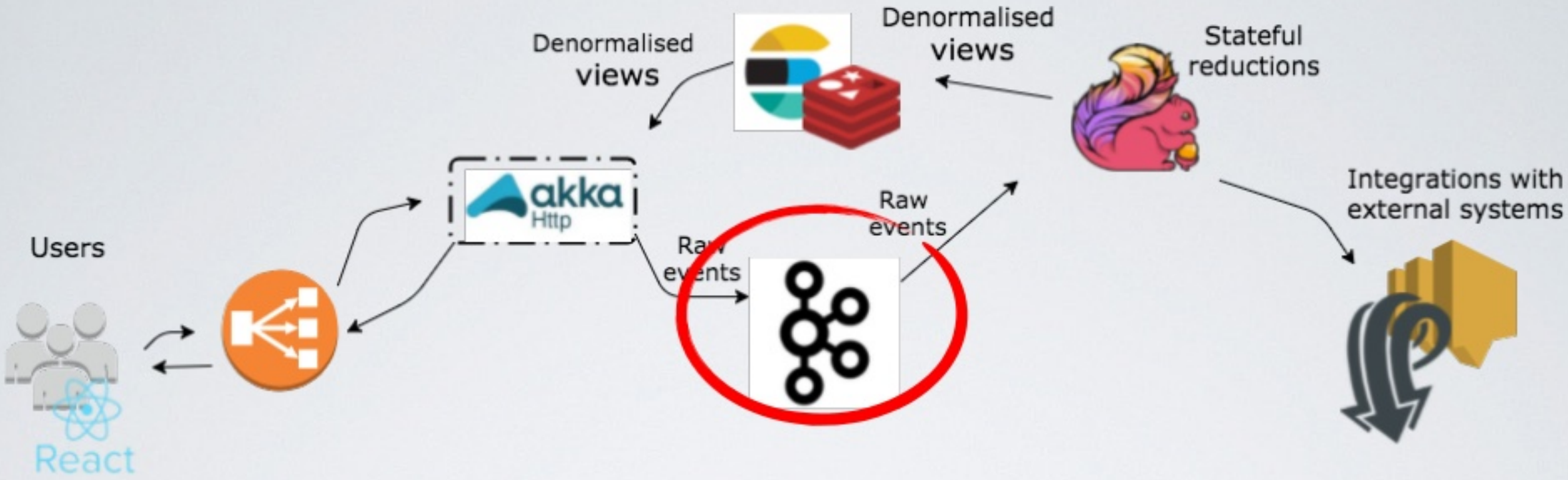
Putting everything together…


Denormalised
views
Denormalised
views
Stateful
reductions
akka
Http
Integrations with
external systems
Raw
events
Raw
events
Users
React

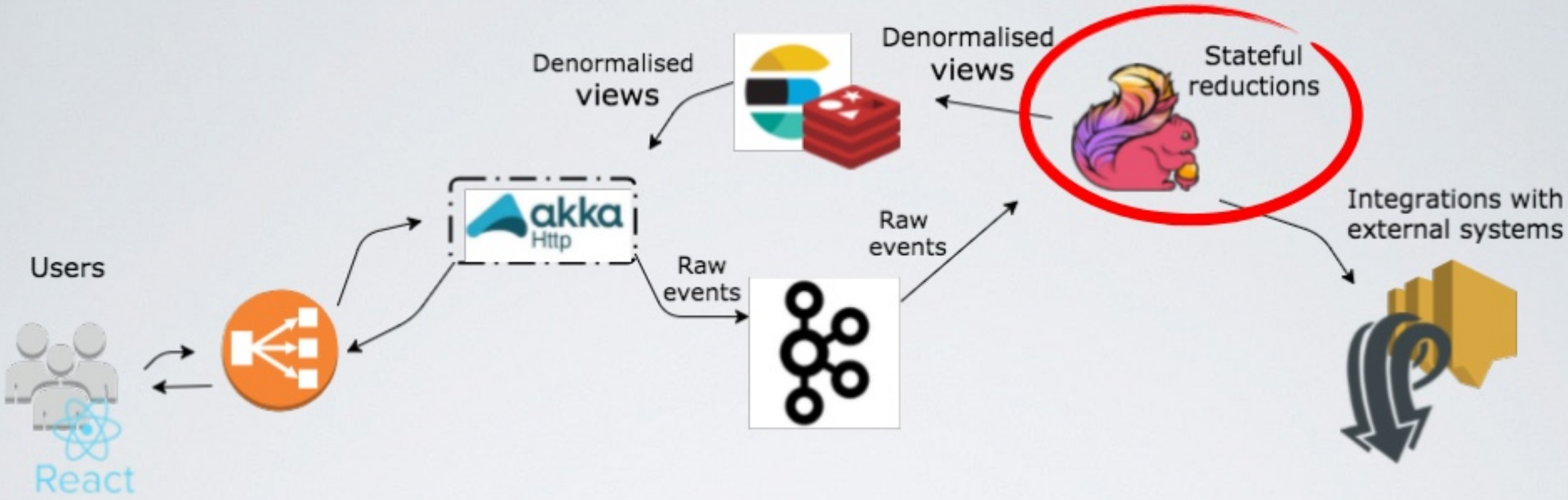
Attempt #1


Denormalised
views
Denormalised
views
Stateful
reductions
akka
Http
Integrations with
external systems
Raw
events
Raw
events
Users
React

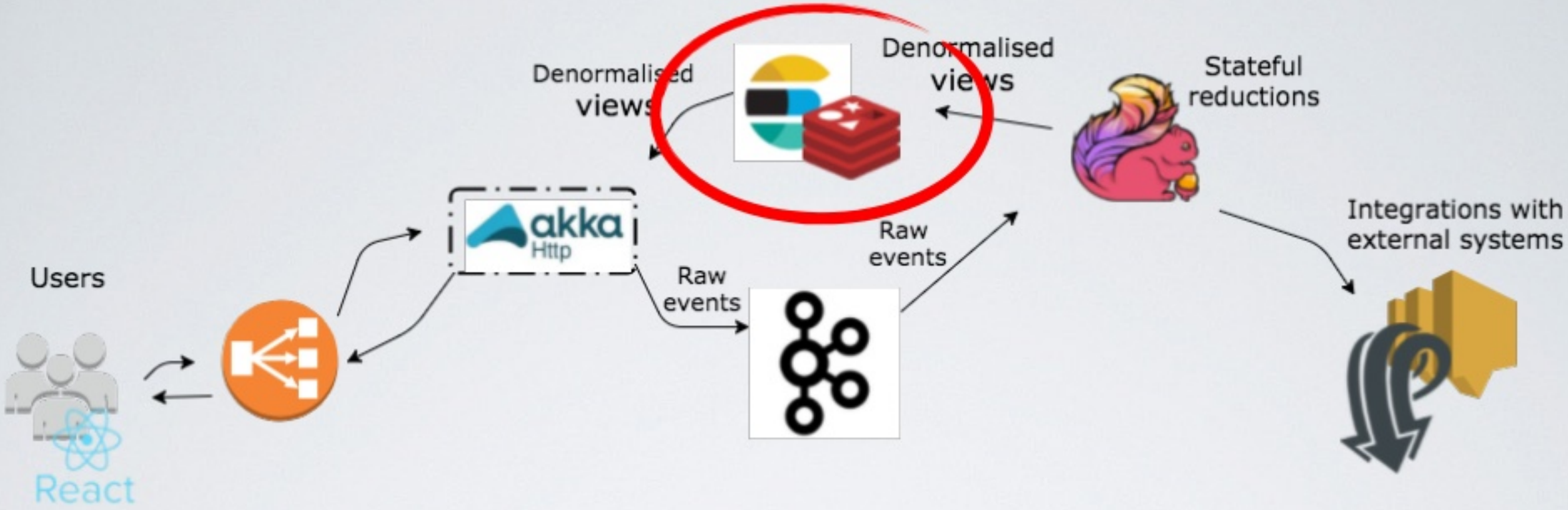
Attempt #1


Denormalised
views
Denormalised
views
Stateful
reductions
akka
Http
Raw
events
Raw
events
Integrations with
external systems
Users
React

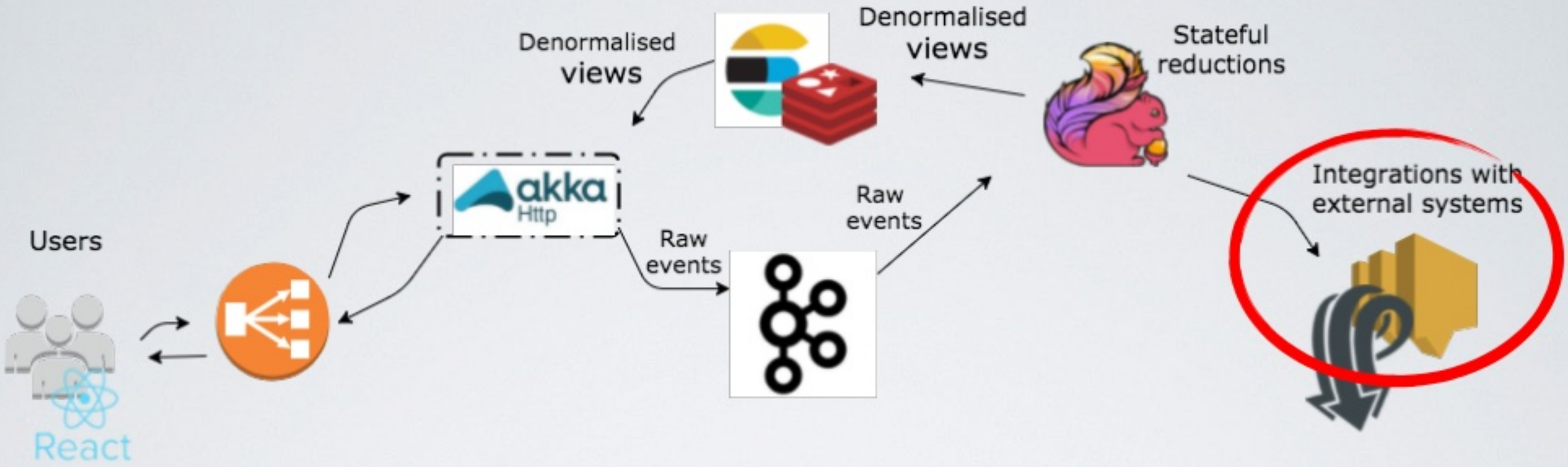
Attempt #1


Denormalised
views
Denormalised
views
Stateful
reductions
akka
Http
Raw
events
Raw
events
Integrations with
external systems
Users
React

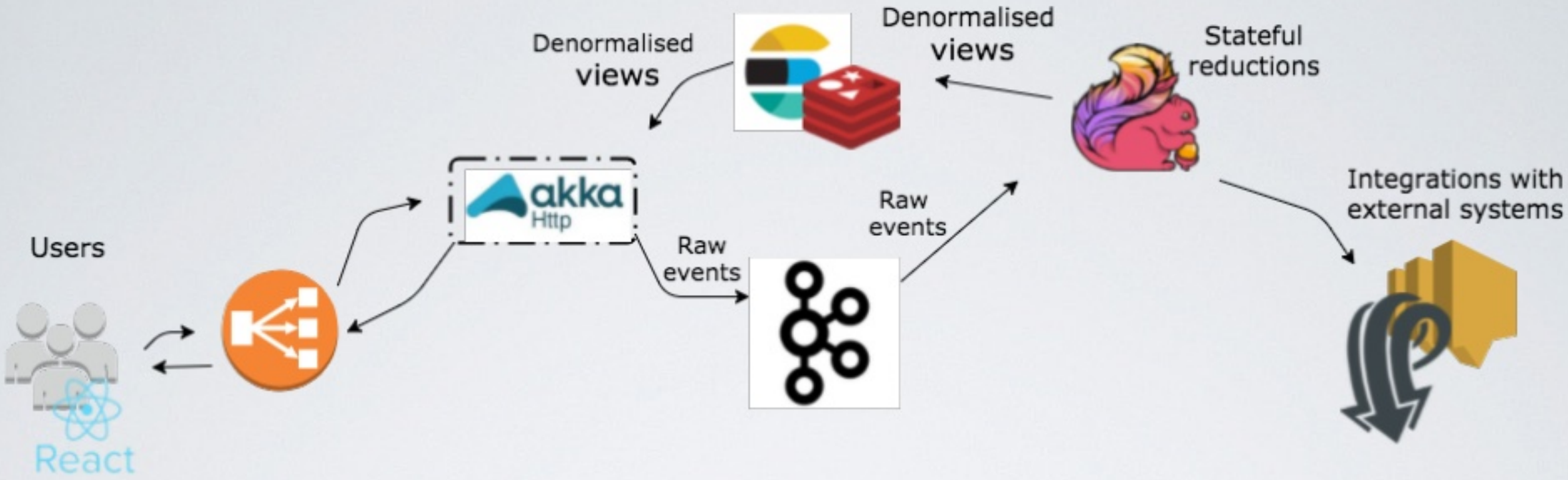
Attempt #1


Denormalised views
Denormalised views
Stateful reductions
akka Http
Users
Raw events
Raw events
Integrations with external systems
React

Attempt #1

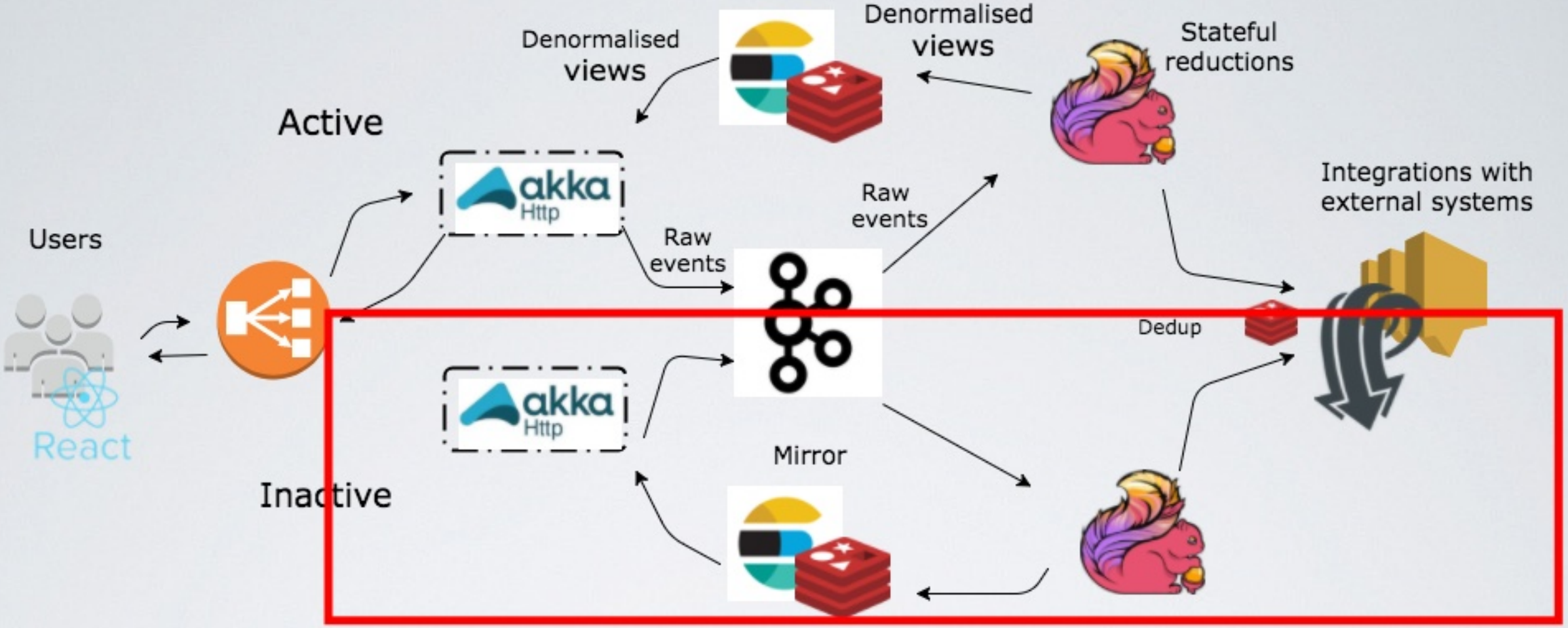

Denormalised
views
Denormalised
views
Stateful
reductions
akka
Http
Raw
events
Raw
events
Integrations with
external systems
Users
React

Attempt #1

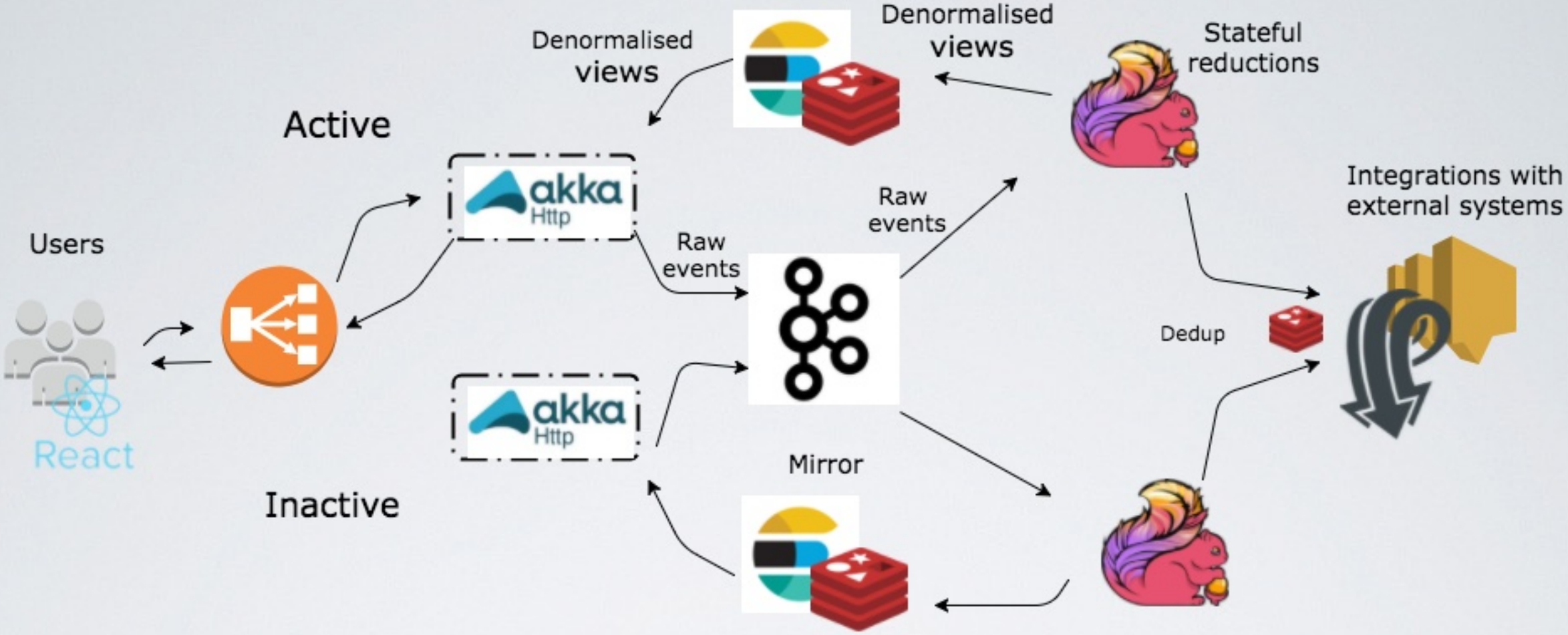


Attempt #1 - Issues

- Deployments. Savepoints not always possible.
- Upgrading components was difficult.

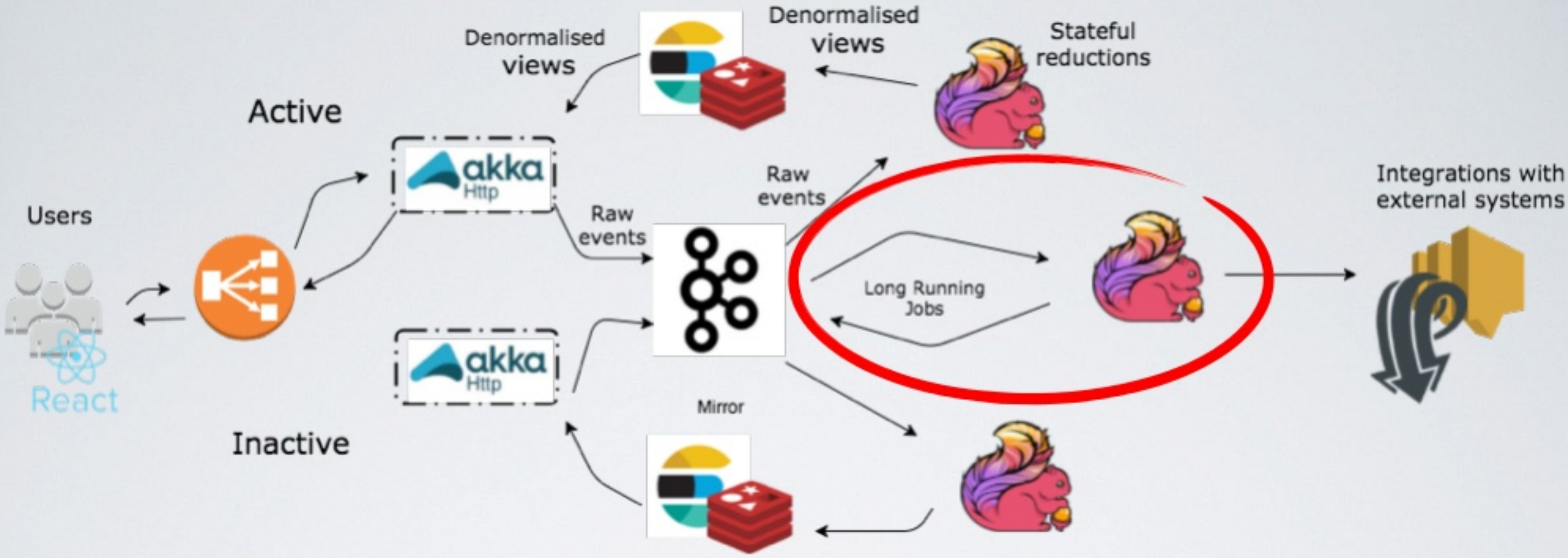


Attempt #2 - Black/White



Attempt #2 - Issues

- Messages to external integrations were duplicated.
- Many jobs are environment agnostic.


Denormalised views
Denormalised views
Stateful reductions
Active
akka Http
Raw events
Raw events
Integrations with external systems
Users
Long Running Jobs
akka Http
Mirror
Inactive
React

Winner

Is this really simplifying things?

Let’s look at some code…

Counting big numbers in real time

```scala
case class LikeEvent(userId: Id[User], postId: Id[Post], timestamp: DateTime)
case class LikeCounterState(postId: Id[Post], counter: HyperLogLog)
case class LikeCount(postId: Id[Post], count: Long)

implicit val likeCounterStateSemigroup = new Semigroup[LikeCounterState] {
  override def combine(x: LikeCounterState, y: LikeCounterState): LikeCounterState =
    LikeCounterState(x.postId, x.counter + y.counter)
}

val likeCounts: DataStream[LikeCount] = env.addTopicSource[LikeEvent](kafkaConfig)
    .map(like => LikeCounterState(like.postId, HyperLogLog.prepare(like.userId.value)))
    .keyBy(_.postId)
    .reduce(_ |+| _)
    .map(ls => LikeCount(ls.postId, ls.estimate))
```

Counting big numbers in real time

```scala
case class LikeEvent(userId: Id[User], postId: Id[Post], timestamp: DateTime)
case class LikeCounterState(postId: Id[Post], counter: HyperLogLog)
case class LikeCount(postId: Id[Post], count: Long)

implicit val likeCounterStateSemigroup = new Semigroup[LikeCounterState] {
  override def combine(x: LikeCounterState, y: LikeCounterState): LikeCounterState =
    LikeCounterState(x.postId, x.counter + y.counter)
}

val likeCounts: DataStream[LikeCount] = env.addTopicSource[LikeEvent](kafkaConfig)
    .map(like => LikeCounterState(like.postId, HyperLogLog.prepare(like.userId.value)))
    .keyBy(_.postId)
    .reduce(_ |+| _)
    .map(ls => LikeCount(ls.postId, ls.estimate))
```

Counting big numbers in real time

```
case class LikeEvent(userId: Id[User], postId: Id[Post], timestamp: DateTime)
case class LikeCounterState(postId: Id[Post], counter: HyperLogLog)
case class LikeCount(postId: Id[Post], count: Long)

implicit val likeCounterStateSemigroup = new Semigroup[LikeCounterState] {
  override def combine(x: LikeCounterState, y: LikeCounterState): LikeCounterState =
    LikeCounterState(x.postId, x.counter + y.counter)
}

val likeCounts: DataStream[LikeCount] = env.addTopicSource[LikeEvent](kafkaConfig)
    .map(like => LikeCounterState(like.postId, HyperLogLog.prepare(like.userId.value)))
    .keyBy(_.postId)
    .reduce(_ |+| _)
    .map(ls => LikeCount(ls.postId, ls.estimate))
```

Counting big numbers in real time

```scala
case class LikeEvent(userId: Id[User], postId: Id[Post], timestamp: DateTime)
case class LikeCounterState(postId: Id[Post], counter: HyperLogLog)
case class LikeCount(postId: Id[Post], count: Long)

implicit val likeCounterStateSemigroup = new Semigroup[LikeCounterState] {
  override def combine(x: LikeCounterState, y: LikeCounterState): LikeCounterState =
    LikeCounterState(x.postId, x.counter + y.counter)
}

val likeCounts: DataStream[LikeCount] = env.addTopicSource[LikeEvent](kafkaConfig)
    .map(like => LikeCounterState(like.postId, HyperLogLog.prepare(like.userId.value)))
    .keyBy(_.postId)
    .reduce(_ |+| _)
    .map(ls => LikeCount(ls.postId, ls.estimate))
```

Adding a time element

```
val likeCountWindow: DataStream[LikeCount] = env.addTopicSource[LikeEvent](kafkaConfig)
  .map(like => LikeCounterState(like.postId, HyperLogLog.prepare(like.userId.value)))
  .keyBy(_.postId)
  .timeWindow(Time.hours(24), Time.minutes(2))
  .reduce(_ |+| _)
  .map(ls => LikeCount(ls.postId, ls.estimate))
```

There is a learning curve

But this ends up being pretty

Simple.

Q&A

FIN