

Events Everywhere

breaking monoliths and silos



Cédric Pontet
@cpontet

Who is this guy ?

What does he do ?

How does he dare stand in front of us today and speak about such grave things as system architectures and monoliths and coupling ?



Let's speak about you instead

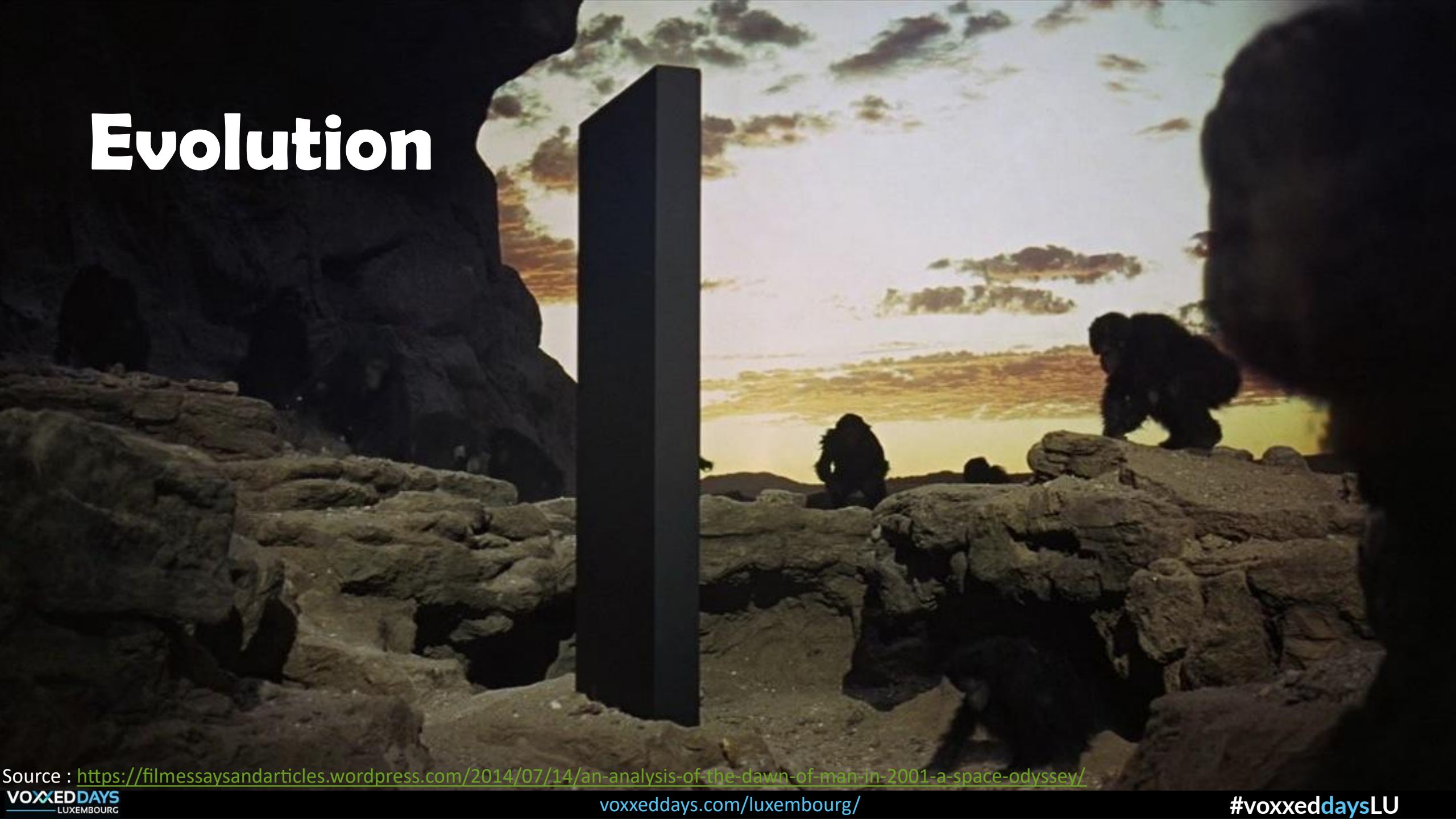


Signed CommitStrip album to win

Game ends at 5 PM



Evolution



Source : <https://filmessaysandarticles.wordpress.com/2014/07/14/an-analysis-of-the-dawn-of-man-in-2001-a-space-odyssey/>

**What is so bad about a
monolith after all ?**

The next step of evolution

Go buy yourselves an

Enterprise	Expensive
------------	-----------

Service	Shit
---------	------

Bus	Ball
-----	------

and all your problems will be solved



Any idea what SOA means in Dutch ?

The holy relational database

Every problem is a nail.
The only tool you need is a hammer.



News flash
RDBMS vendors tell lies

The new ~~big~~ idea : μ Services small

There must a
better way

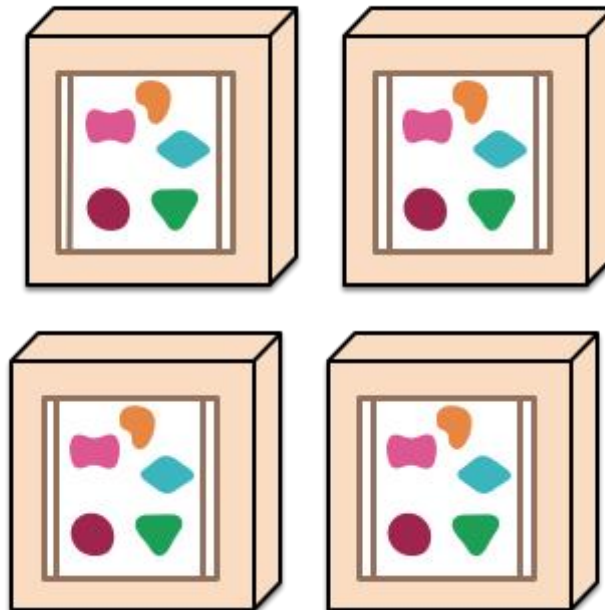


μ Services
==
SOA done properly

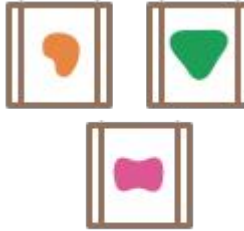
*A monolithic application puts all its
functionality into a single process...*



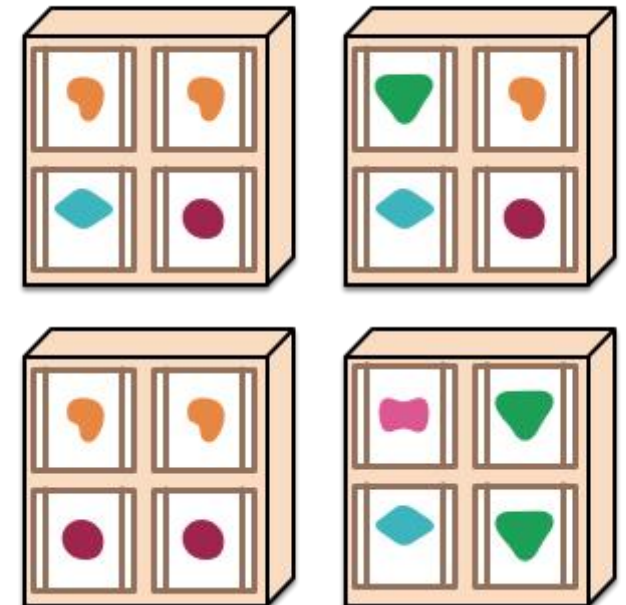
*... and scales by replicating the
monolith on multiple servers*



*A microservices architecture puts
each element of functionality into a
separate service...*



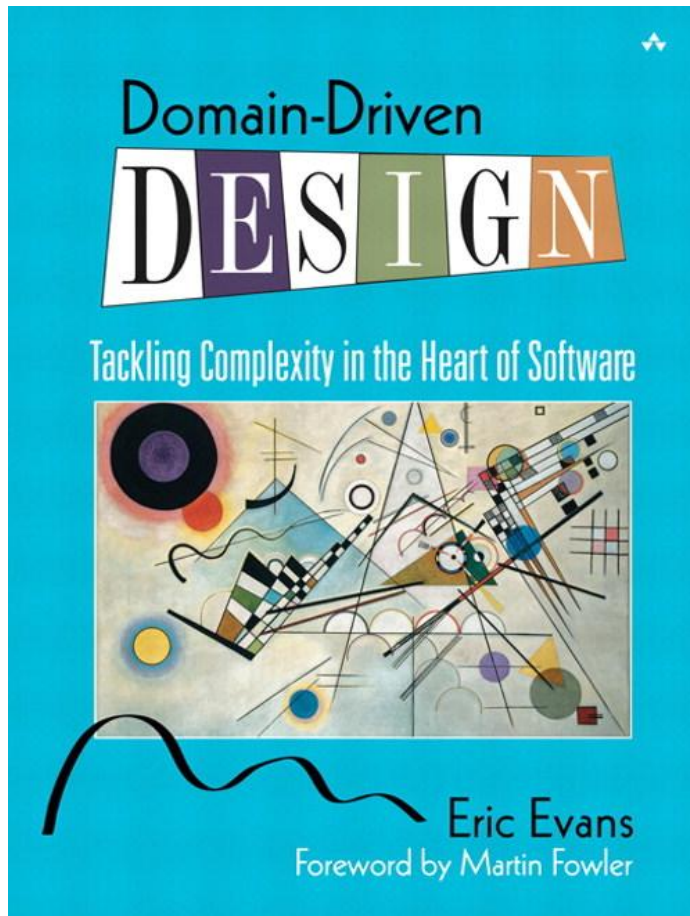
*... and scales by distributing these services
across servers, replicating as needed.*



Source : <http://martinfowler.com/articles/microservices.html>

**It's not a technical problem
It's a business problem**

Enters Domain-Driven Design



[Domain-Driven Design: Tackling Complexity in the Heart of Software](#) by [Eric Evans](#)

<http://dddcommunity.org>

Bounded context

Explicitly define the context within which a model applies.

Explicitly set boundaries in terms of

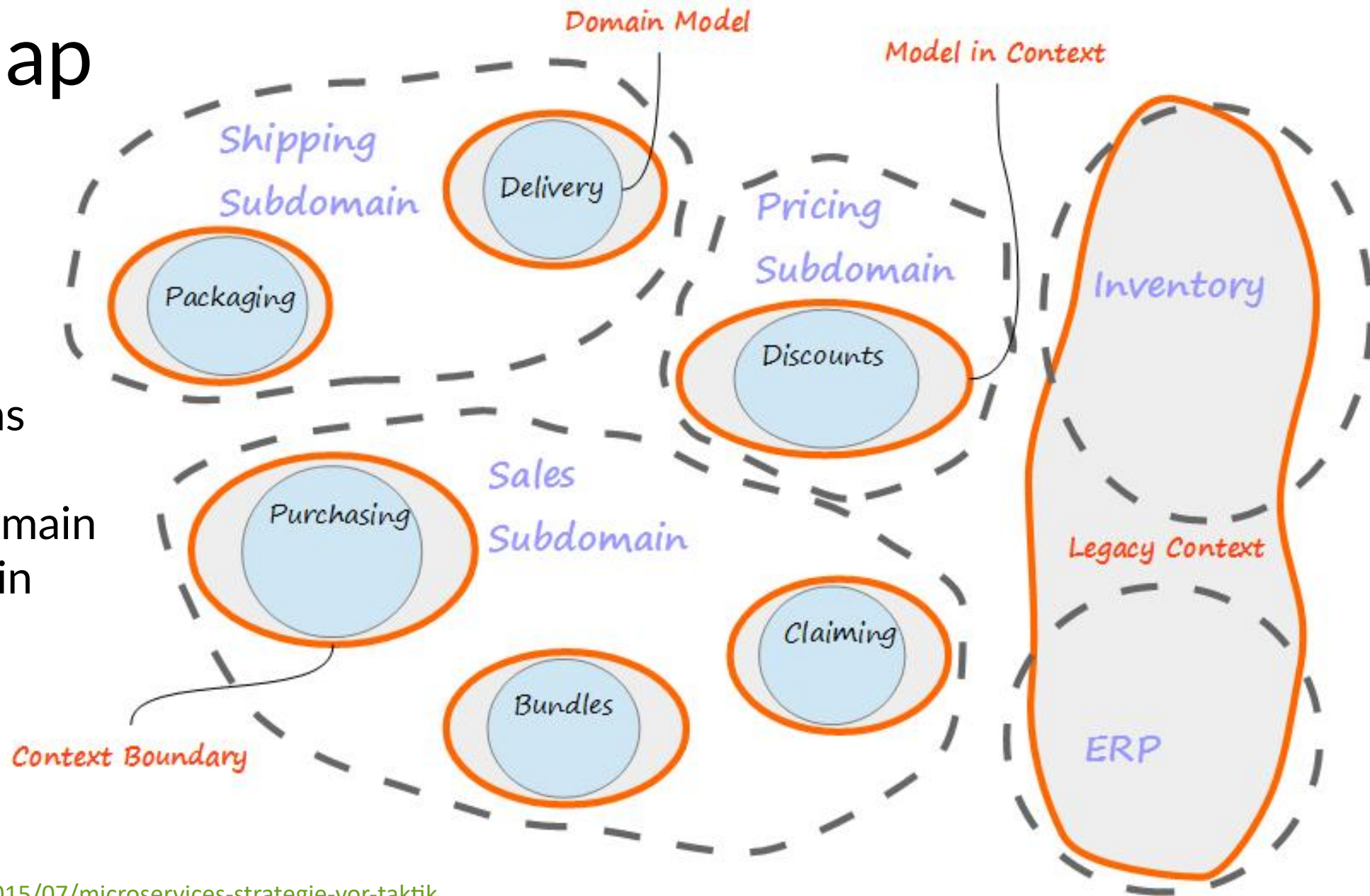
- team organization
- usage within specific parts of the application,
- and physical manifestations such as code bases and database schemas.

Standardize a single development process within the context, which need not be used elsewhere.

Model expressions, like any other phrase, only have meaning in context.

Source : [Domain-Driven Design: Tackling Complexity in the Heart of Software](#) by Eric Evans - Chapter 14 : Maintaing model integrity

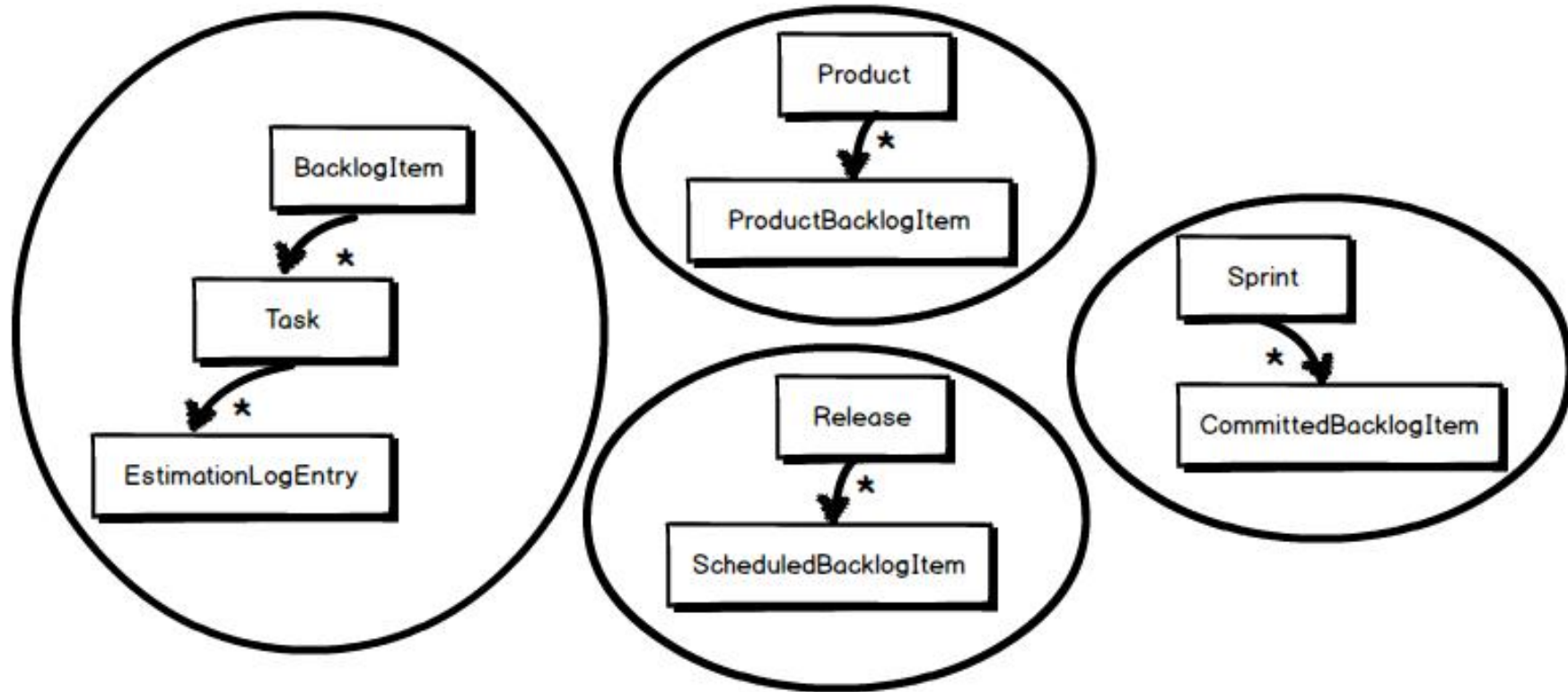
Context map



- Different sub-domains
- Core domain
 - Supporting sub-domain
 - Generic sub-domain

Source : <https://blog.codecentric.de/2015/07/microservices-strategie-vor-taktik>

Aggregates



- Transactional consistency boundaries
- Aggregate root is the gatekeeper of the aggregate
 - it enforces business rules
 - it cannot be blindsided

Source : [Modeling Aggregates with DDD and Entity Framework](#) by [Vaughn Vernon](#)

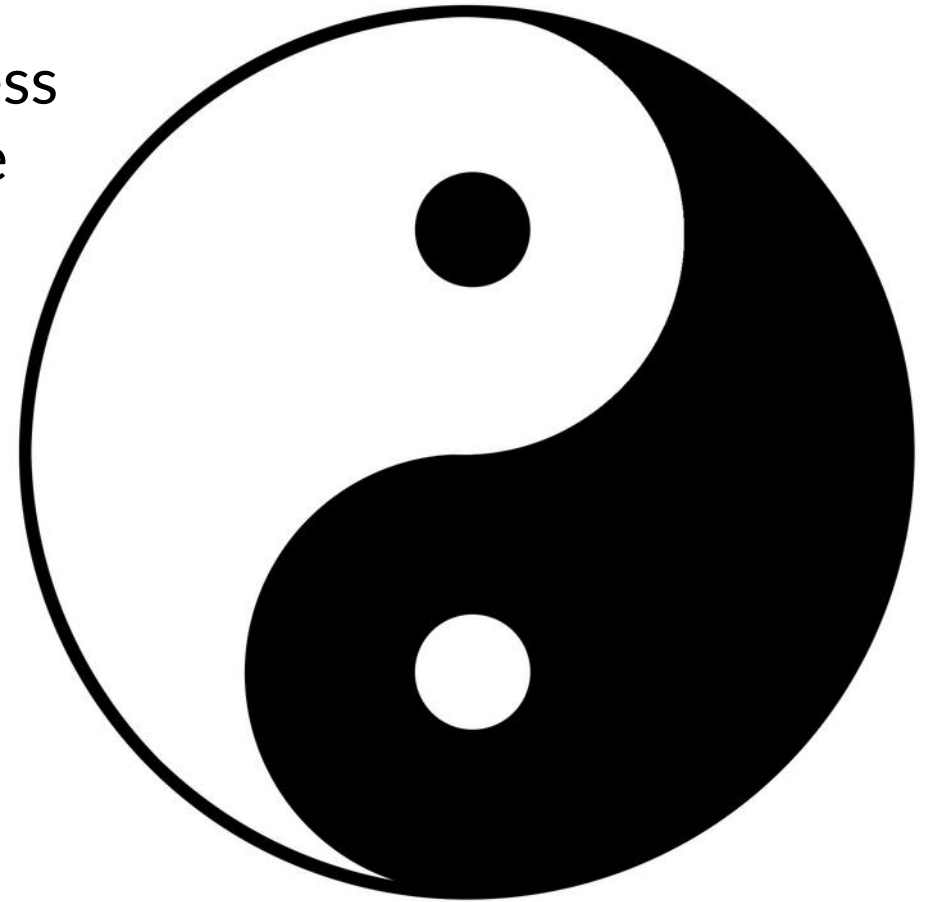
Ubiquitous language

Breaking the language barrier between business and tech people by sharing the same language

It's all about communication and learning

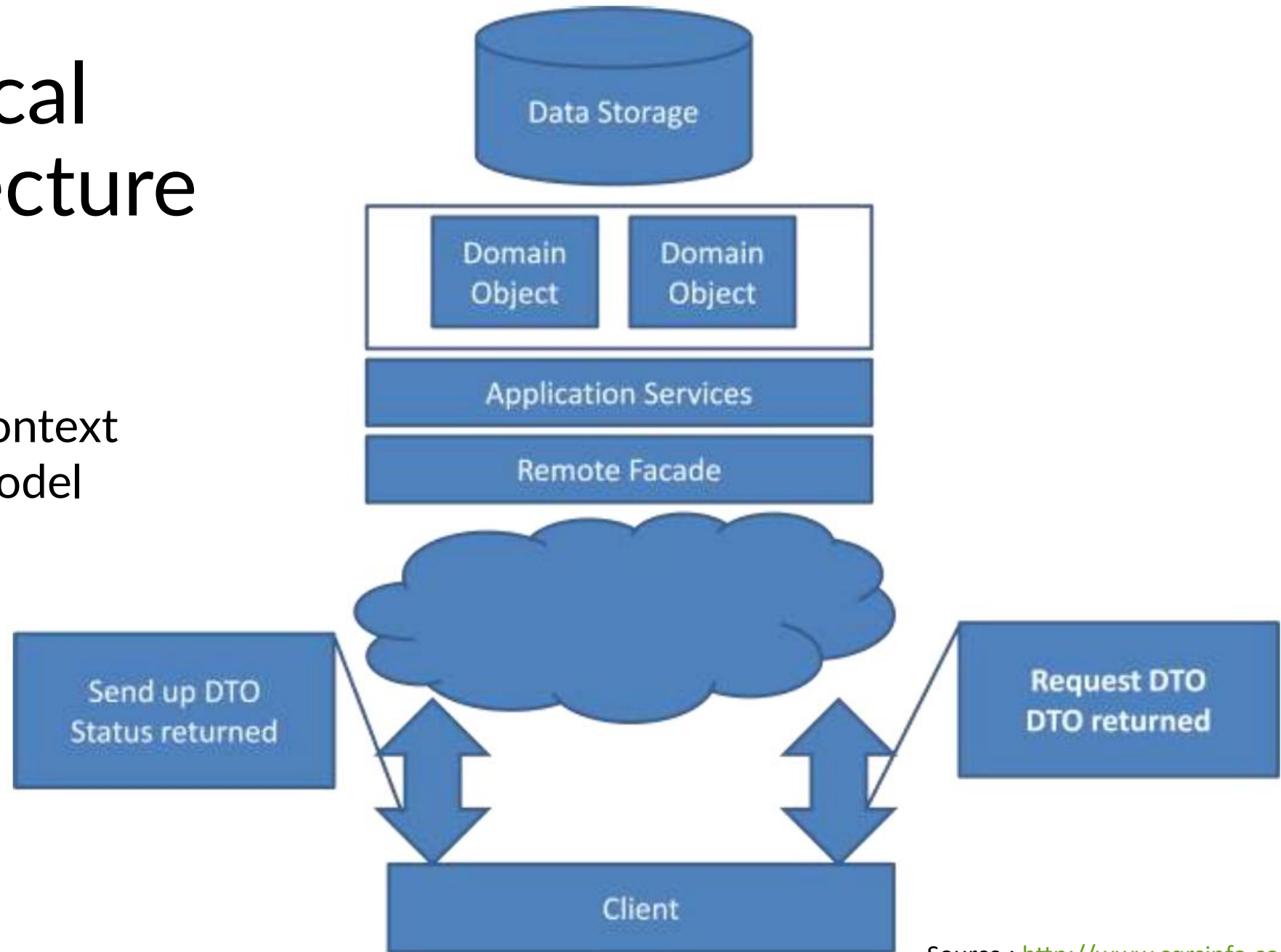
Breaking silos starts with communication

→ Language is ubiquitous in a context



A stereotypical DDD architecture

One per Bounded Context
Traditional scaling model



Source : <http://www.cqrsinfo.com>

You applied Strategic Design and now have several pristine Domain Models in context

Your system is divided into well defined Bounded Contexts and your contexts do not leak

Your domain logic and business rules are explicitly expressed in your code through well designed Aggregates

All is for the best

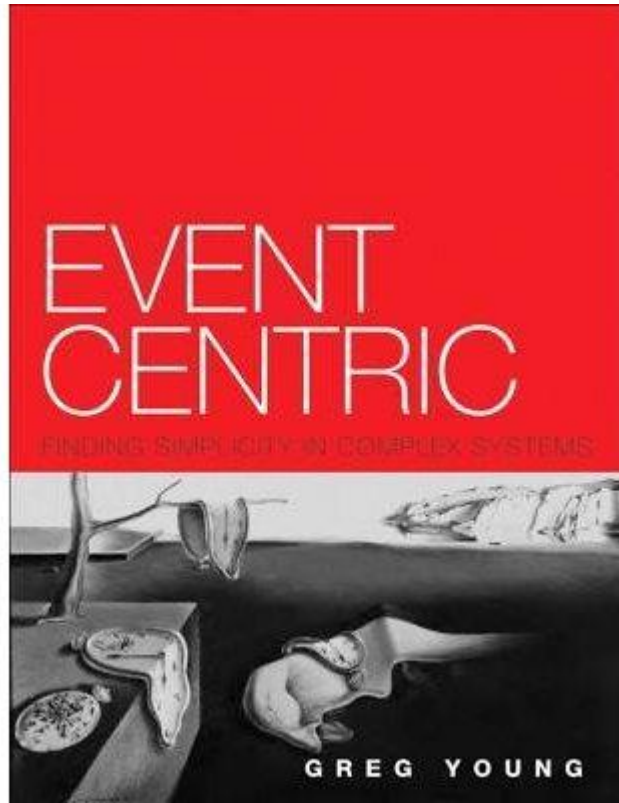
But

- How do you write reports ?
- Should you load 200 000 aggregates to compute statistics ?

CQRS

Yet another evolutionnary step

Command / Query Responsibility Segregation



Estimated release date

Sometime between Q3 2016
and 2035

[Event Centric: Finding Simplicity in Complex Systems](#) by [Greg Young](#)



Greg Young



<http://buildstuff.it>

Commands and Queries

Command

- It requests an action
- It mutates state
- It does not return data



Query

- It answers a question
- It does not mutate state
- It returns data



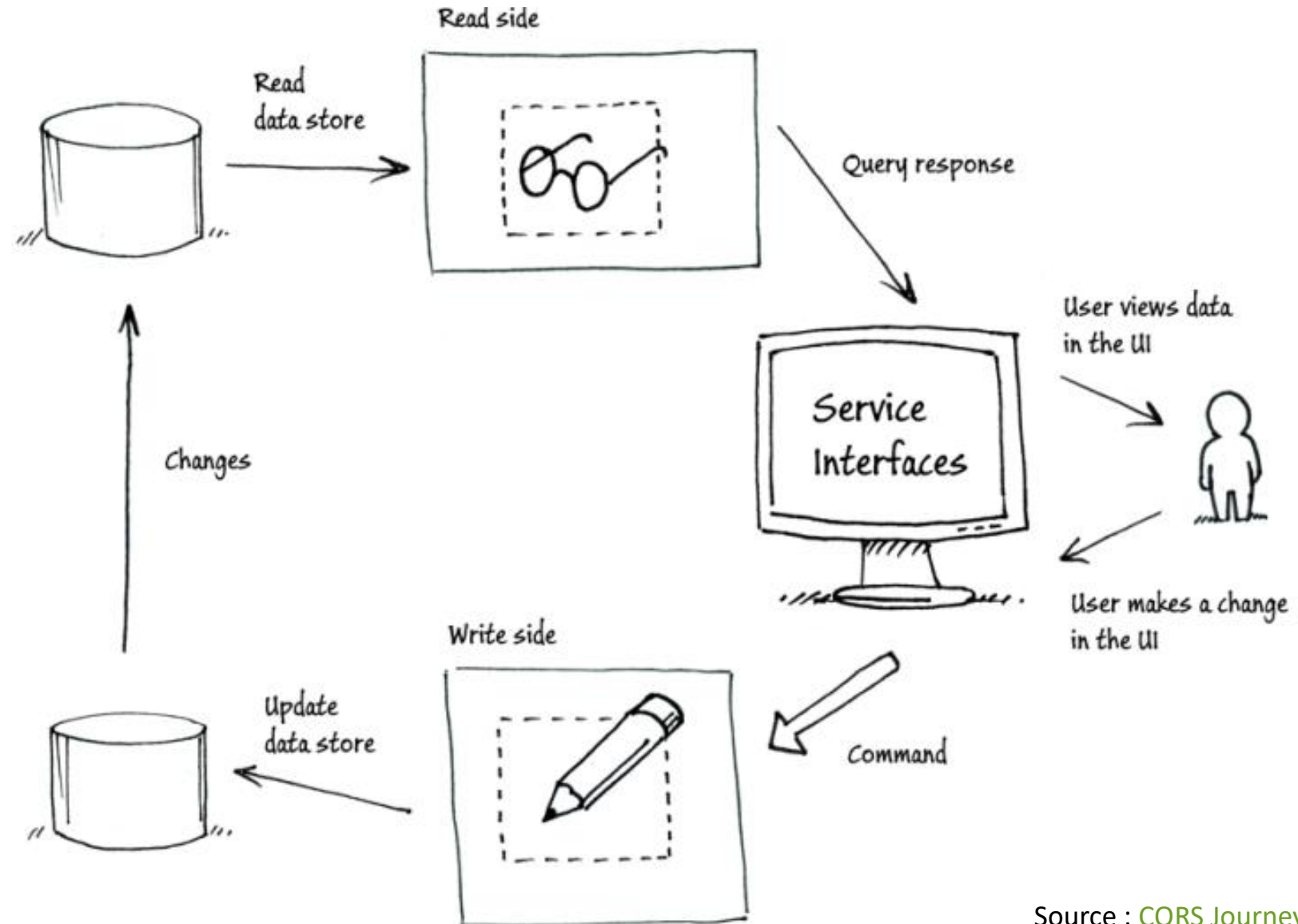
CQRS

Optimized for reads

- 80/20 law
- Denormalization brings direct display in UI
- Thin data access layer
- Caching
- BASE on the read side

Write side ensures consistency

- Use of DDD
- Domain logic and rules are enforced by aggregates
- ACID on the write side



Source : [CQRS Journey](#)

What about consistency ?

Brewer's CAP theorem

- Consistency : all nodes see the same data at the same time
- Availability : a guarantee that every request receives a response about whether it succeeded or failed
- Partition tolerance : the system continues to operate despite arbitrary partitioning due to network failures

No, you can't always get what you want

→ Sometimes, you need to make hard a choices

Eventual consistency

Don't underestimate the cost of maintaining consistency all the time

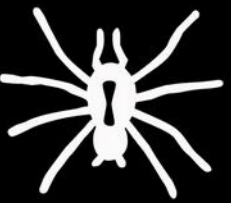
- What is the book of record ?

It's ok to use stale data

- Your data is stale as soon as it has been read from the database anyway

Don't fight it, embrace it

- It might even become a competitive advantage



KEEP
CALM
AND
SCARY
ON

**How do I keep the read
model in sync ?**

Distributed transaction

Characteristics

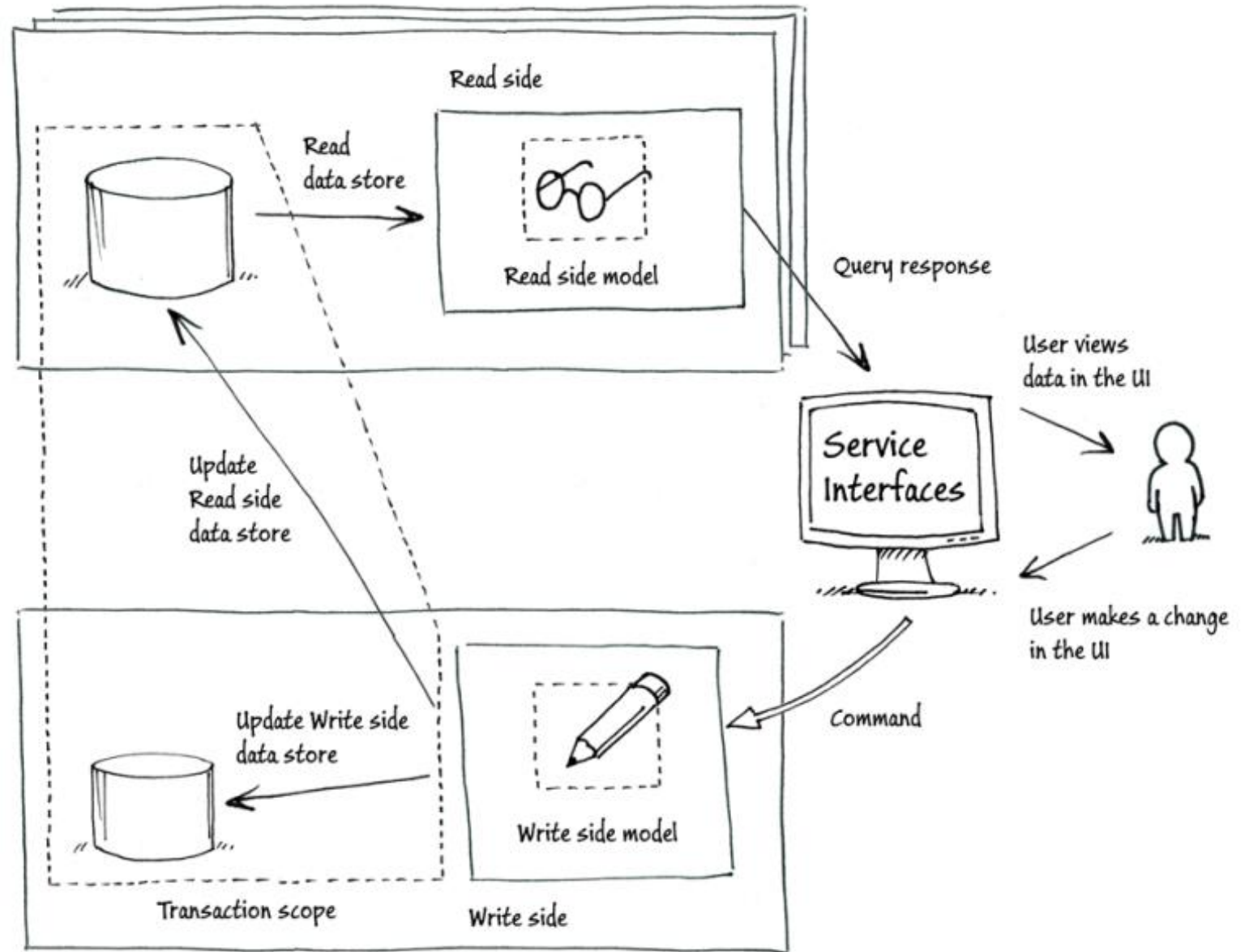
- Consistent (2 phase commit)

Problems ?

- Performance
 - Expensive to implement and maintain
 - Slow → 2 phase commit
- Availability
 - Does not scale
- Accidental complexity

Solution

→ Relax consistency ?



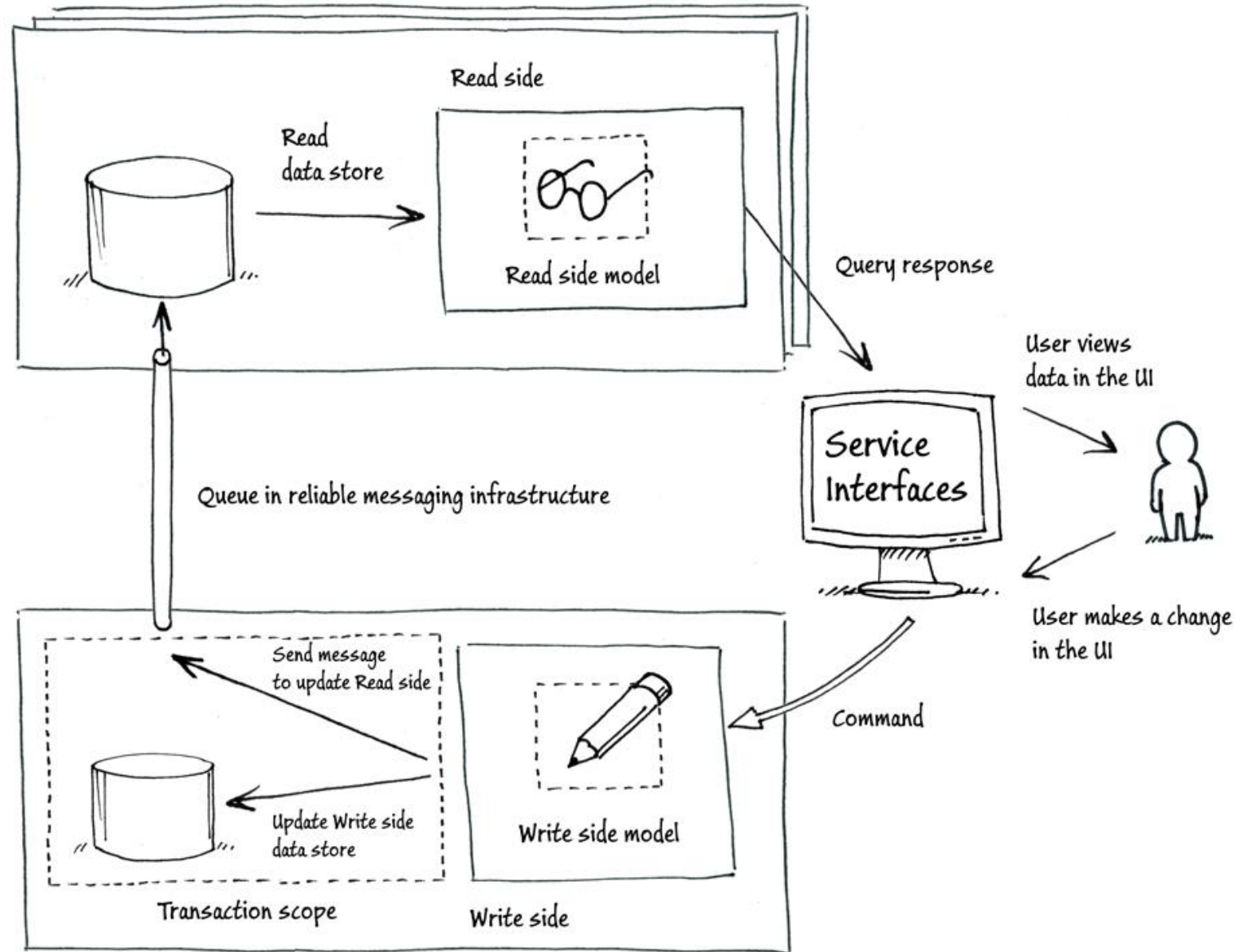
Using a reliable message transport

Characteristics

- Eventually consistent
- Performance
- Availability

Problems ?

- Your infrastructure needs to support transaction over database and queue



Source : [CQRS Journey](#)

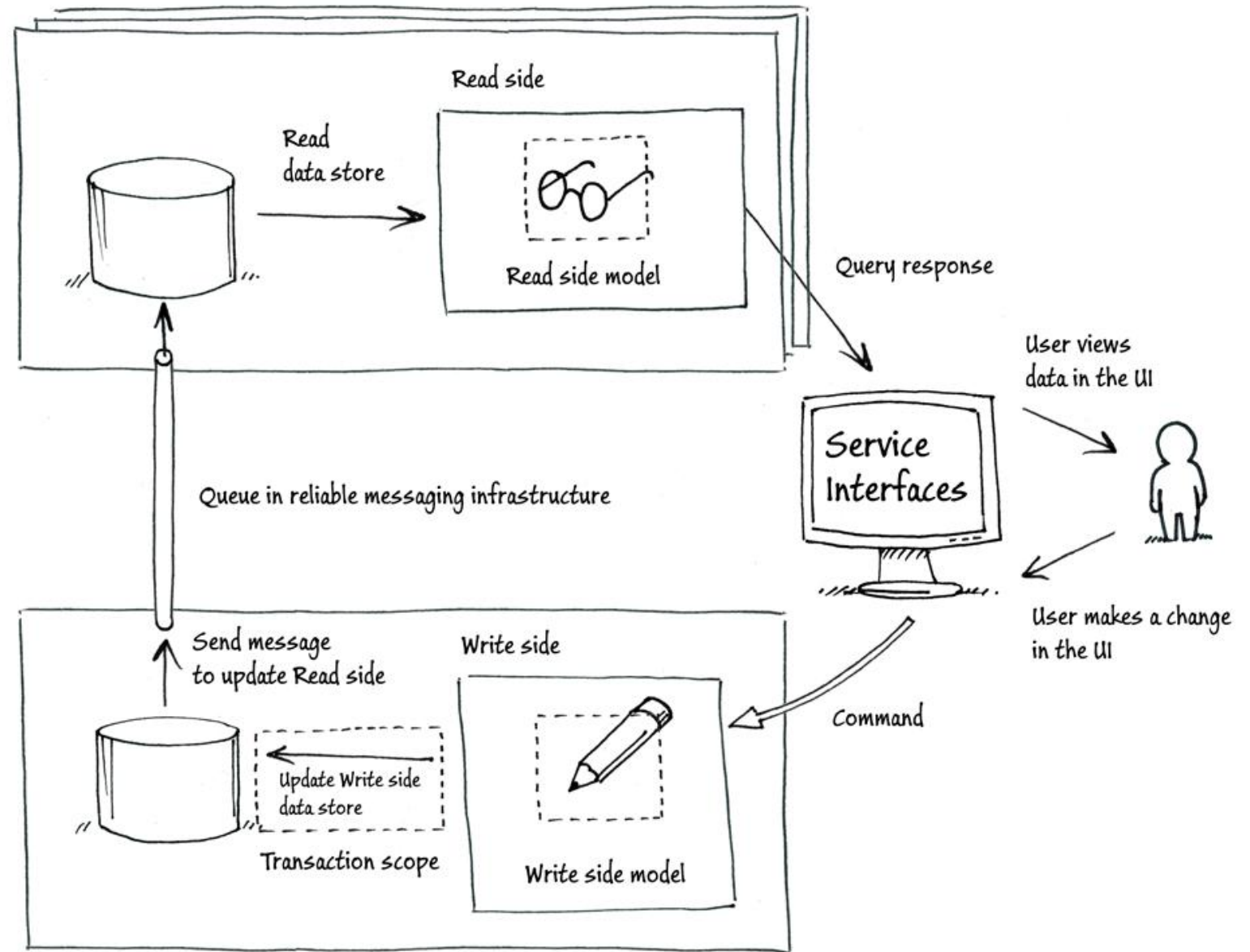
No transaction

Characteristics

- Eventually consistent
- Performance
- Availability

Problems ?

- Your data persistence has to cope with it
- Did I hear triggers ☹ ?



Source : [CQRS Journey](#)

You separated your reads from your writes

You bind DTO directly in your UI

Your UI is task oriented

Your write side ensures business rules and consistency

Your read side is eventually consistent

You did good

But

- You persist only current state and you still need an ORM
- Syncing read from write is expensive and might require some kind of transaction

Introducing Event Sourcing

Commands and Events

Command

- It's an action
- Requested (can say f*ck off)
- Imperative verb
- Self contained



Event

- It's a fact
- Always true (immutable)
- Passed tense verb
- Self contained



Event stream

Performance

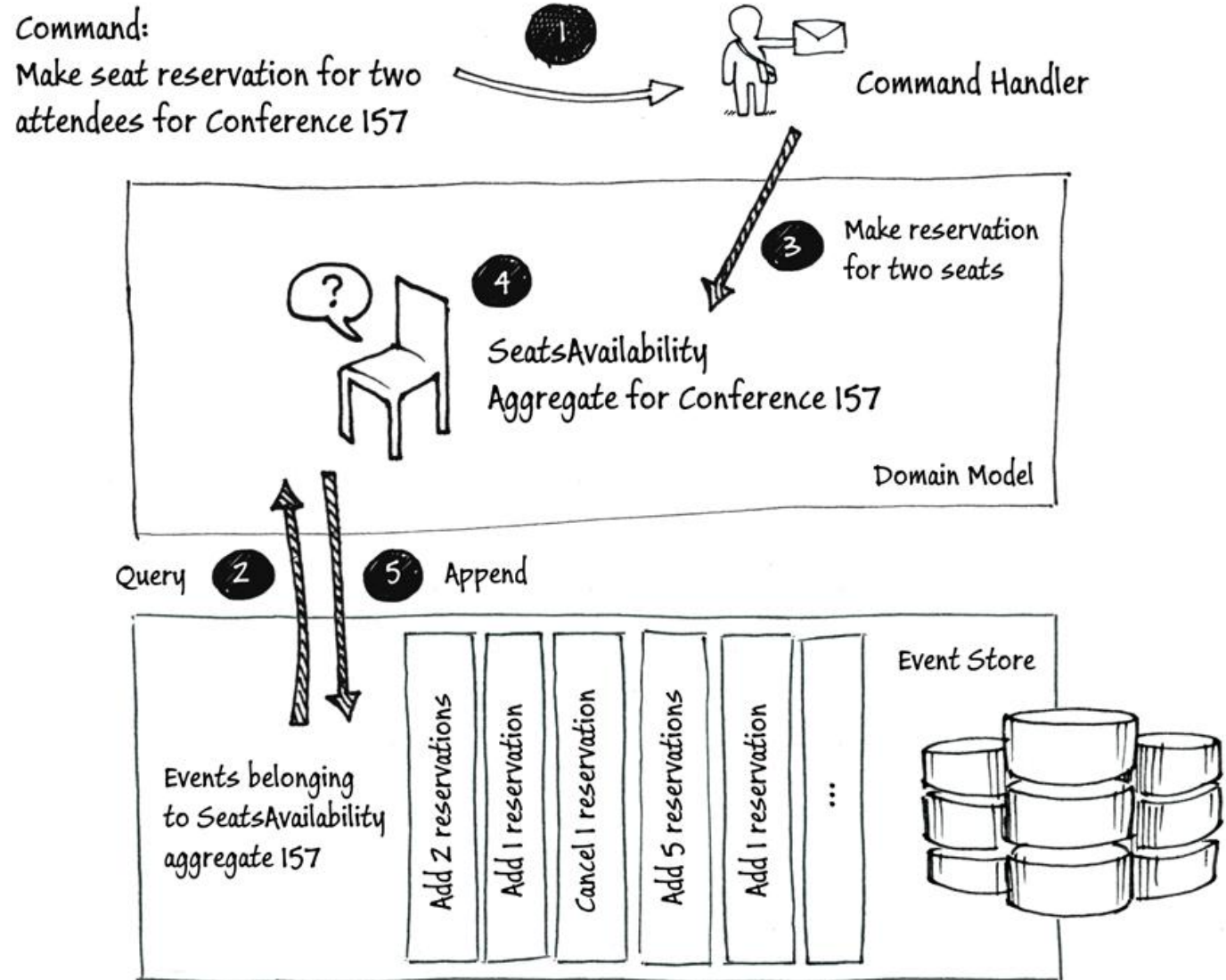
- Immutability
- Append-only

Simplicity

- Events are simple
- Minimal data access

History

- Audit log for free
- Record/playback
- Debugging



Event Sourcing

Save

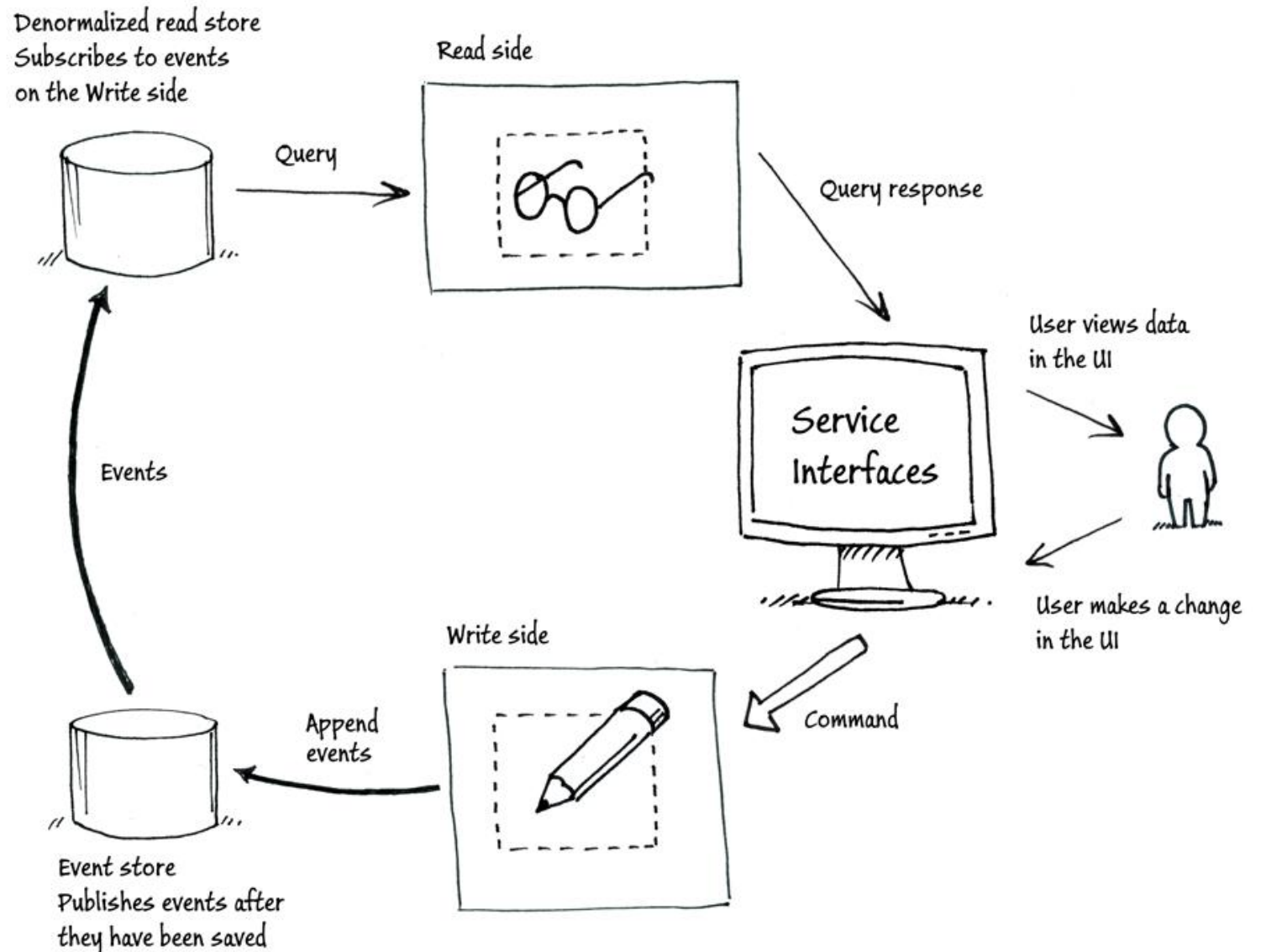
- Don't save current state
- Save a stream of small immutable events instead

Load

- Replay events to restore current state
- Use snapshots if need be

Sync

- Use events to sync the read model
- Use a publish/subscribe pattern to scale denormalization



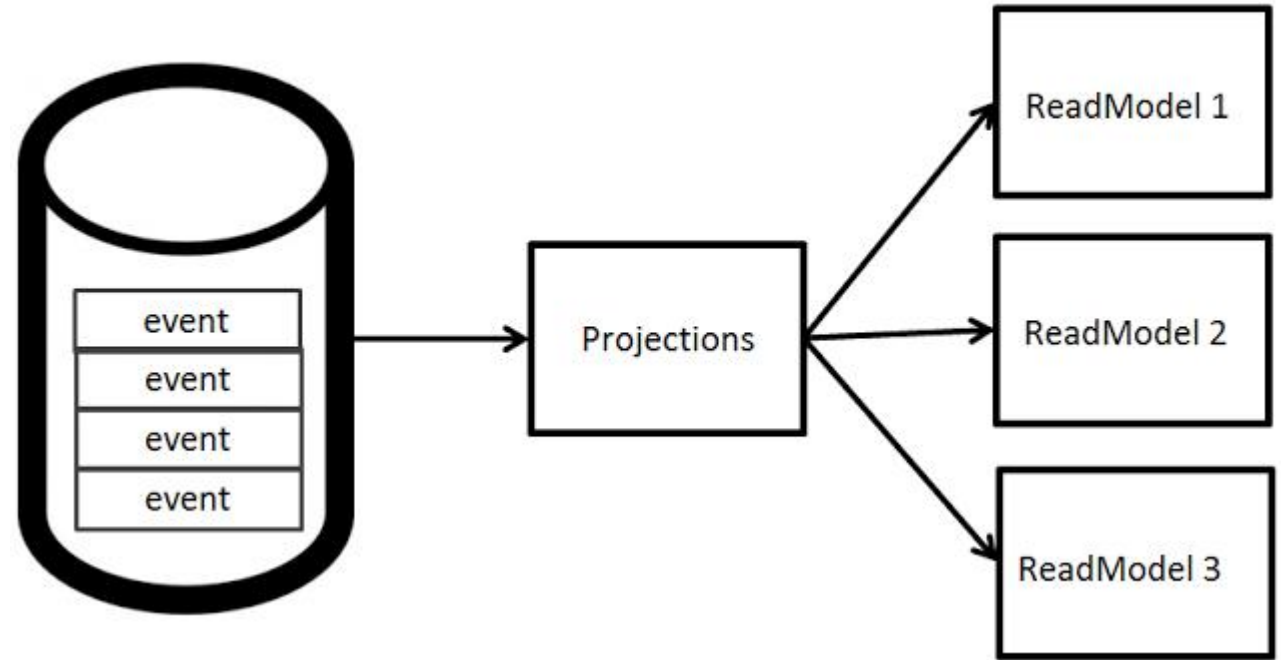
Source : [CQRS Journey](#)

Projections

React to events as they are written

Create new events when interesting combinations occur

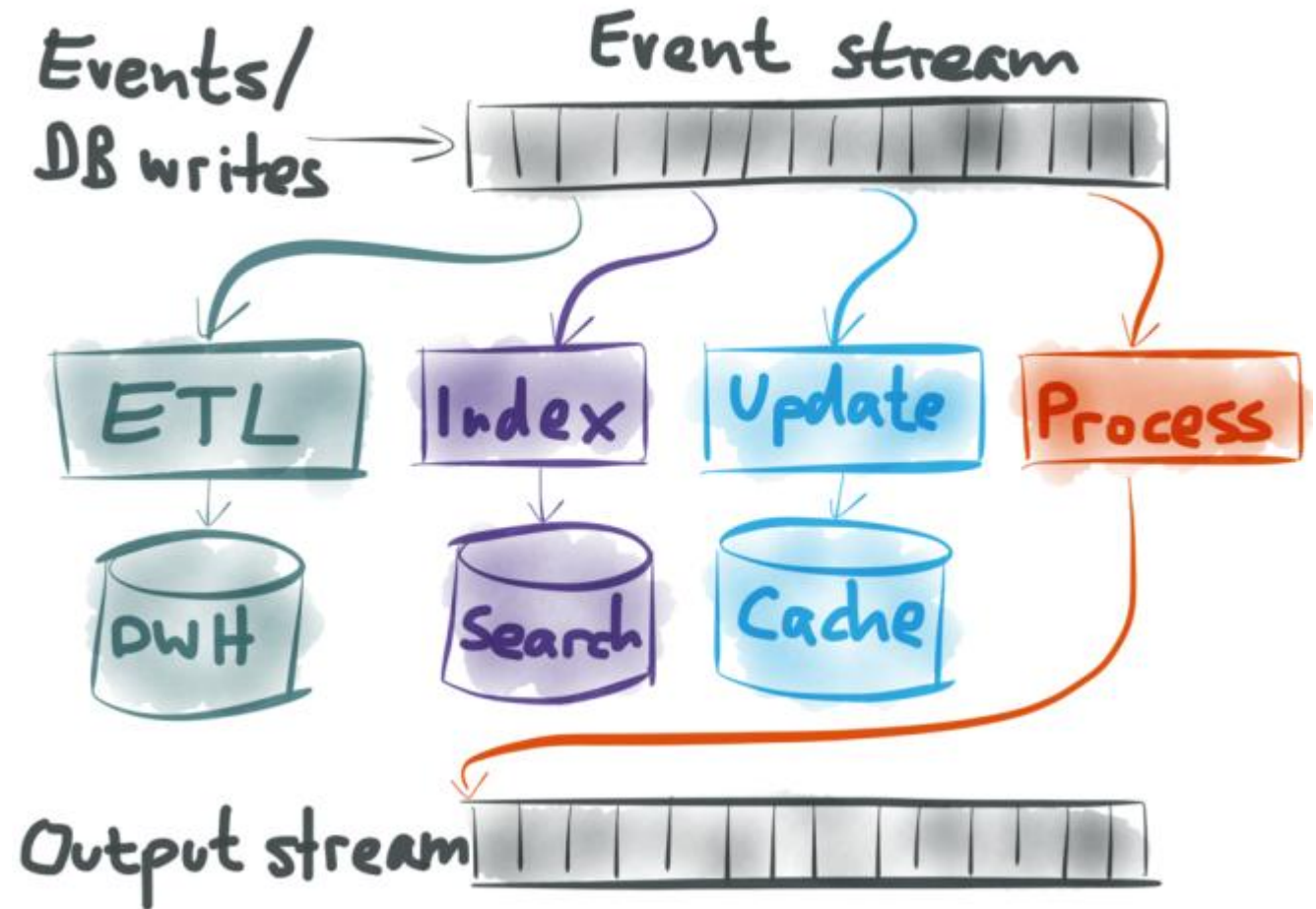
Write **temporal correlation queries** that run over historical data and on into the future



Source : <http://www.jefclaes.be/2013/10/event-projections.html>

Integration made easy

- Your even stream can be used as a message queue
- Use event projections to denormalize in multiple read models or external systems
- Leverage publish/subscribe infrastructure



Source : <http://www.confluent.io/blog/making-sense-of-stream-processing/>



Event sourced by design

- Written as an event sourced system
- Open source → go check the code out

Projections

- Consume or create new events
- Volatile, catch-up or persistent subscriptions

Portable

- Runs on mono on Linux, OS X and Windows
- Clients API for HTTP, .NET, Akka, Erlang

Great performances

- 15,000 writes per second
- 50,000 reads per second



<https://geteventstore.com/>

What else can I do with events ?

EventStorming



Alberto Brandolini

Introducing EventStorming : An act of Deliberate Collective Learning by Alberto Brandolini

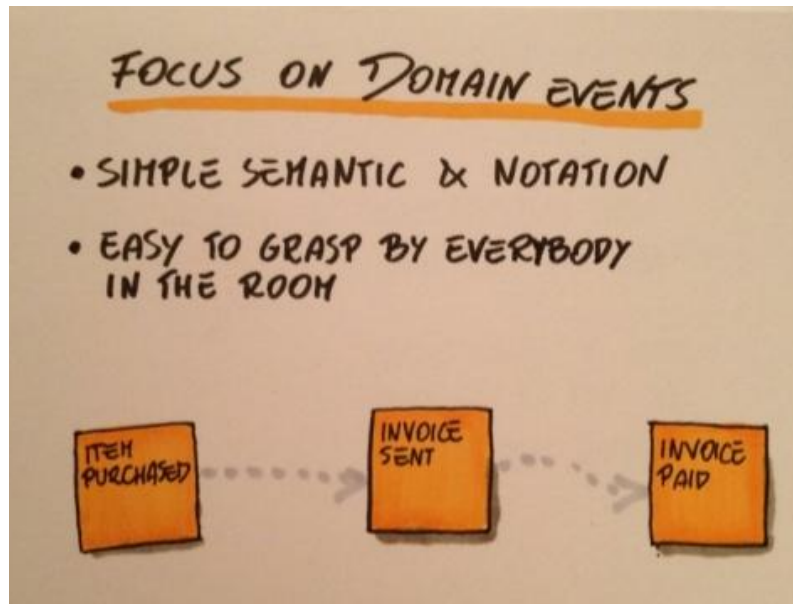
The bullshit asymmetry principle

*The amount of energy
necessary to refute
bullshit is an order of
magnitude bigger
than to produce it*

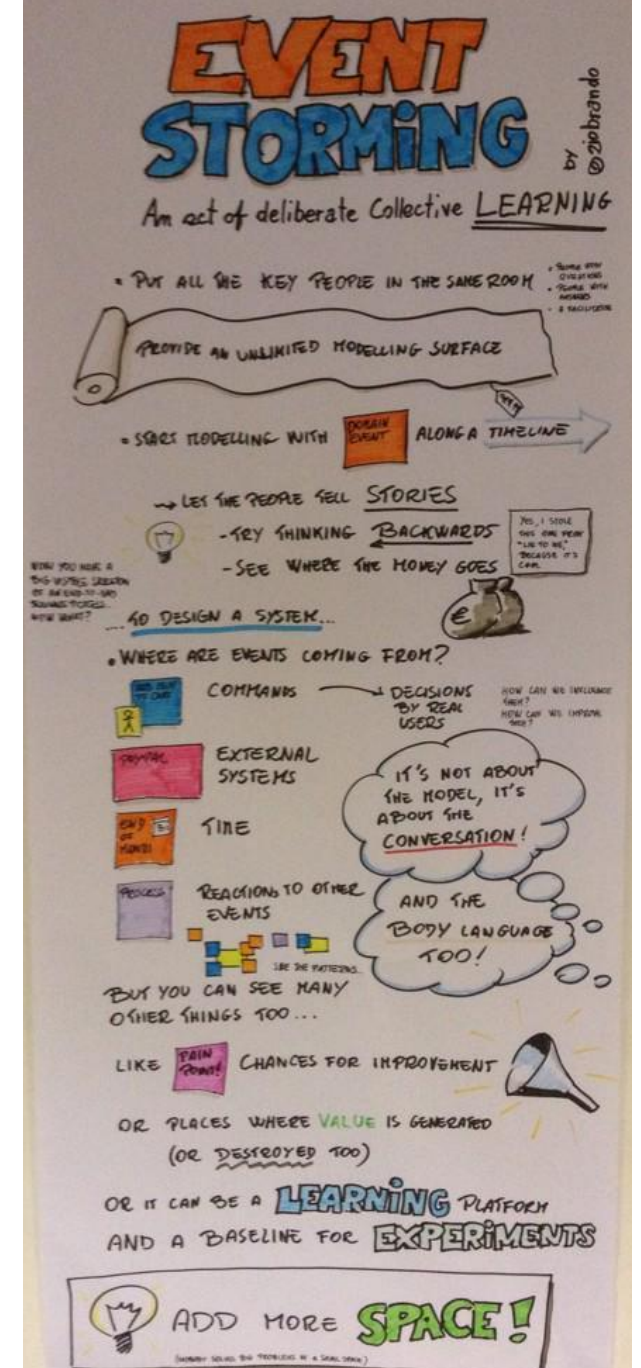


EventStorming

A workshop format for quickly exploring complex business domains



It is powerful
It is engaging
It is efficient
It is easy
It is fun



Source : <https://twitter.com/ziobrando/status/535828549102759936>

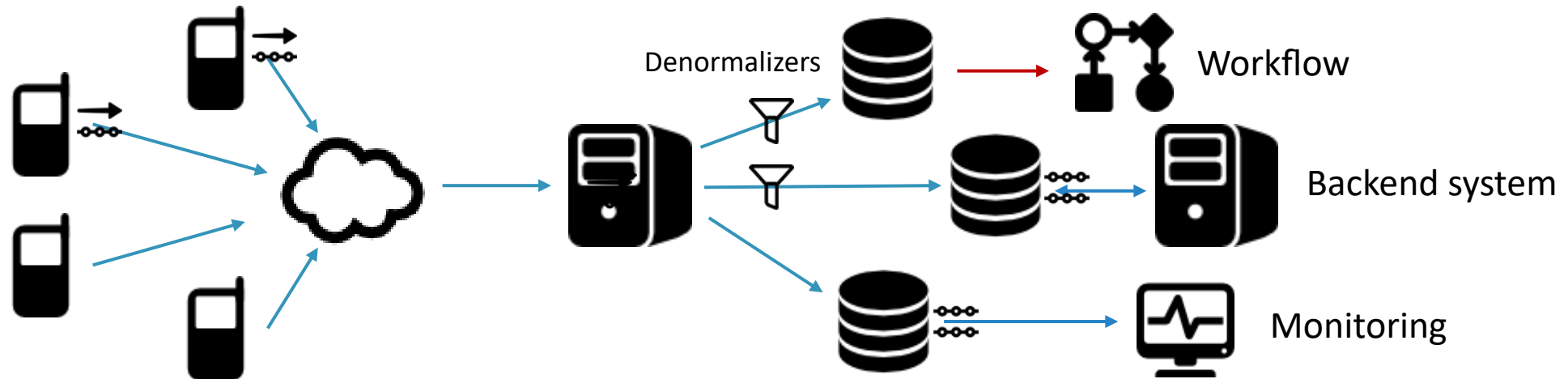
Source : <http://fr.slideshare.net/ziobrando/event-storming-recipes>

Things we've learnt from modeling with events

- Applies to many domains
- Changes the way domain experts think about the system
- Focus on behavior instead of structure
- Focus on temporal aspects : this happens before that

→ Develop new competitive advantage

Occasionally Connected Applications



Source : <http://www.slideshare.net/CdricPontet/events-at-the-tip-of-your-fingers-42179091>

EventSourced != EventDriven



CQRS/EventSourcing is NOT a top level architecture
→ Use it only when it makes sense

MicroService, Reactive, Actors

What is all this fuss about ?

Imperative vs Reactive

Responsive

- Respond in a timely manner
- Consistent quality of service

Resilient

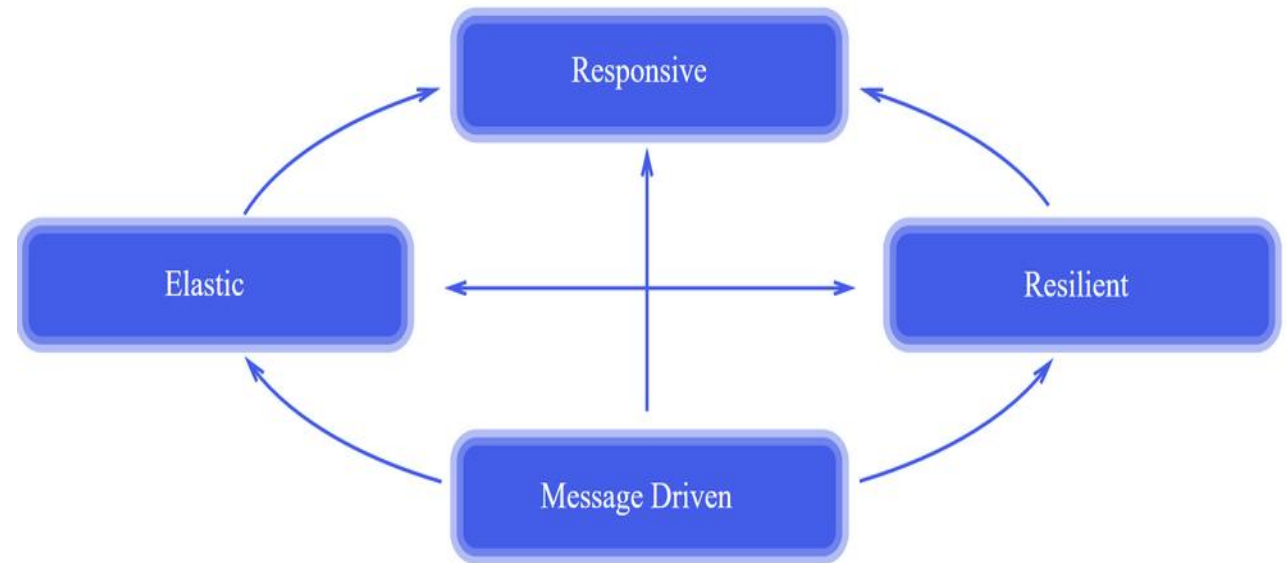
- Stay responsive in the face of failure
- Failures are contained
- Recovery is delegated to an other component

Elastic

- Stay responsive under varying workloads
- No bottlenecks

Message driven

- Asynchronous message passing
- Clear boundaries ensure loose coupling



Source : <http://www.reactivemanifesto.org>

Reactive Extensions



Languages

- Java: RxJava
- JavaScript: RxJS
- C#: Rx.NET
- C#(Unity): UniRx
- Scala: RxScala
- Clojure: RxClojure
- C++: RxCpp
- Ruby: Rx.rb
- Python: RxPY
- Groovy: RxGroovy
- JRuby: RxJRuby
- Kotlin: RxKotlin
- Swift: RxSwift
- PHP: RxPHP

Observables fill the gap by being the ideal way to access asynchronous sequences of multiple items

Pull
vs
Push

	single items	multiple items
synchronous	<code>T getData()</code>	<code>Iterable<T> getData()</code>
asynchronous	<code>Future<T> getData()</code>	<code>Observable<T> getData()</code>

An Observable is the asynchronous/push “dual” to the synchronous/pull Iterable

event	Iterable (pull)	Observable (push)
retrieve data	<code>T next()</code>	<code>onNext(T)</code>
discover error	throws <code>Exception</code>	<code>onError(Exception)</code>
complete	<code>!hasNext()</code>	<code>onCompleted()</code>

Example code showing how similar high-order functions can be applied to an Iterable and an Observable

Iterable

```
getDataFromLocalMemory()
    .skip(10)
    .take(5)
    .map({ s -> return s + " transformed"
    })
    .forEach({ println "next => " + it })
```

Observable

```
getDataFromNetwork()
    .skip(10)
    .take(5)
    .map({ s -> return s + " transformed"
    })
    .subscribe({ println "onNext => " + it
    })
```

Imperative
vs
Reactive

Observer pattern
done properly

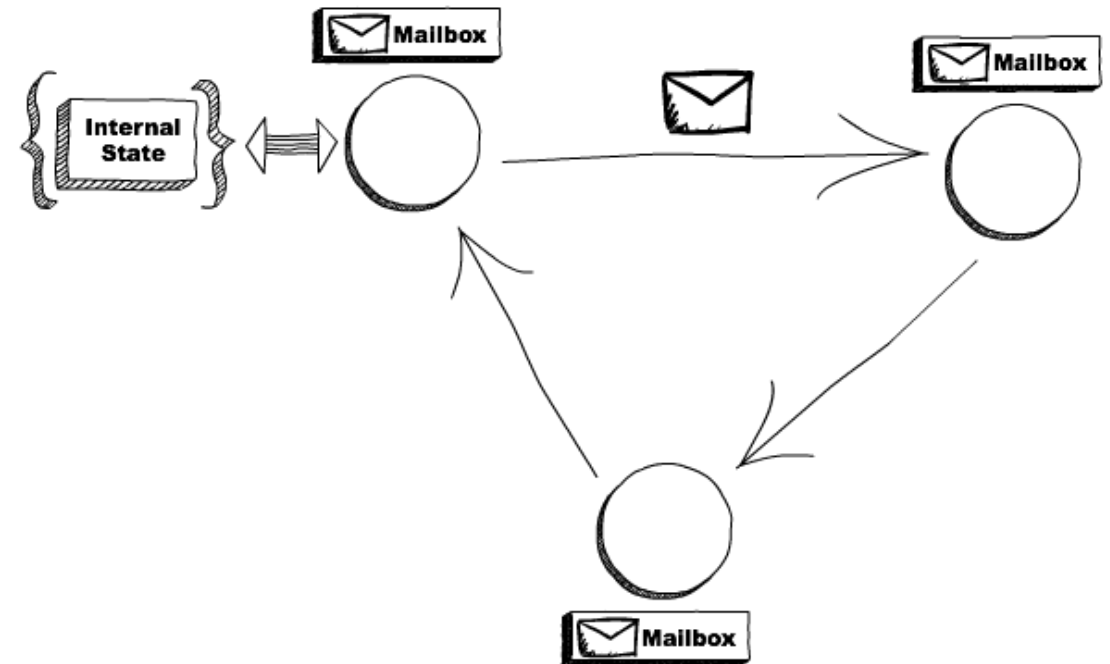
Source : <http://reactivex.io>

Actor model

An actor is a computational entity that, in response to a message it receives, can concurrently:

- send a finite number of messages to other actors;
- create a finite number of new actors;
- designate the behavior to be used for the next message it receives.

→ Computational model
→ Inherently concurrent
→ Infinitely composable
→ Fault tolerant through supervision



Source : <http://www.brianstorti.com/the-actor-model>

Better languages & tools are ready

Leverage your infrastructure

It's all functional

- Functional language
 - Immutable state
 - Functions with no side effect
 - Type system enforces invariants
- Event processing is a left fold

State $\leq$ State + Event

C#

```
class PersonalName : Value
{
    public PersonalName(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }

    public string FirstName { get; private set; }
    public string LastName { get; private set; }

    public override int GetHashCode()
    {
        return this.FirstName.GetHashCode() +
            this.LastName.GetHashCode();
    }

    public override bool Equals(object other)
    {
        return Equals(other as PersonalName);
    }

    public bool Equals(PersonalName other)
    {
        if ((object) other == null)
        {
            return false;
        }
        return FirstName == other.FirstName &&
            LastName == other.LastName;
    }
}
```

F#

```
[<StructuralEquality;NoComparison>]
type PersonalName = {
    FirstName : string;
    LastName : string }
```

```
class Person : IEntity
{
    public Person(int id, PersonalName name)
    {
        this.Id = id;
        this.Name = name;
    }

    public int Id { get; private set; }
    public PersonalName Name { get; set; }

    public override int GetHashCode()
    {
        return this.Id.GetHashCode();
    }

    public override bool Equals(object other)
    {
        return Equals(other as Person);
    }

    public bool Equals(Person other)
    {
        if ((object) other == null)
        {
            return false;
        }
        return Id == other.Id;
    }
}
```

```
[<NoEquality;NoComparison>]
type Person = {
    Id : int;
    Name : PersonalName }
```

Messaging & storage

Messaging infrastructure

- RabbitMQ
- ZeroMQ
- Apache Kafka
- Azure Service Bus



Choose the right tool according to your context
→ No dogma



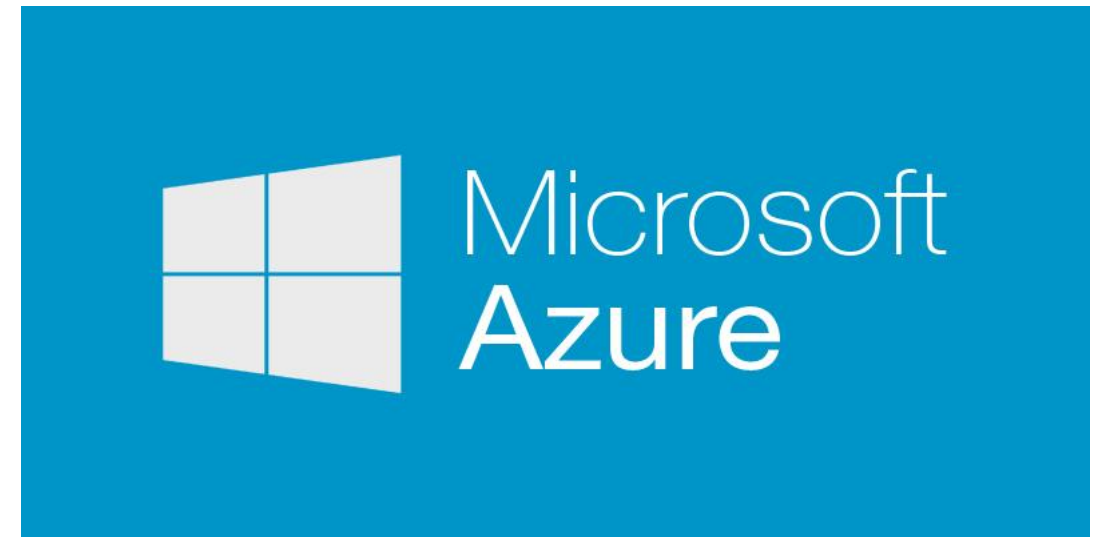
Make your pick

- Column
- Key-Value
- Document
- Graph
- Multi-model

The cloud is your best friend



Google Cloud Platform

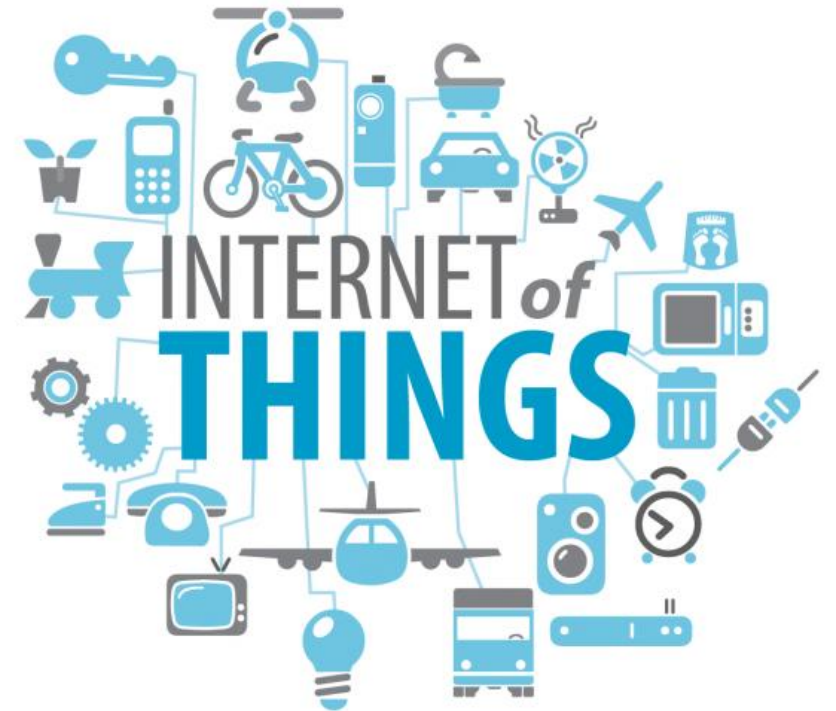


Embrace the new trends



IoT

- Telemetry == Events
- Stream analytics
- Smart Everything



Big data

- Machine learning
- Deep learning
- Data lake

To be continued...

What is the next big thing ?

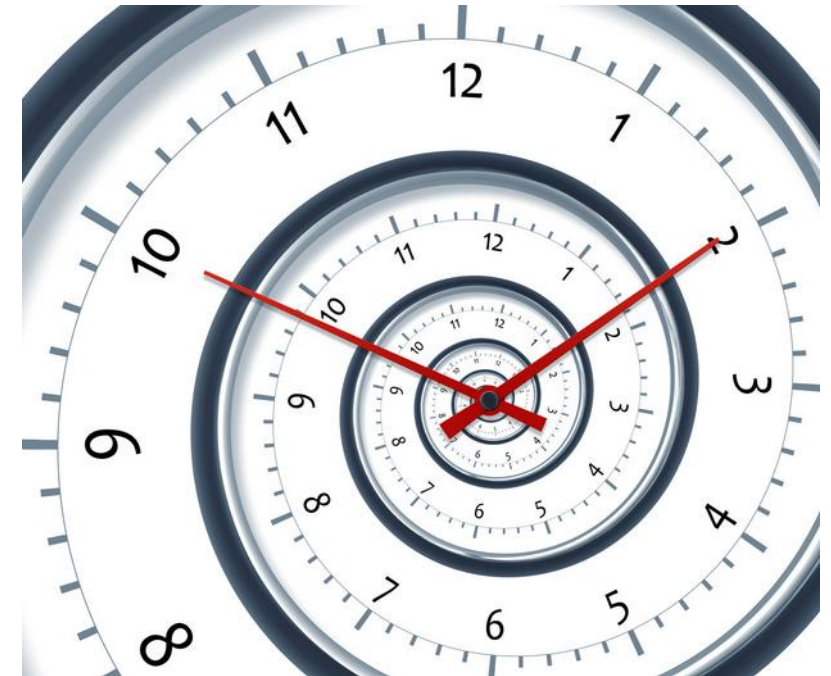
MicroServices is just the latest trend

→ Something else will come up !

Good modeling & design principles stay valuable through time

- SOLID
- Patterns of Enterprise Application Architecture
- Enterprise Integration Patterns
- Domain-Driven Design

→ **Loose coupling + high cohesion**



Distributed software is hard

First rule of distributed software

DON'T DISTRIBUTE unless you genuinely need to

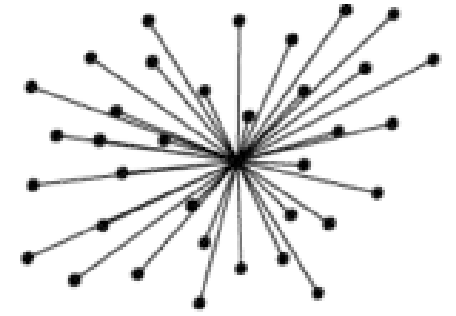
→ Design your software so that you might distribute

Beware of accidental complexity

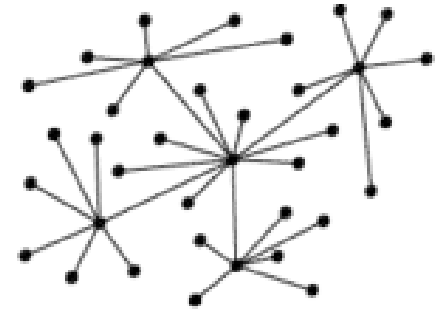
- Distribution makes things more difficult to understand
- The learning curve is steep

Remember : CQRS/ES is not a top level architecture

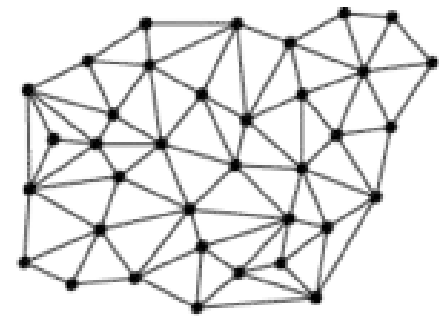
→ Use it only when it makes sense



centralised



decentralised



distributed



Interesting video content

- [DDDEurope - Eric Evans — Tackling Complexity in the Heart of Software](#)
- [DDDEurope - Greg Young — A Decade of DDD, CQRS, Event Sourcing](#)
- [DDDEurope - Alberto Brandolini — The Precision Blade](#)
- [DDDx - Eric Evans - DDD and Microservices: At Last, Some Boundaries!](#)
- [DDDx - Scott Wlaschin – Domain Driven Design with the F# type system](#)
- [React 2014 - Erik Meijer – What does it mean to be Reactive?](#)
- [Lang.NEXT - Hewitt, Meijer and Szyperski – The Actor Model](#)