Designing Systems that Age like Wine

"Somm" Lagom

Kiki, Principal Architect @ lightbend



Let’s go on a short journey





Why Wine?

because wine ages better than leftovers

Systems bring economies to life.



Rapid Rate of Change
gives rise to Rapid Aging



Take a look at the differences between aging architectures





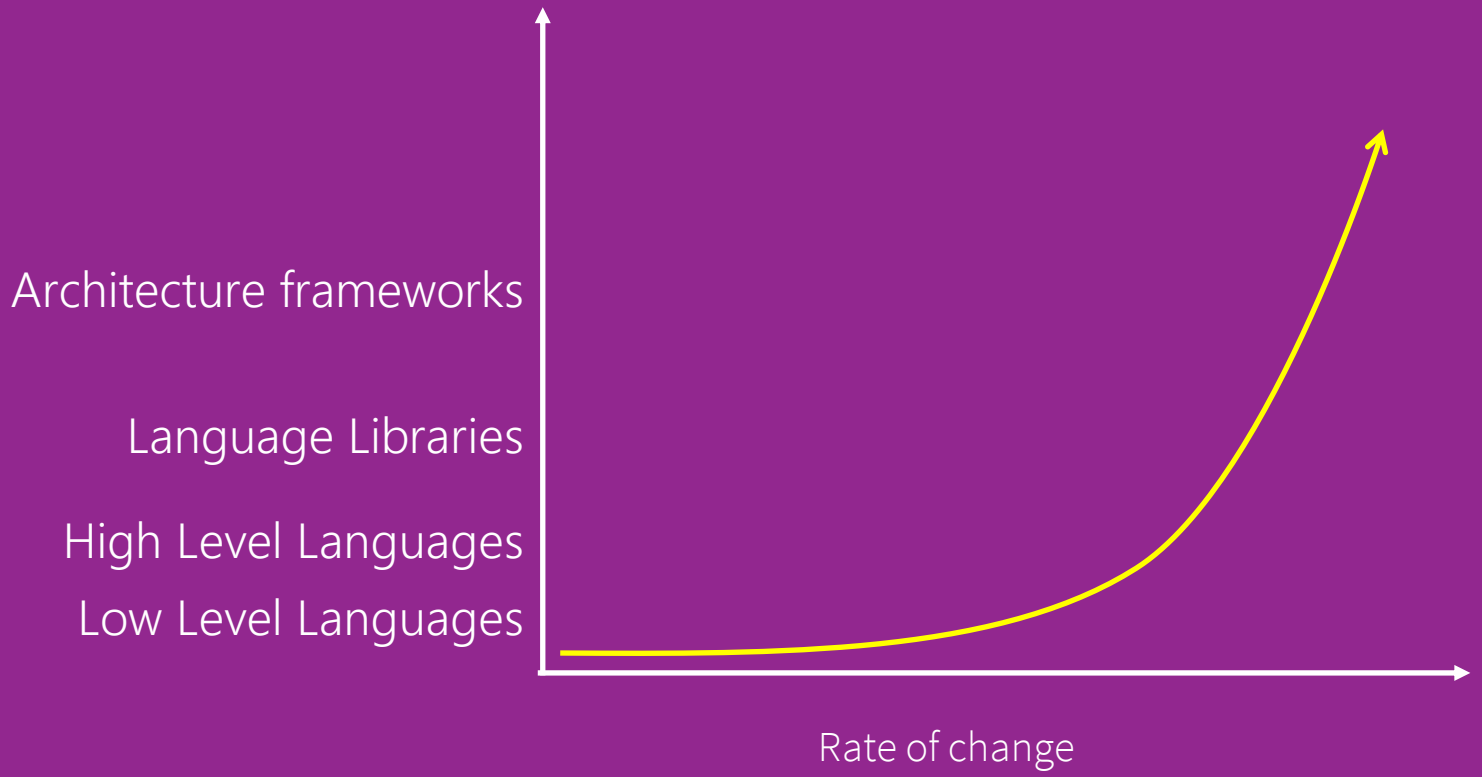
Aya Sofia and ….this thing

Architecture Evolution

Our tools have evolved over time to help us meet the demands

Apply machine instructions	Apply machine instructions using richer expression through data structures	Apply common coding patterns or algorithms to data structures for solving problems	Apply Architectural Patterns to or within an Application
Low level languages	High level languages	libraries	frameworks
Assembly	Python Java C	Java Collections PyMath	MVC MV*

Rate of change vs levels of abstraction



Apply machine instructions

Apply machine instructions using richer expression

Apply common coding patterns or algorithms to data structures

We've reach a level of evolution beyond application level frameworks

Apply Architectural Patterns to or within an Application

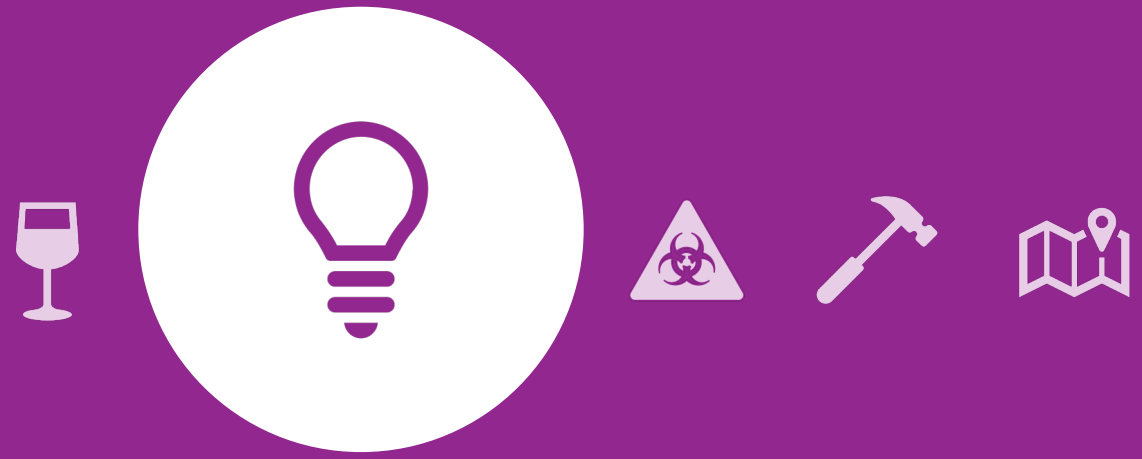
frameworks

MVC
MV*

Assembly

Python
Java
C

Java Collections
PyMath



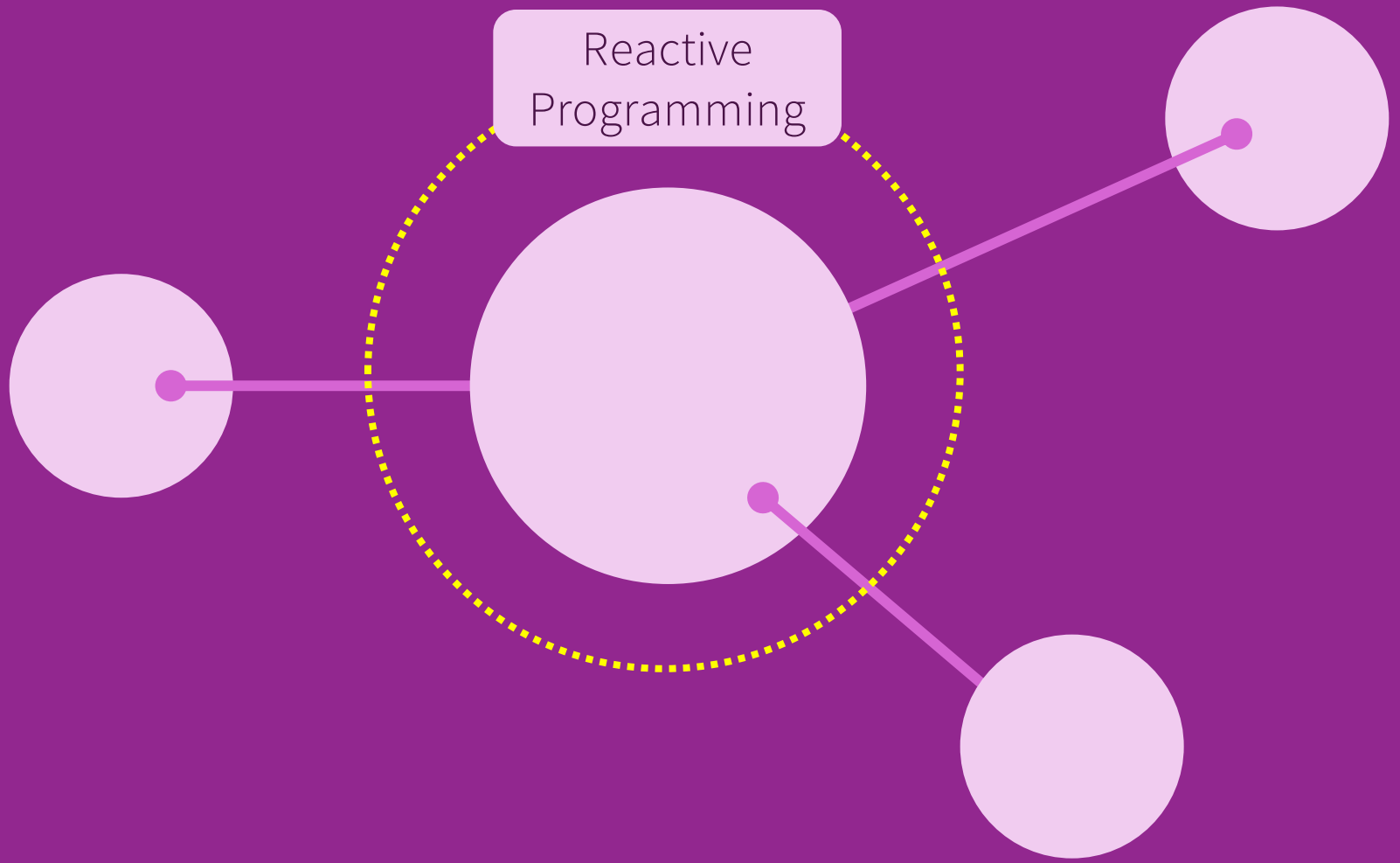
Systems are the next level of evolution

This landscape of microservices should cause us to think system first

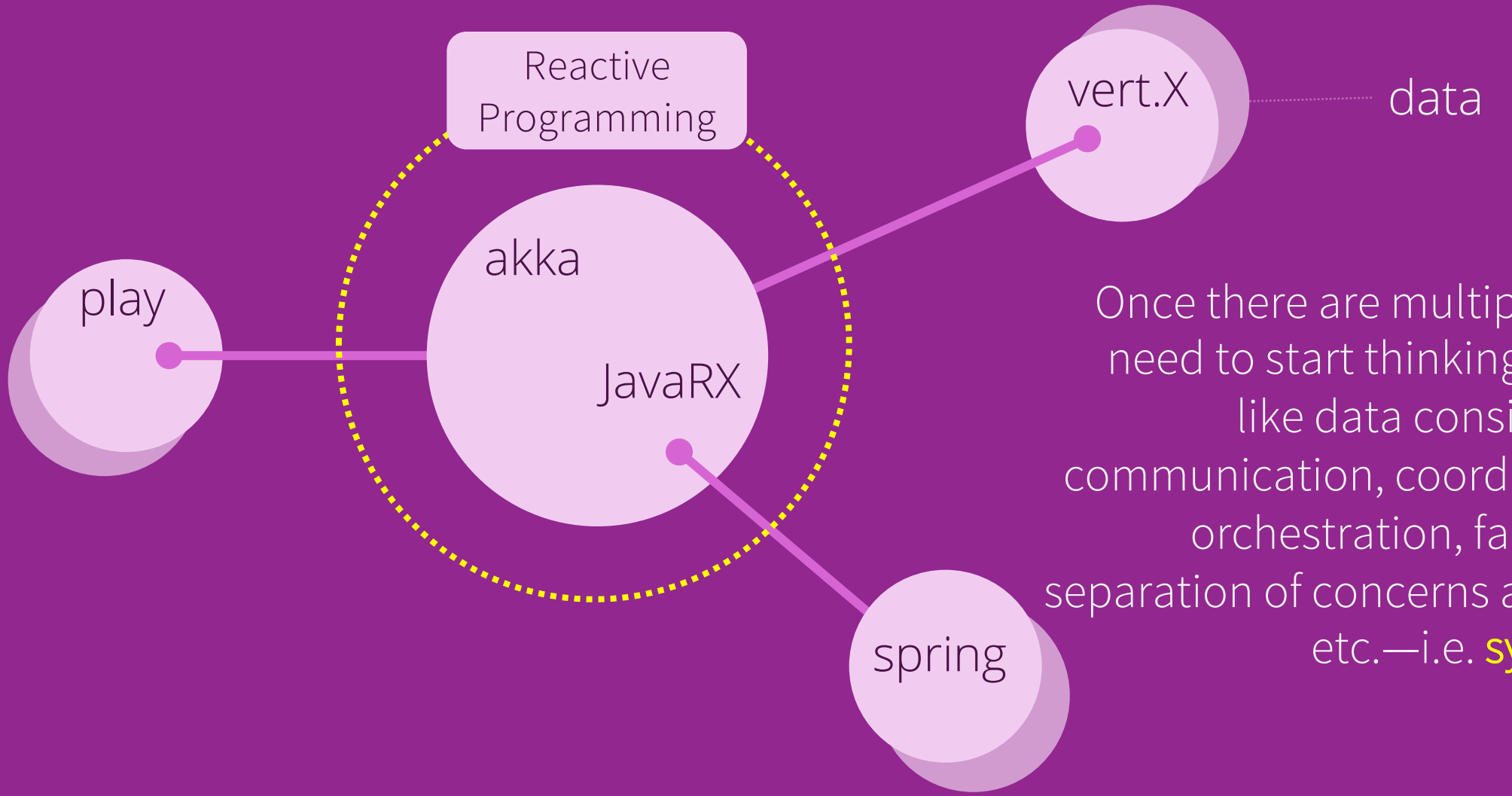
Apply Architectural Patterns to A System of Services or Applications

Distributed computing, Distributed asynchronous systems have given rise to specific challenges met by specific patterns – system building patterns.

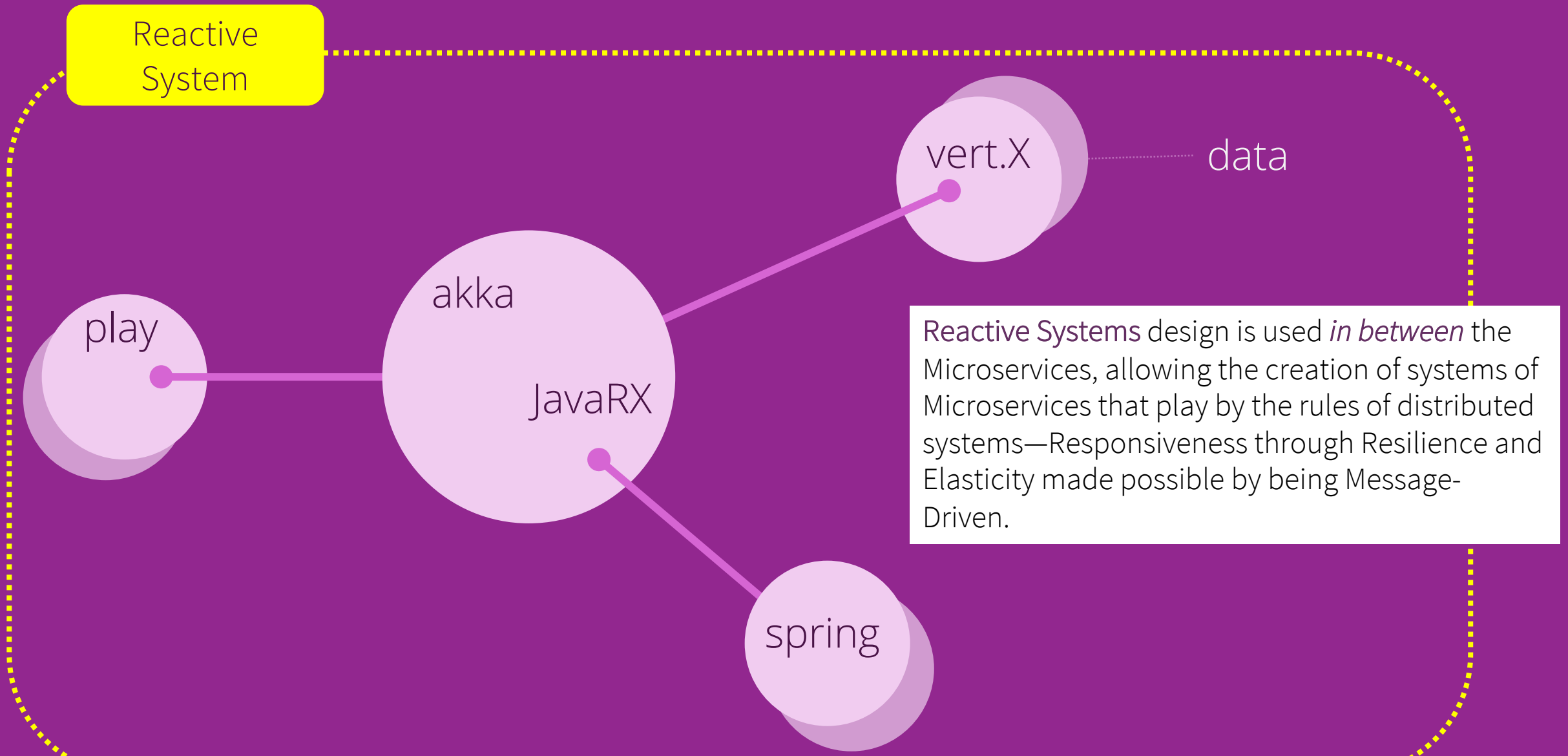
Reactive Programming & Reactive Systems



Reactive Programming is used within a *single* micro*service* to implement the service-internal logic and dataflow management.



Once there are multiple nodes, there is a need to start thinking hard about things like data consistency, cross-node communication, coordination, versioning, orchestration, failure management, separation of concerns and responsibilities etc.—i.e. system architecture.



Reactive Systems design is used *in between* the Microservices, allowing the creation of systems of Microservices that play by the rules of distributed systems—Responsiveness through Resilience and Elasticity made possible by being Message-Driven.



lagom

Microservice system framework

Thin, powerful abstraction over



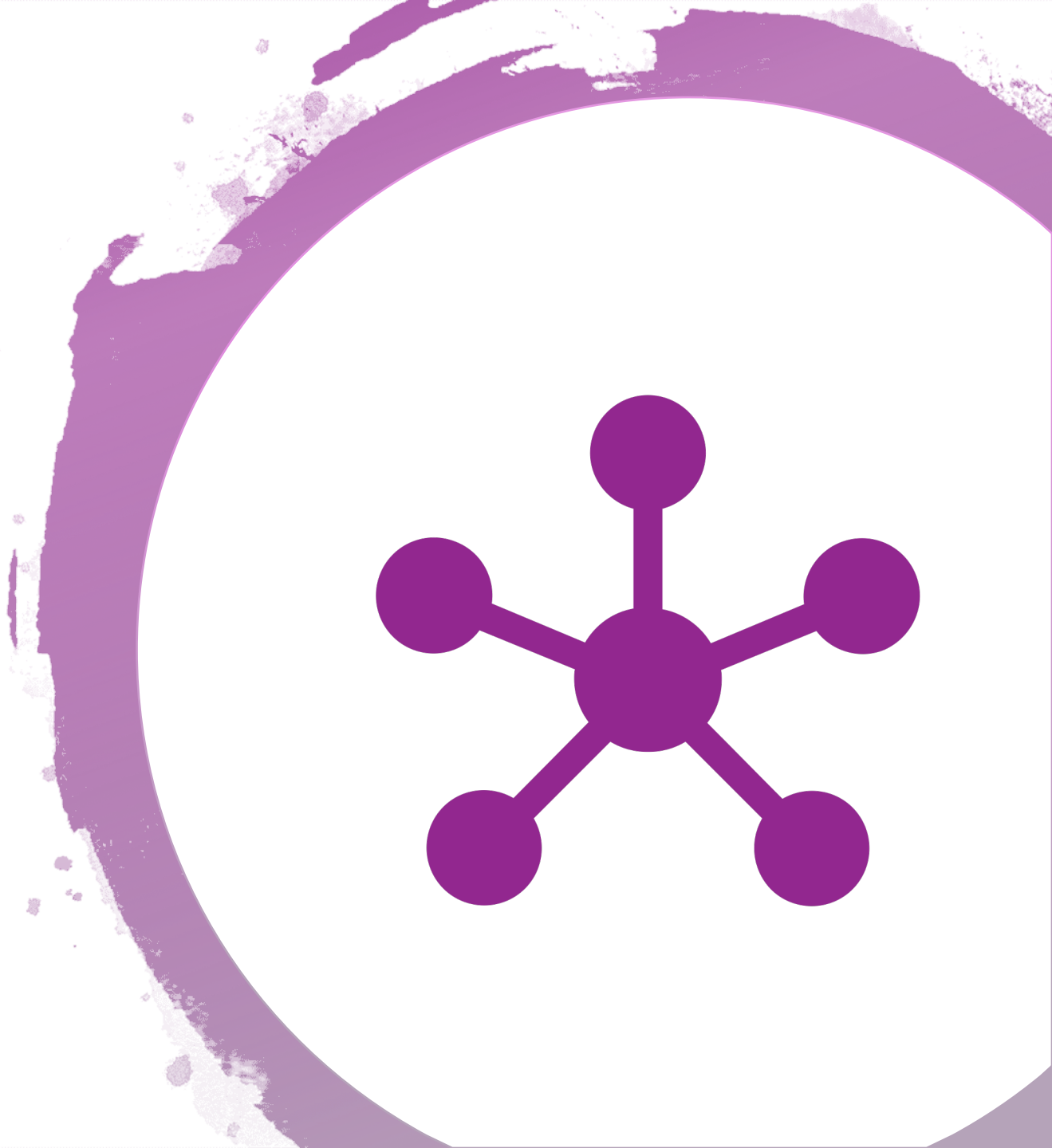
Web Application Framework



Concurrency Toolkit \ Runtime

Other curated tools, architectural principles and patterns used
for building reactive microservice systems

Lagom uses asynchronous messaging, asynchronous IO & distributed persistence patterns





There are challenges when trying to build distributed systems

The Overall Challenge

Adapt to rapid rate of change & Maintain Architectural Integrity

Too many choices – Analysis Paralysis



Lagom makes reasonable choices for you.

It’s built on message-driven technology that’s highly responsive, elastic and fault-tolerant.

It’s Reactive.

Lagom Components

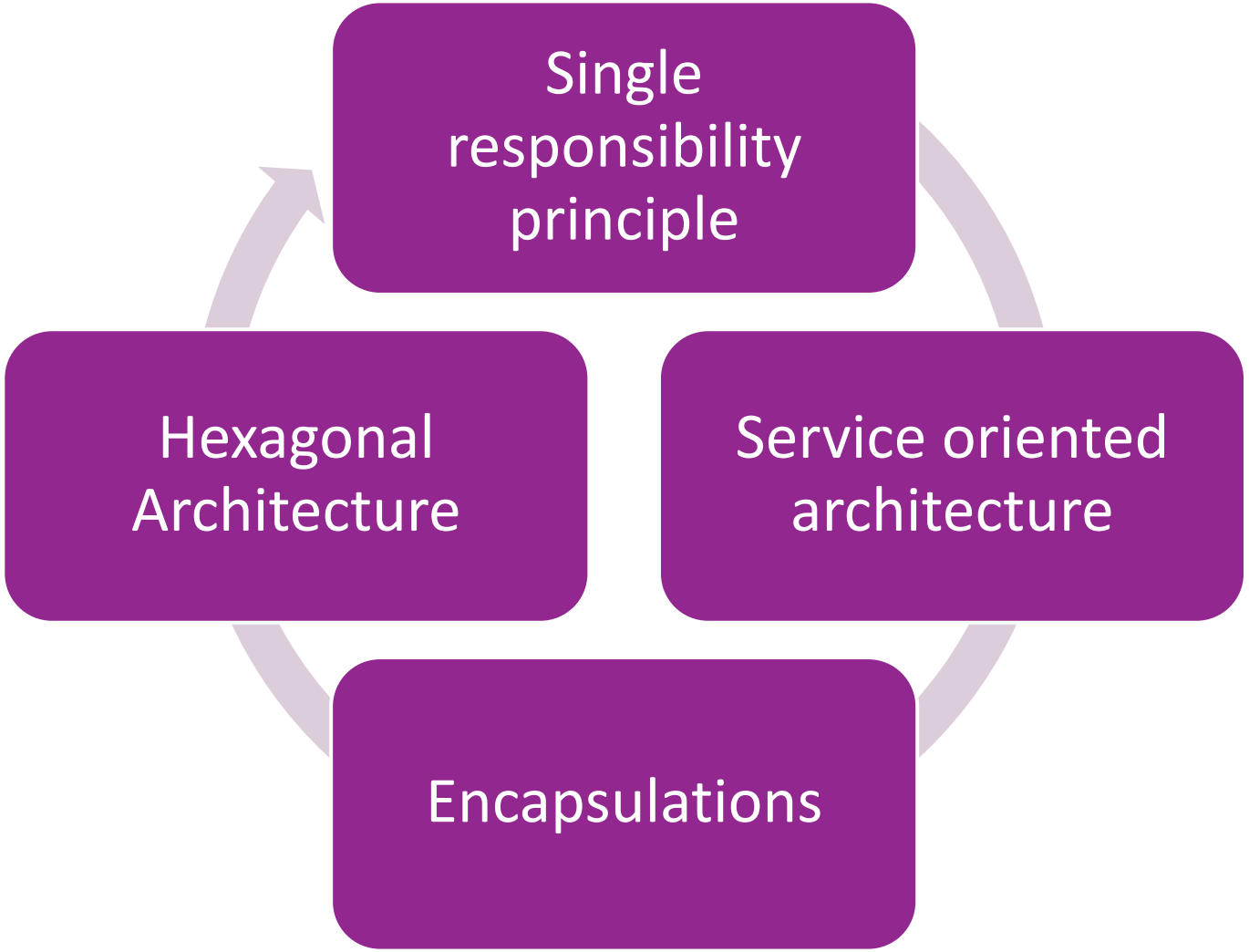
- **Akka** – Persistence, PubSub, Cluster, Streaming Async Services
- **Play Framework** – Web framework
- **Cassandra** – managing data persistence
- **Guice** – dependency injection
- **SLF4J & Logback** – Logging
- **Typesafe Config** – configuration
- **JSON Serialization** – Play JSON (Scala) Jackson (Java)
- **Message Broker** - Kafka

Difficult to scale architectural integrity




Lagom consistently
applies patterns for
Reactive System Building

Architectural Principles Enforced by Lagom



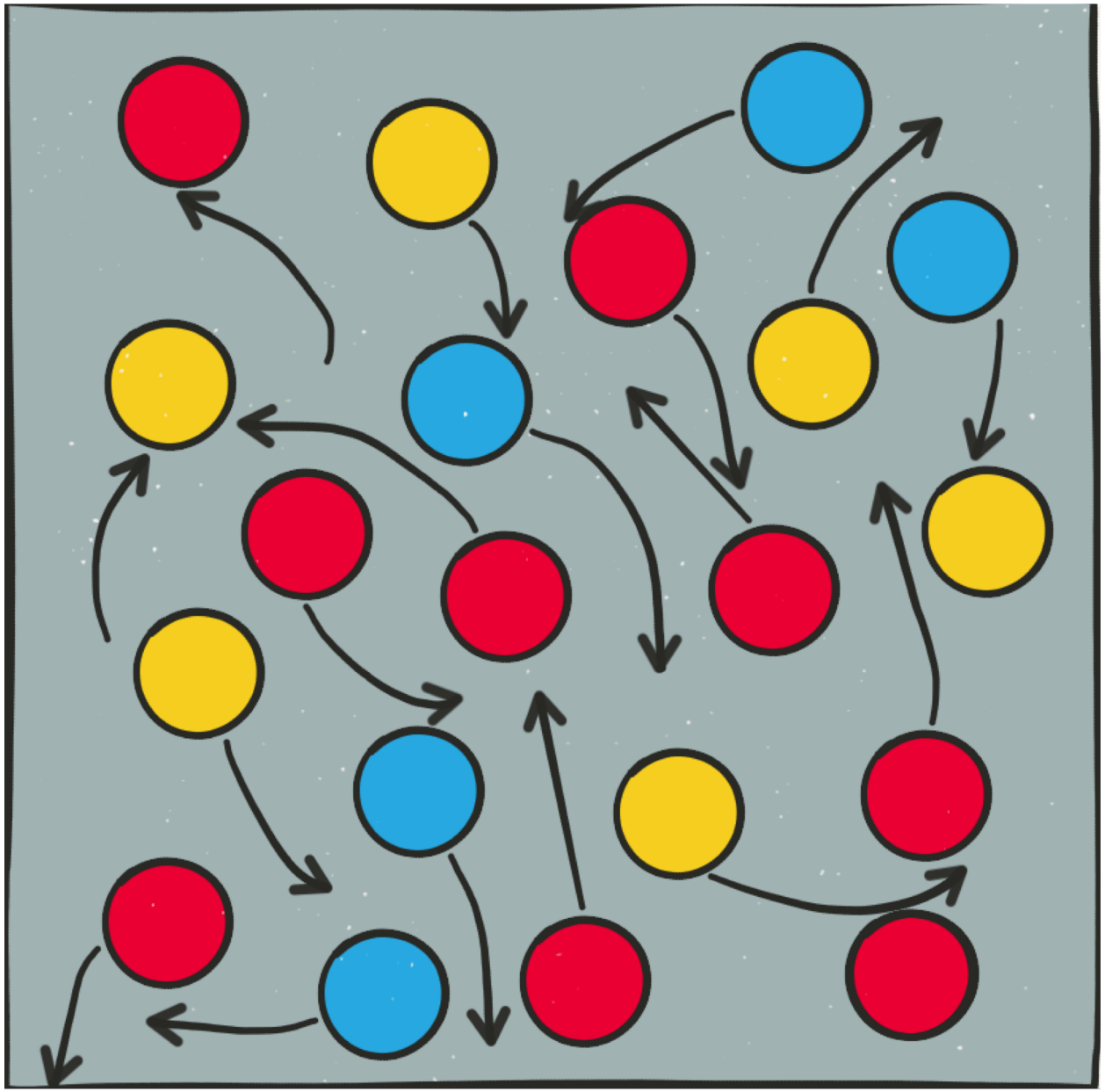
Lagom Design Patterns

- Asynchronous & non-blocking
- Eventual consistency
- Event sourcing
- Domain driven design
- CQRS
- Service locator

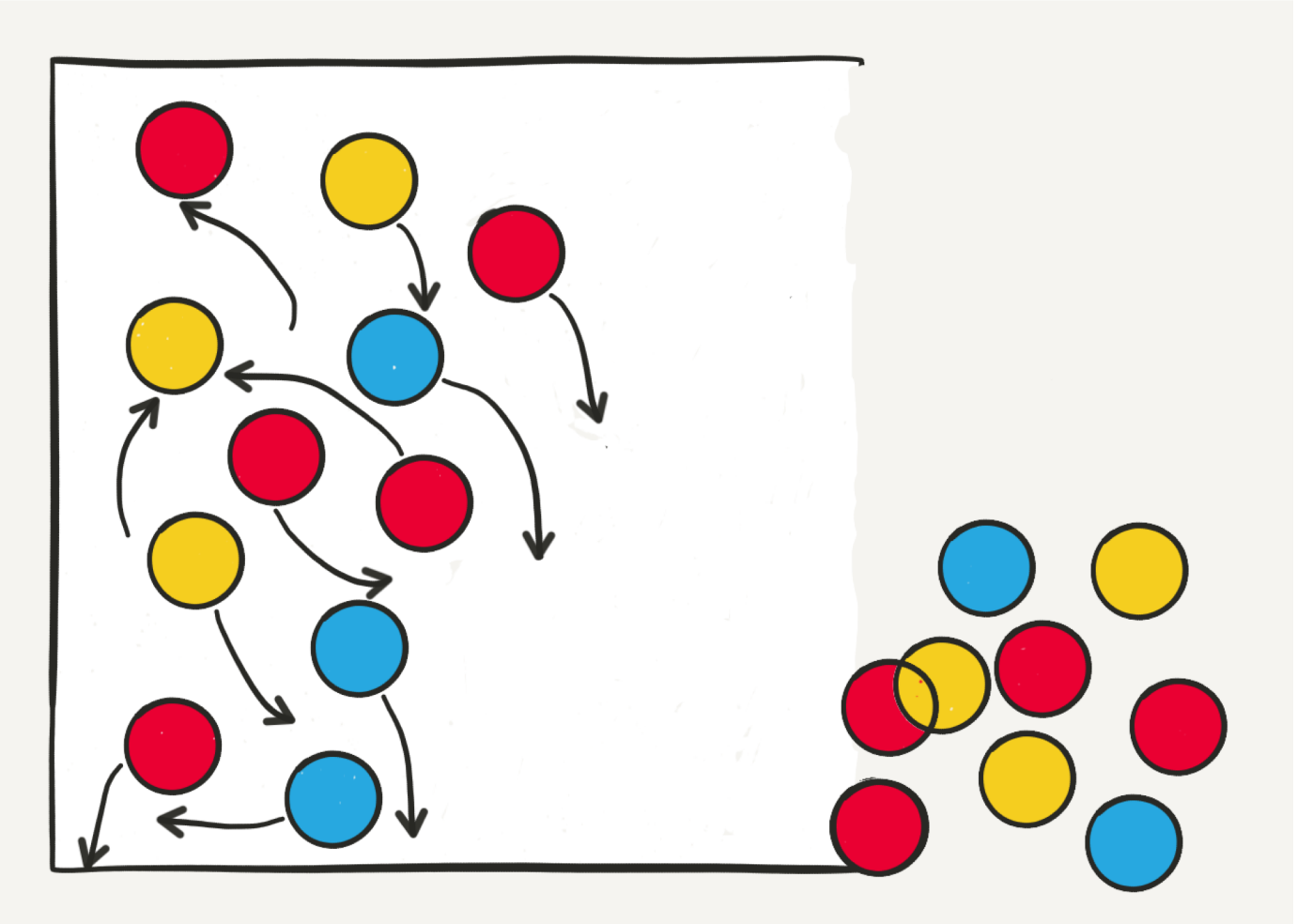
Unplanned
complexity or chaos



Monolith with many moving parts.



You cannot simply break the seal and expect everything to function correctly

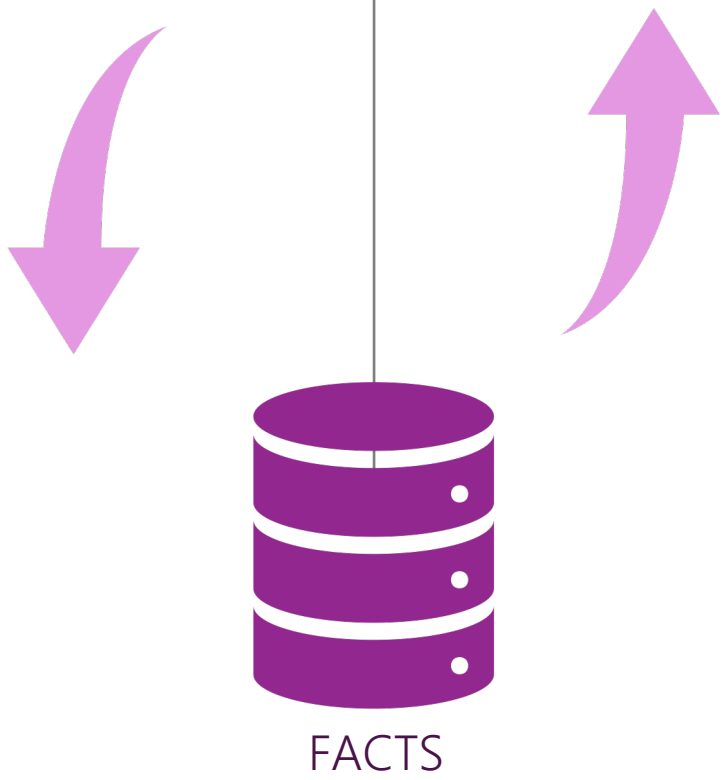


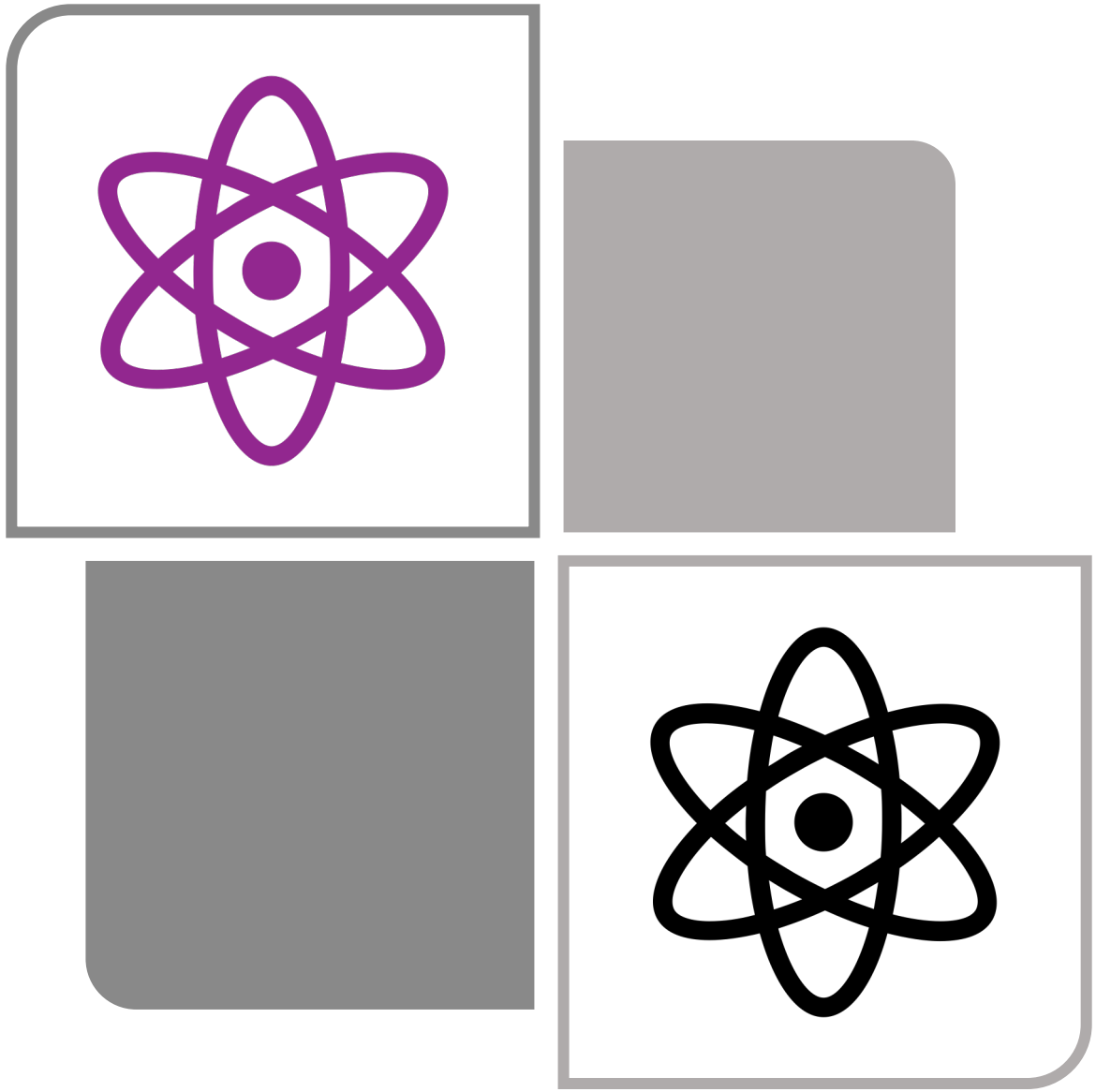
Lagom minimizes chaos through isolation, autonomous services



Promote Isolation & Autonomy by Separating **Facts** from **Views**

- **Facts** – actions that change the state of an entity
- **Views** – fact-based interpretations, presentation, visualization





In distributed systems

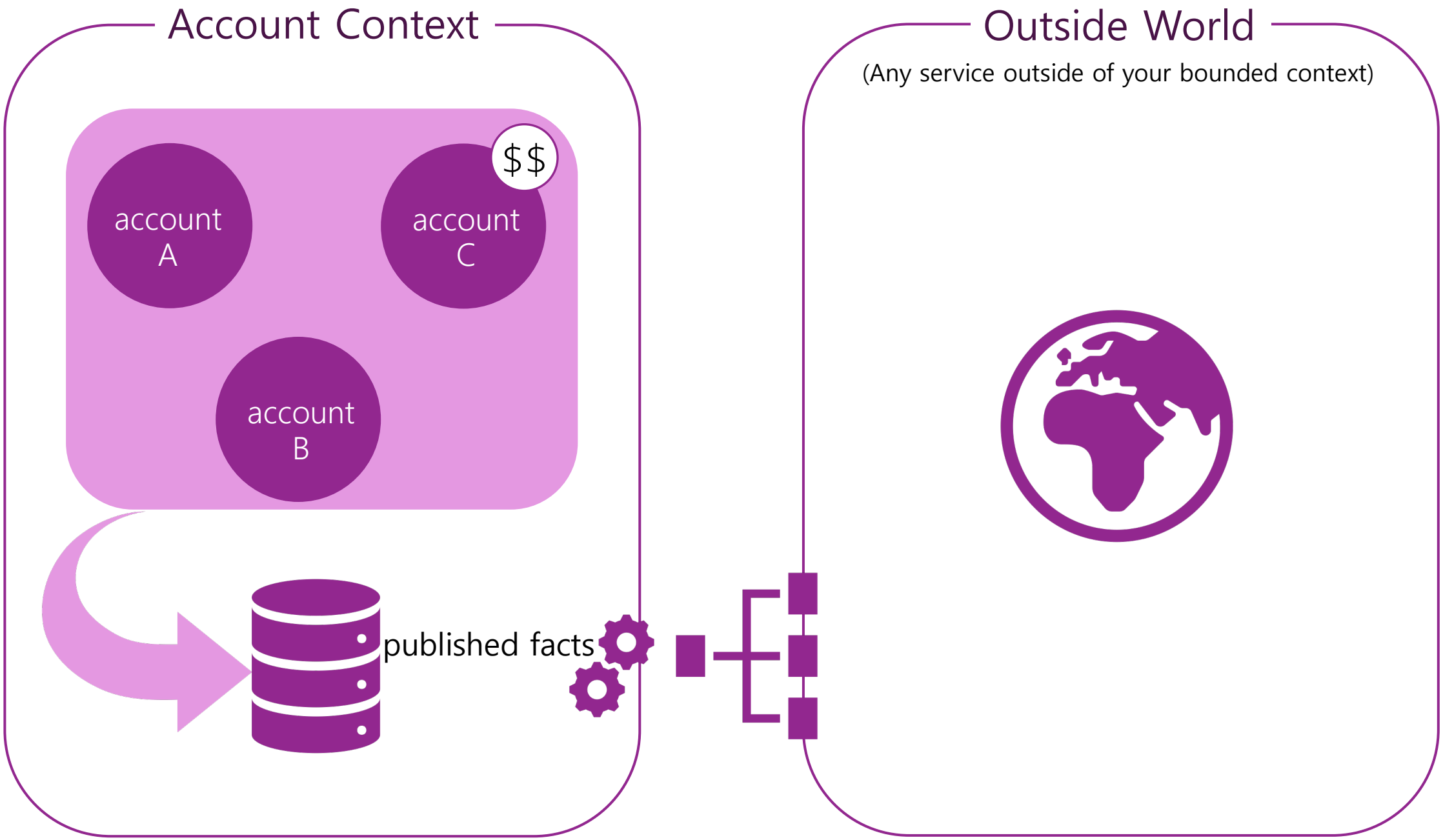
- Apparent state
- Interpreted state

Is a DB or Distributed Cache the right place for State to make sure you don’t lose it?



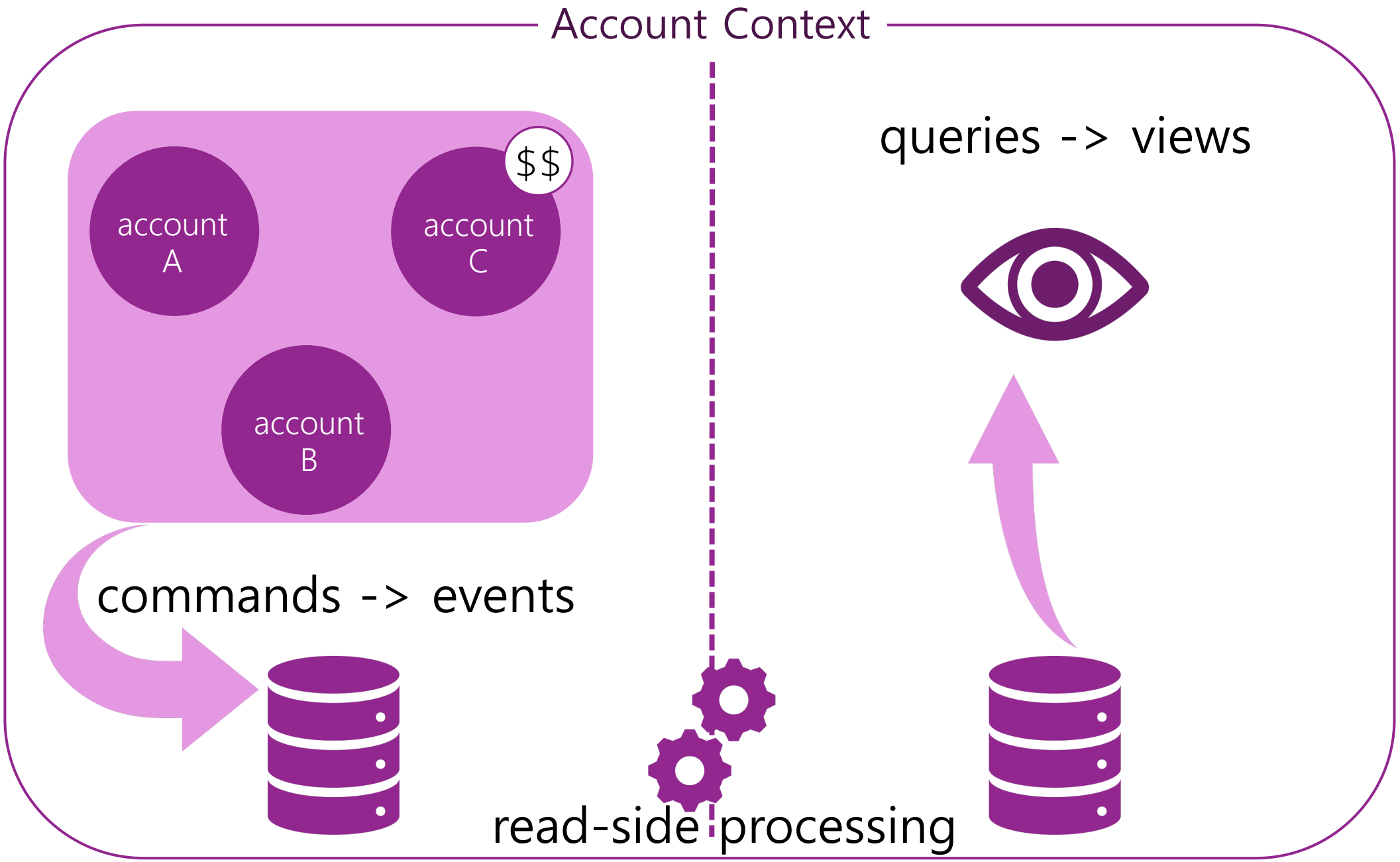
Lagom Encourages and Enables You to Contain Mutable State & Publish Facts




Account Context
account A
account C
$$
account B
published facts
Outside World
(Any service outside of your bounded context)

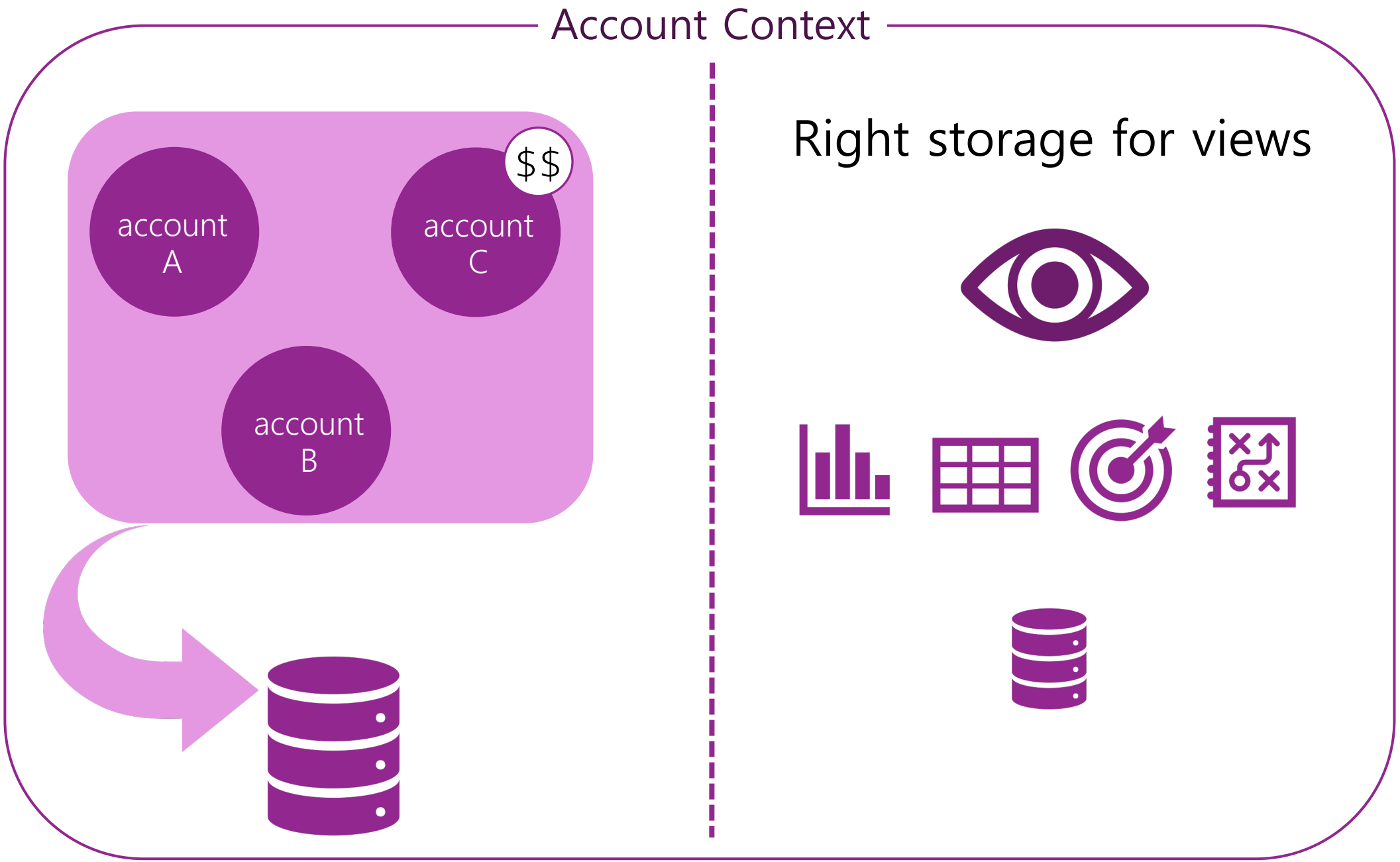
Lagom Allows You to Untangle Read & Write Models with CQRS & Event Sourcing




Account Context
account A
account C
$$
account B
queries -> views
commands -> events
read-side processing

Inflexible Data Models that prevent proper management of State



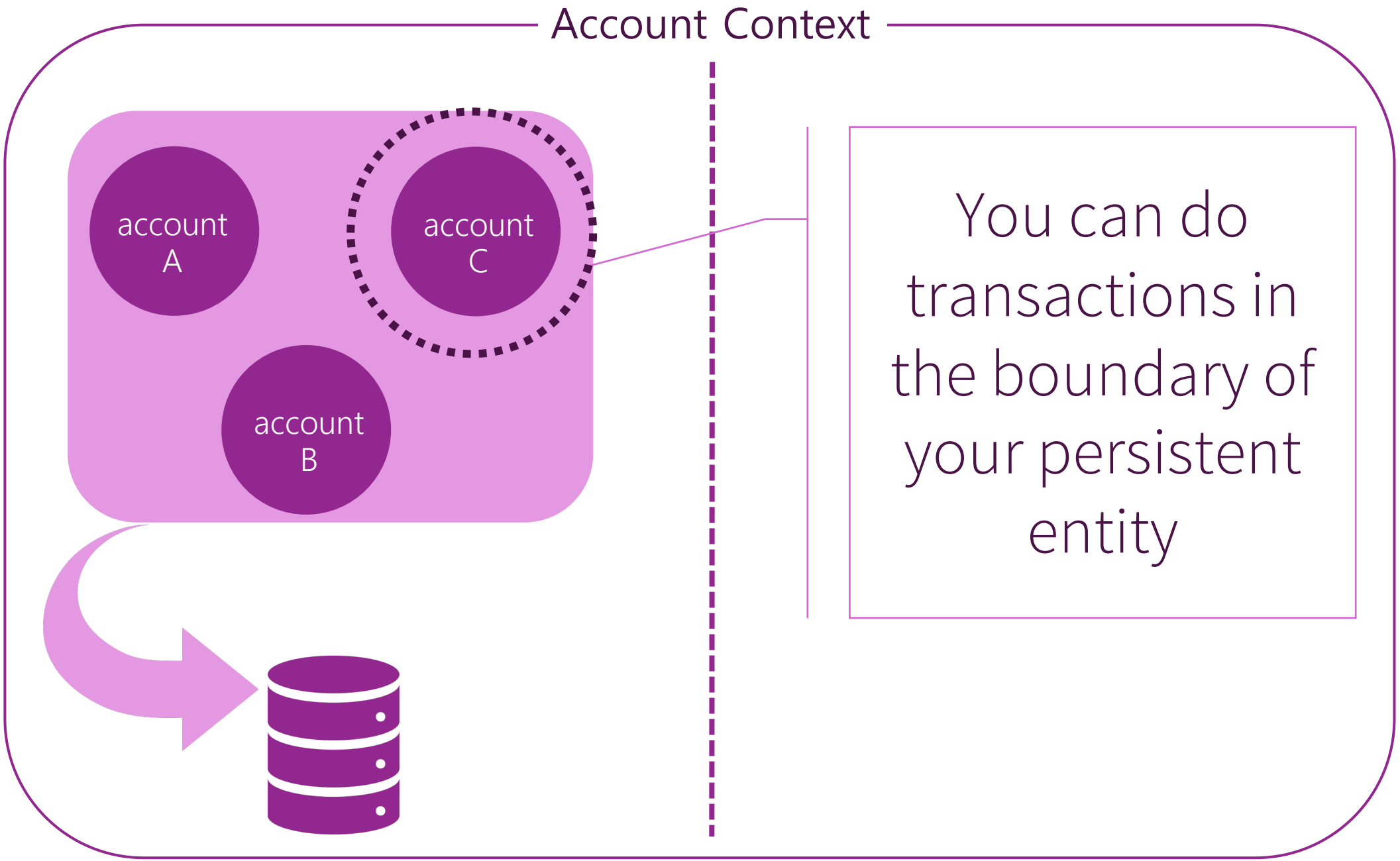

Account Context
account A
account C
$$
account B
Right storage for views

Maintaining the illusion of now



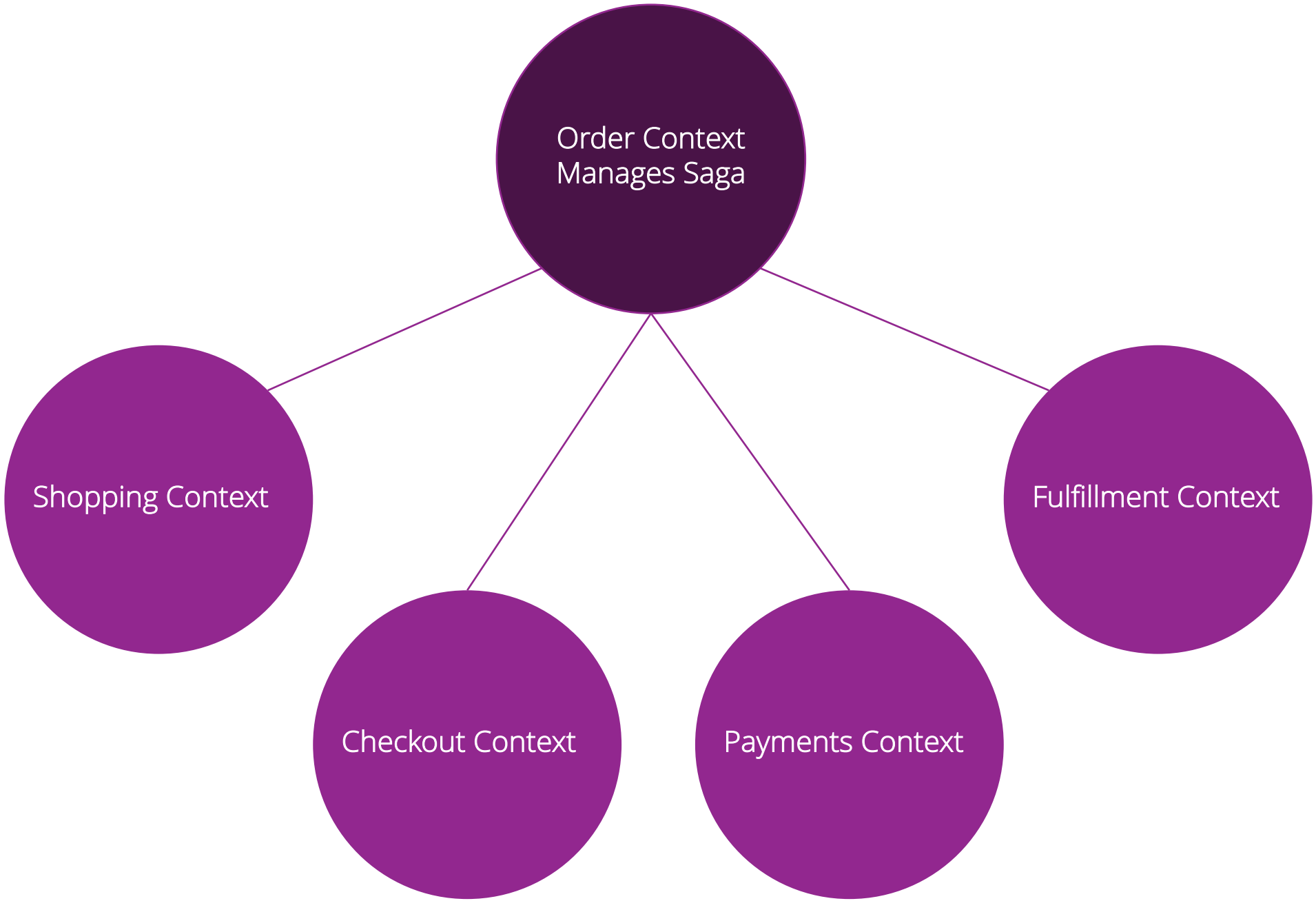
“we’ve always used transactions”

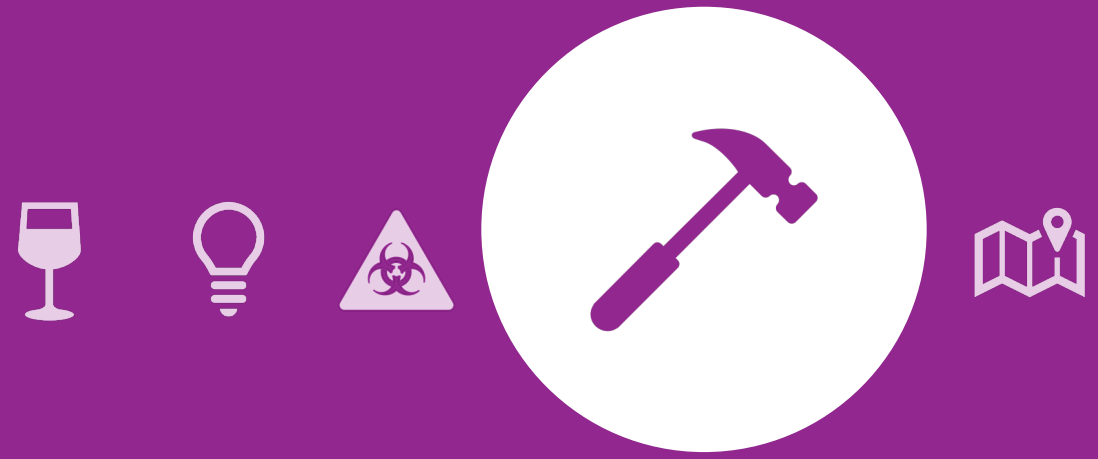



Account Context
account A
account C
account B
You can do transactions in the boundary of your persistent entity

“We need transactions outside the bounded context”




Order Context
Manages Saga
Shopping Context
Checkout Context
Payments Context
Fulfillment Context



Can we build something meaningful given these challenges?

Experts Required



Experts NOT! Required



You don't need "Lagom developers" just Scala or Java developers willing to apply Lagom semantics.

Must protect developer productivity





Practical Productivity

- *Scala & java API*
- *Use any IDE*
- *Quick start w/ ”batteries included”*
- *Hot reload & single command to “run all”*

All or Nothing
This & Everything

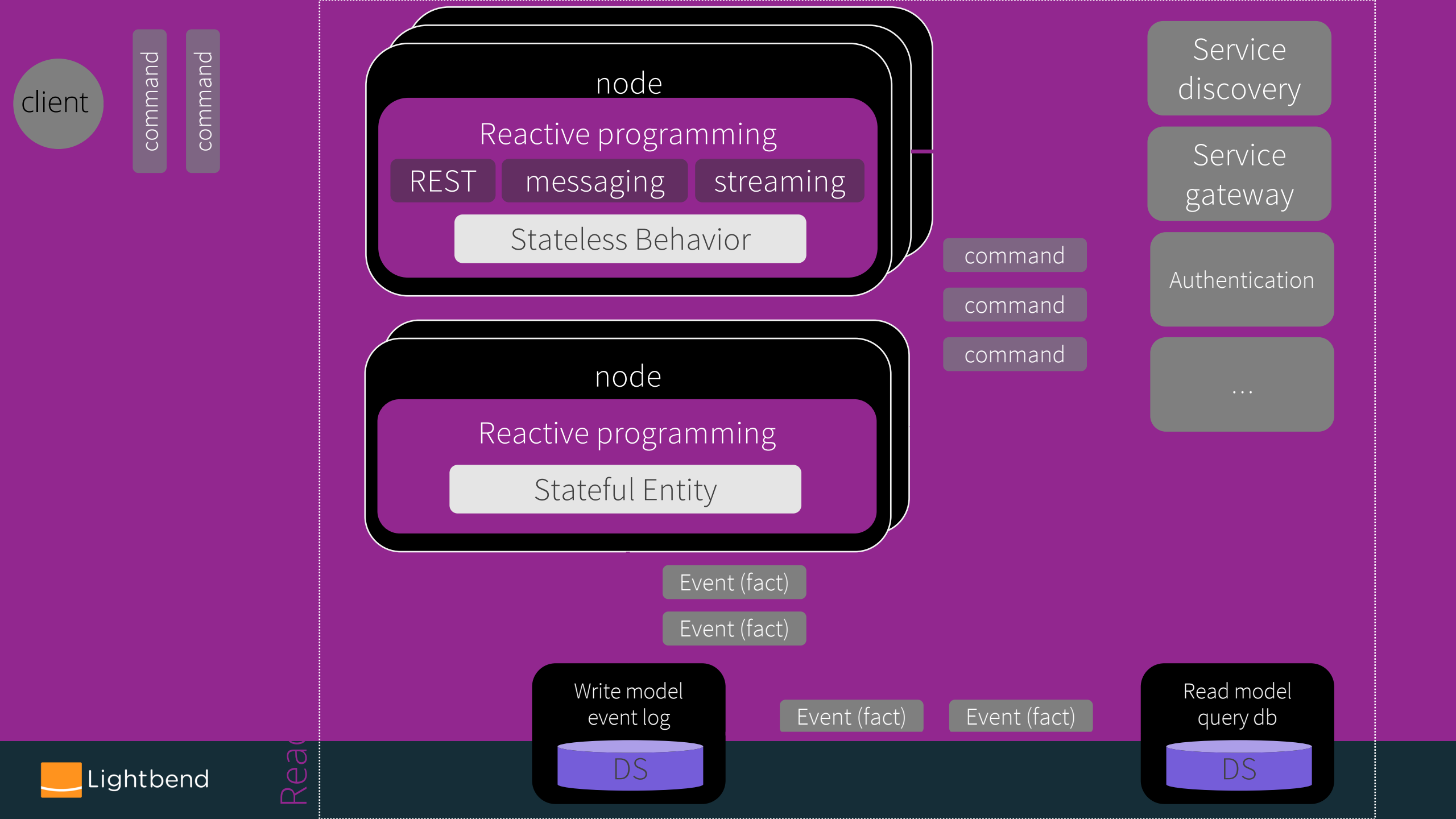


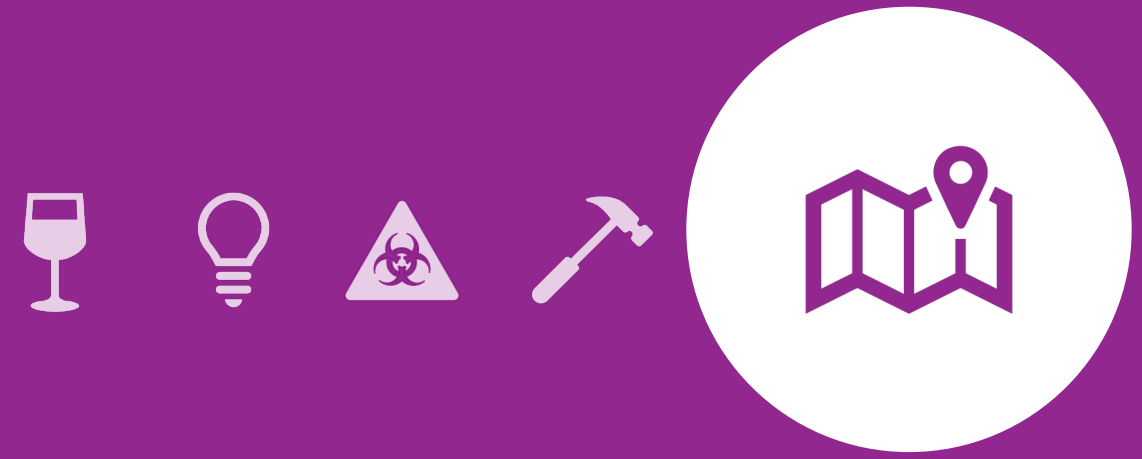




Lagom & your Legacy

- *Language agnostic communication with external services*
- *Safe Interaction with legacy systems via circuit breakers*
- *No need to tear everything down and start over*
- *Embraces polyglot systems*


client
command
command
node
Reactive programming
REST
messaging
streaming
Stateless Behavior
node
Reactive programming
Stateful Entity
Service discovery
Service gateway
Authentication
...
command
command
command
Event (fact)
Event (fact)
Write model event log
DS
Event (fact)
Event (fact)
Read model query db
DS
Lightbend



Where do we go from here?

Let it Change!

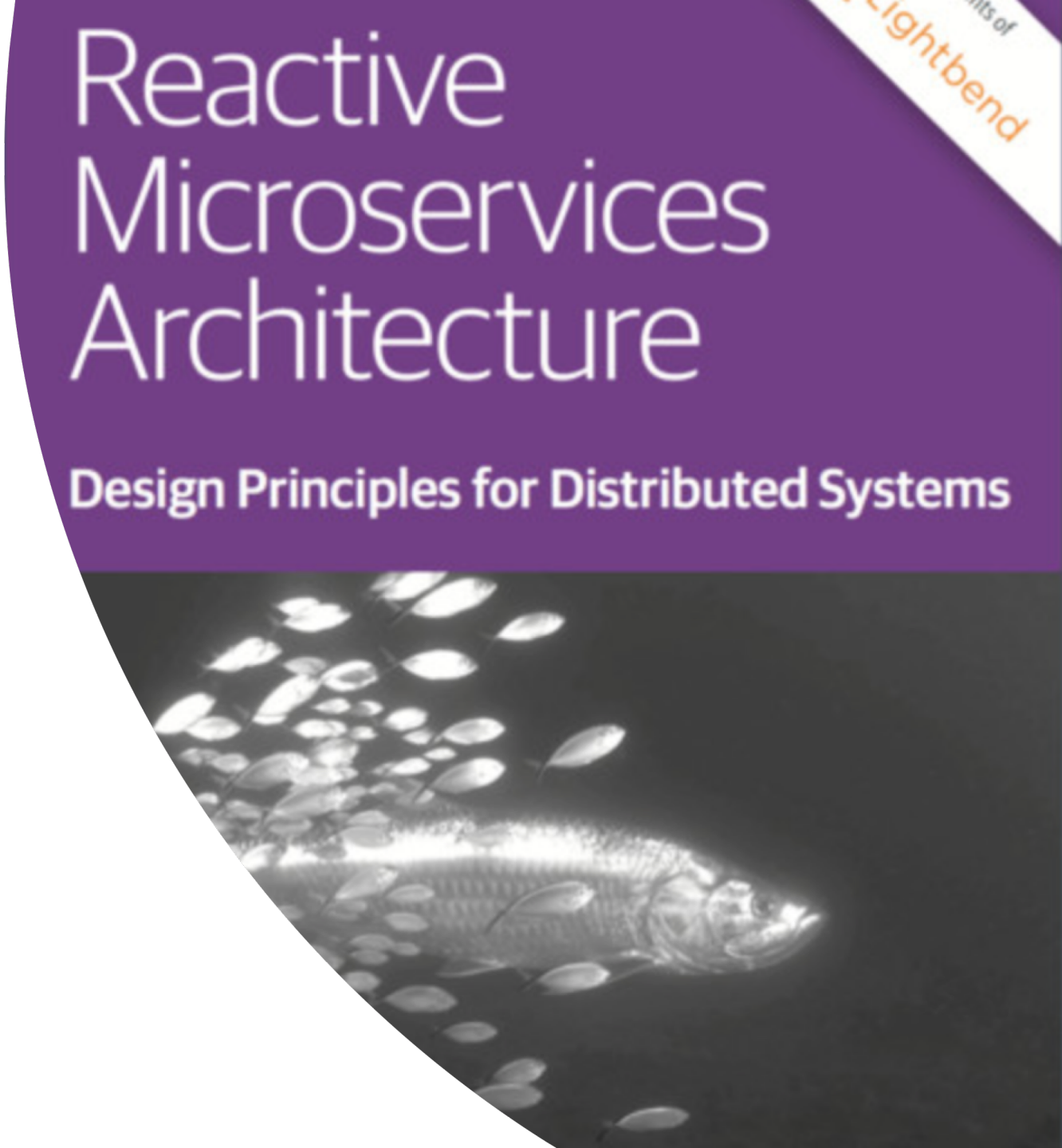


The Takeaway…

- Address the needs of distributed systems
- Address those needs with architectural integrity
- You must scale Integrity
- You must embrace – let it change!

The Takeaway…

- Build a TempleCathedralMosque
- www.lagomframework.com , https://github.com/lagom
- Ask me for a demo:
  - kiki@lightbend.com
  - Twitter: @kikisworldrace





Questions?