Web-scale architecture

Prajeesh Prathap

Agenda

- ❑ Web-Scale architecture Introduction
- ❑ Eventual Consistency
- ❑ CQRS
- ❑ Event Sourcing
- ❑ Design for failure
- ❑ Domain Driven Design
- ❑ Onion Architecture
- ❑ Microservices
- ❑ Actor Model

I am going to talk about web-scale architecture and how it allows organizations to build systems in an agile manner that offer high levels of availability and flexibility and support Continuous delivery method

Praieesh Prathap
IT Architect

CONTACT
InfoSupport

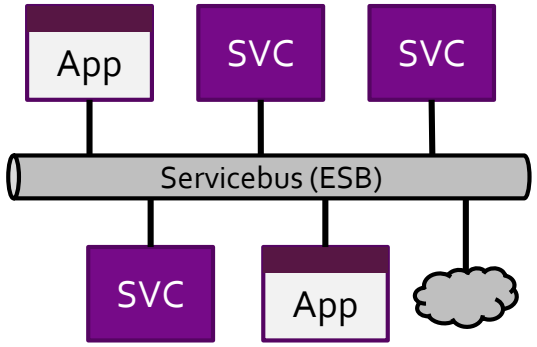

GB
AEF 190


Event sourcing
Bounded context
distributed computing
CAP
Eventual consistency
DDD
DevOps
Web-scale IT
CQS
Web-scale architecture
CQRS
Domain modelling
Domain driven design
Circuit breaker
Distributed systems
Introduction
Aggregate
NOSQL
Ubiquitous language
BASE
CAP
Actor model
Bulkhead
Continuous delivery
Consistency
Partition tolerance
Cloud computing
Event stroming
Hexagonal architecture
Event-based architecture
Serverless computing
Polyglot X
Design for failure
Onion architecture
Micro services

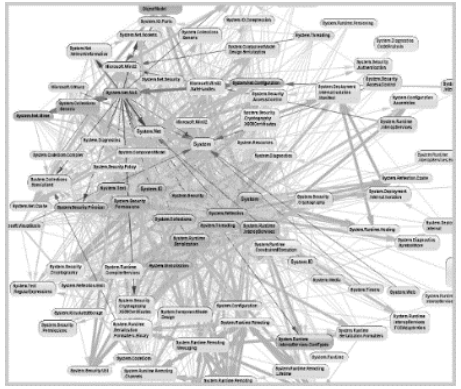
Traditional Architecture

Monolith

SOA



Big Ball of Mud

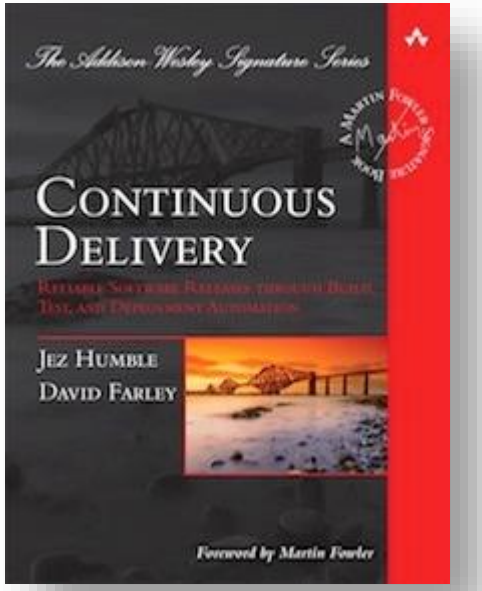


- Bad maintainability
  - Lots of tight coupling
  - Changes resonate throughout the entire application landscape
- Bad scalability
- Low availability
  - Often offline for upgrades or maintenance
  - Services / systems coupled @runtime
- Long release-cycles

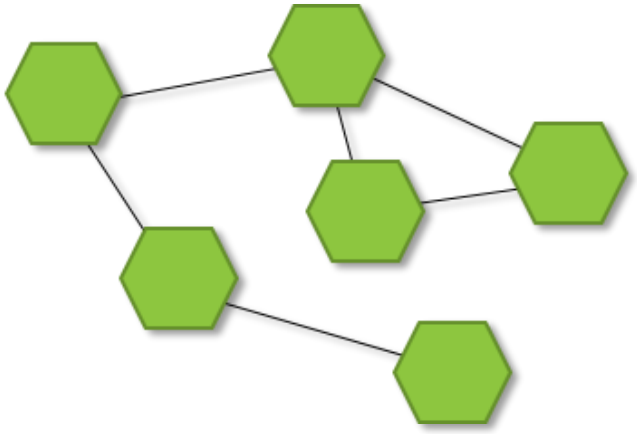
Traditional Architecture

How can we change this?

Agile Approach



Continuous Delivery
Automation
Dev Ops



Modern Web-scale Architecture

Introduction of the term web-scale

In a research note that was published yesterday, Gartner introduced the term "web-scale IT." What is web-scale IT? It's our effort to describe all of the things happening at large cloud services firms such as Google, Amazon, Rackspace, Netflix, Facebook, etc., that enables them to achieve extreme levels of service delivery as compared to many of their enterprise counterparts.

In addition, while the term "scale" usually refers to size, we're not suggesting that only large enterprises can benefit. Another scale "attribute" is speed and so we're stating that even smaller firms (or departments within larger IT organizations) can still find benefit to a web-scale IT approach. Agility has no size correlation so even more modestly-sized organizations can achieve some of the capabilities of an Amazon, etc., provided that they are willing to question conventional wisdom where needed.

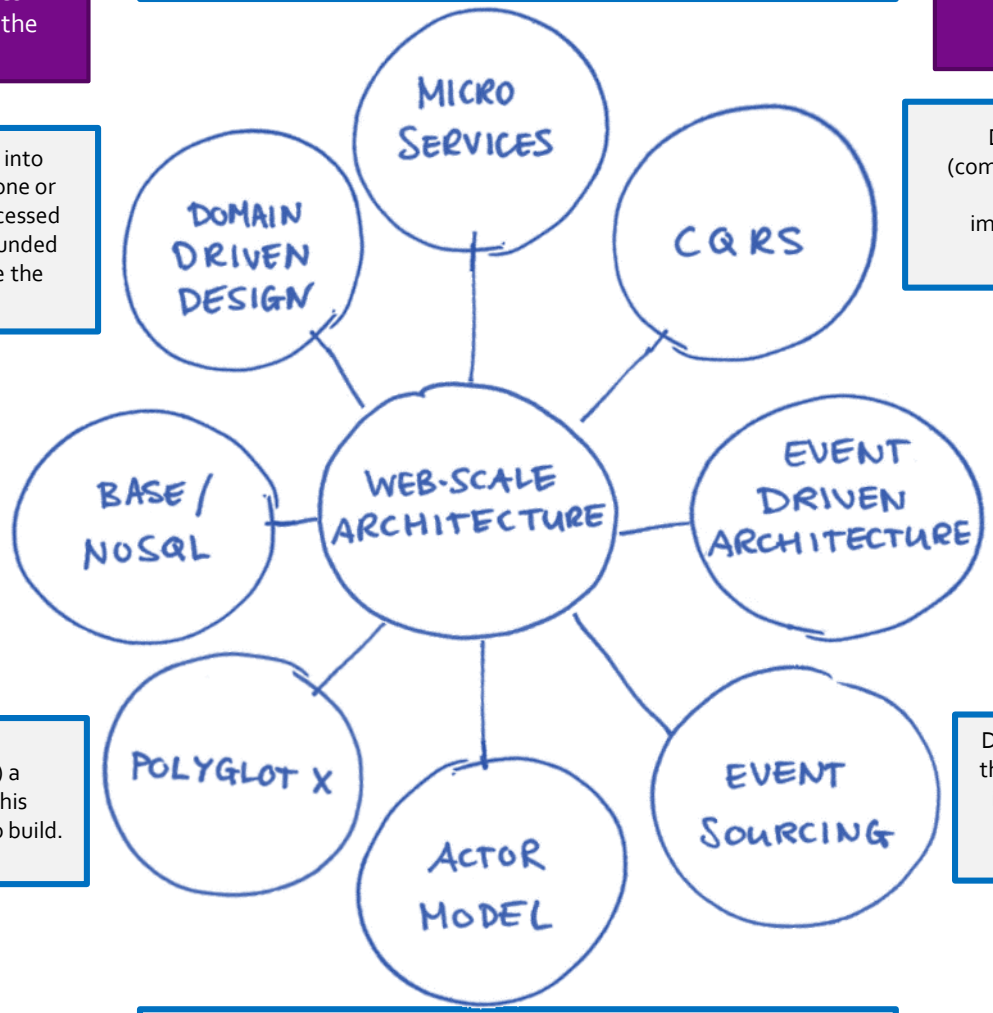
A web-scale architecture enables building systems that offer scalability, high performance and high availability. It also encourages loose-coupling which enables teams to employ continuous delivery for the development of the system.

Architecture pattern based on small, specialized and autonomous services that communicate using events. This pattern enables agile teams to develop services autonomously and release frequently.

See also: **The Reactive Manifesto**
http://www.reactivemanifesto.org/

We Are Reactive

An approach in which functional domains are split into autonomous areas (bounded contexts) containing one or more domain models (aggregates) which can be accessed through only 1 object (aggregate root). Within a bounded context an ubiquitous language is used to describe the entities and behavior within a domain.



Design pattern that separates the 'write side' (commands) and the 'read side' (queries) of a domain model. For both sides the most appropriate implementation is chosen based on the situation (context).

BASE vs. ACID.
"Not Only SQL". Alternative for data persistence that offers more speed, scalability or lower costs. Different types of NoSQL databases exist, each specialist in a specific type of data: documents, graphs, key-value pairs, wide-columns.

Architecture pattern based on asynchronous communication between components rather than synchronous communication. Duplication of data based on events is common in this kind of architecture. This improves scalability and loose coupling.

An approach in which per situation (context) a technical approach is chosen that best suits this situation and the characteristics of the system to build.

Design pattern that stores the state of a domain object as the series of events that occurred over time as opposed to in a normalized data model. Events are 'replayed' on an object after instantiation to recreate the state of the object.

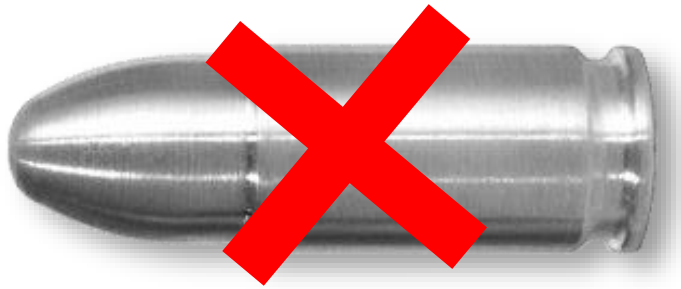
Design pattern that enables dividing a parallel workload over several small fully autonomous actors. An actor receives messages asynchronously, executes some logic and sends messages. An actor can create new actors to offload work.

Is that all brand new?

- **No!**
  - **Actor Model** : 1973 [Carl Hewitt]
  - **CQS** : 1988 [Book van by Bertrand Meyer]]
  - **DDD** : 2003 [Book by Eric Evans]
  - **CQRS** : 2009 [Blog post by Greg Young]
- We see more and more organizations adopting the web-scale architecture patterns

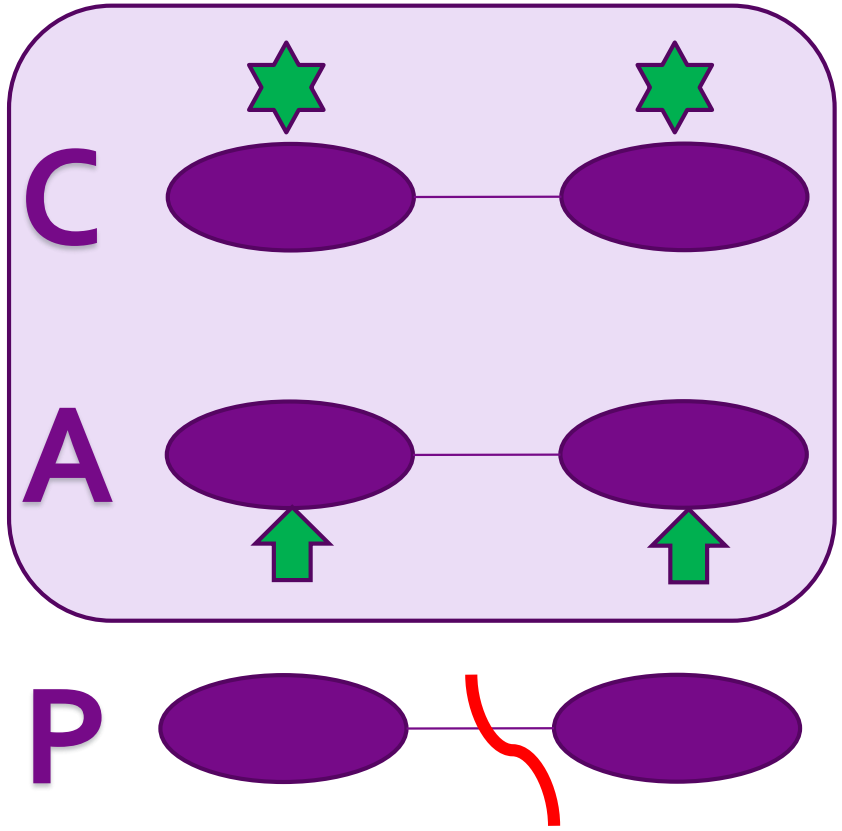
Disclaimer!

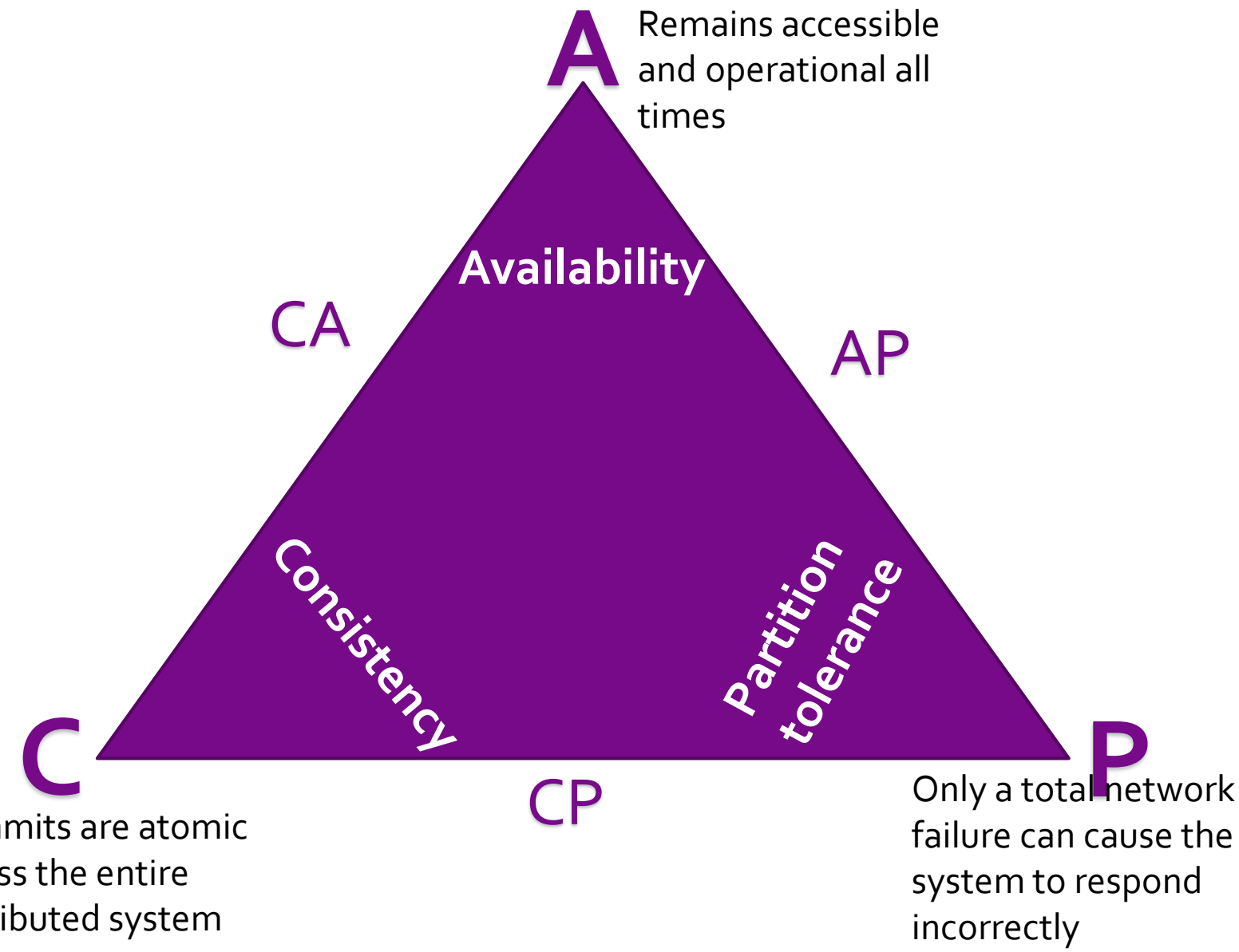
- KISS, common sense and software craftsmanship are still the most important tools of an engineer!
- Choose the best fit-for-purpose solution and architecture style based on complexity and risks!
- Every decision is a trade-off!




Onion architecture
Event-based architecture
Domain driven design
Domain modelling
DevOps
Micro services
Continuous delivery
Polyglot
X
Distributed systems
Agile
CAP
Actor model
CQRS
CQS
Event stroming
Hexagonal architecture
Bounded context
Circuit breaker
Eventual consistency
BASE
Partition tolerance
DDD
Web-scale
IT
Bulkhead
CAP
Consistency
Serverless computing
Cloud computing
distributed computing
Aggregate
NOSQL
Event sourcing
Design for failure
Ubiquitous language
Web-scale architecture

Eventual consistency – CAP theorem





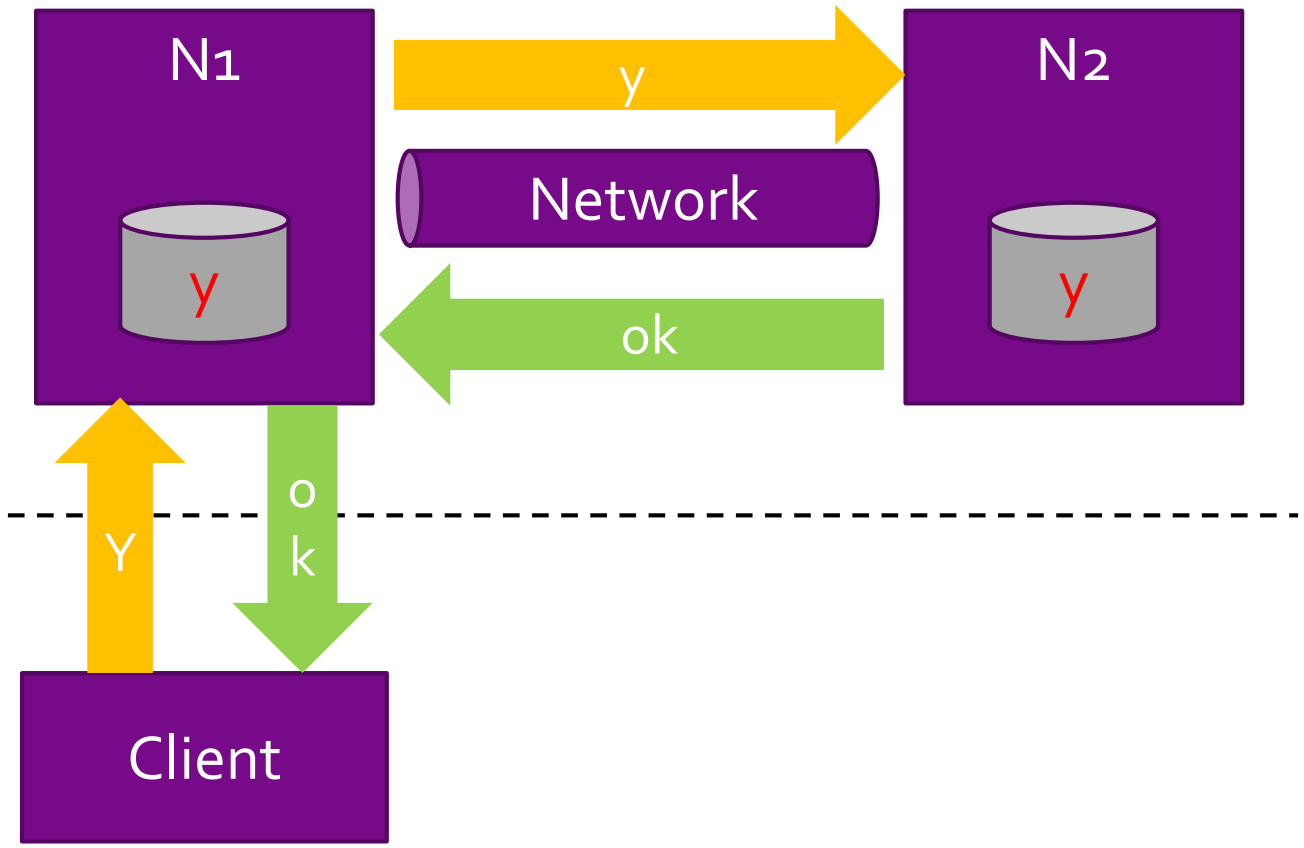
Remains accessible and operational all times

Commits are atomic across the entire distributed system

Only a total network failure can cause the system to respond incorrectly

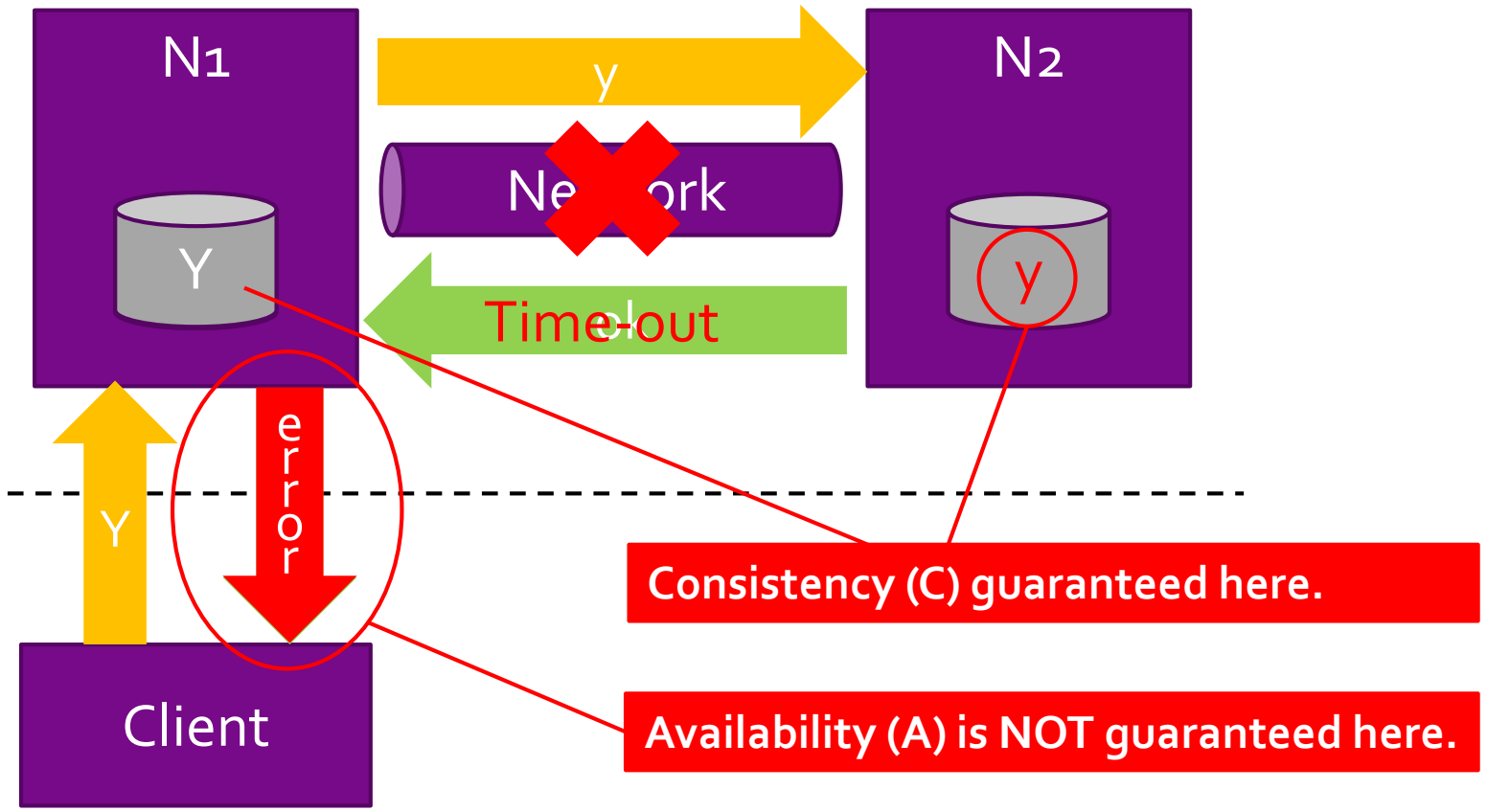
Eventual consistency - CP

Ensures data is always consistent throughout the system



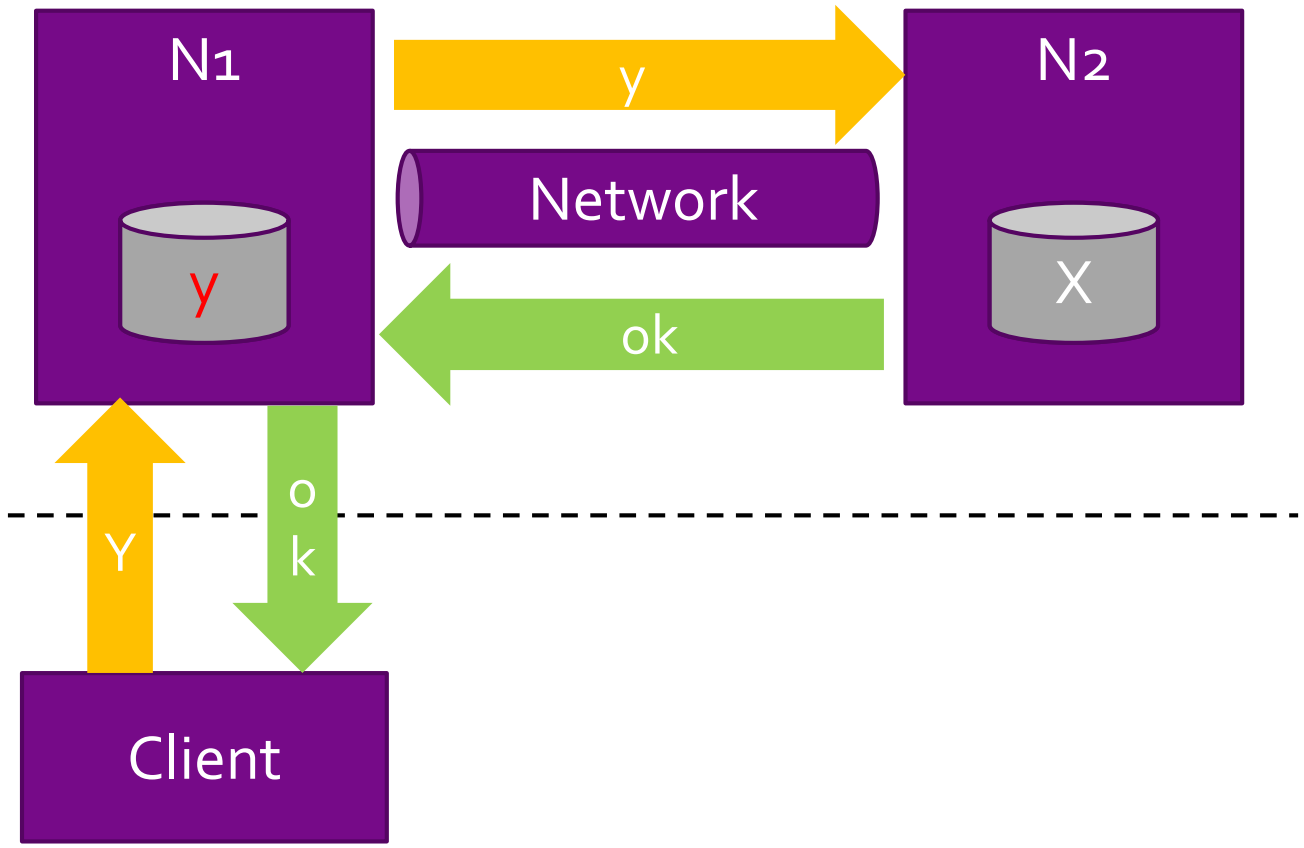
Eventual consistency - CP

Ensures data is always consistent throughout the system



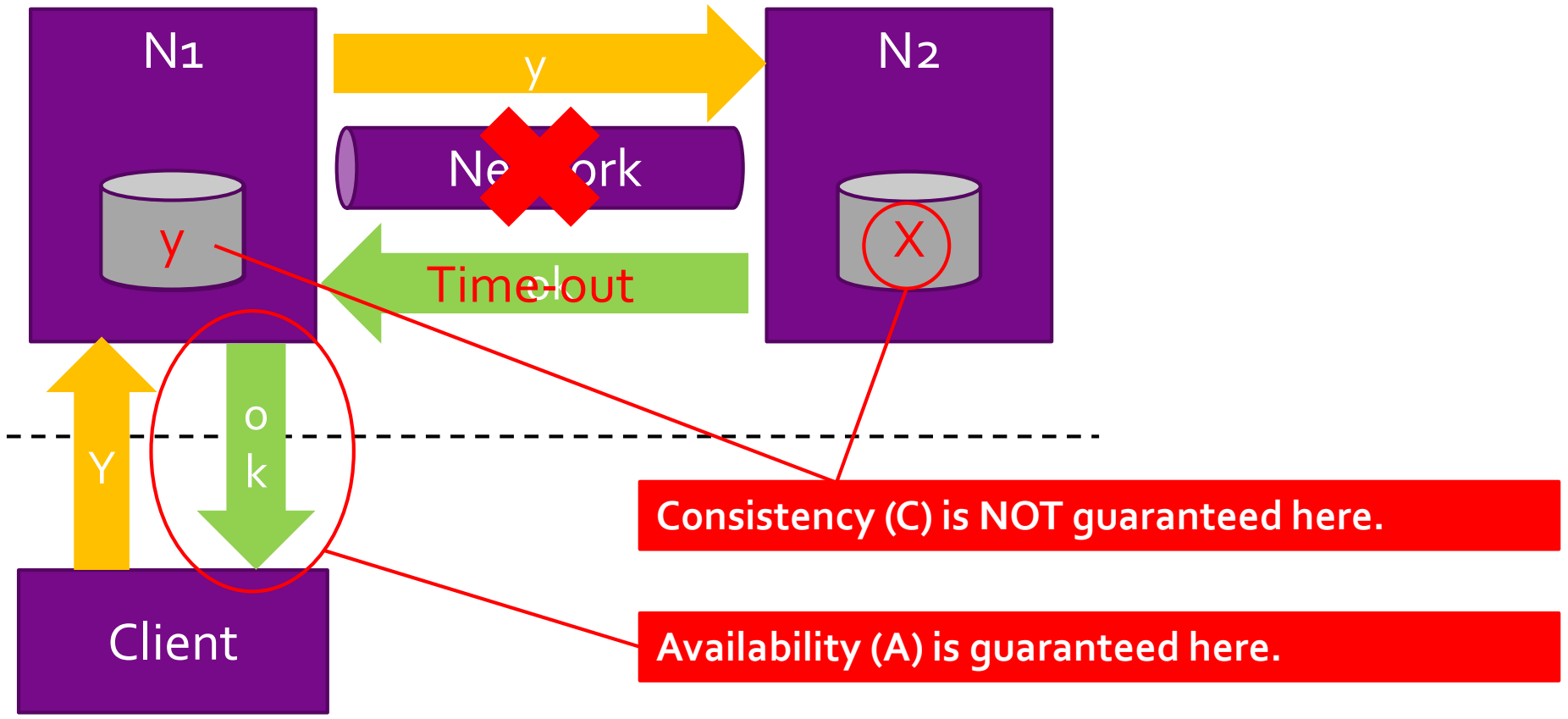
Eventual consistency - AP

Ensures services are always available



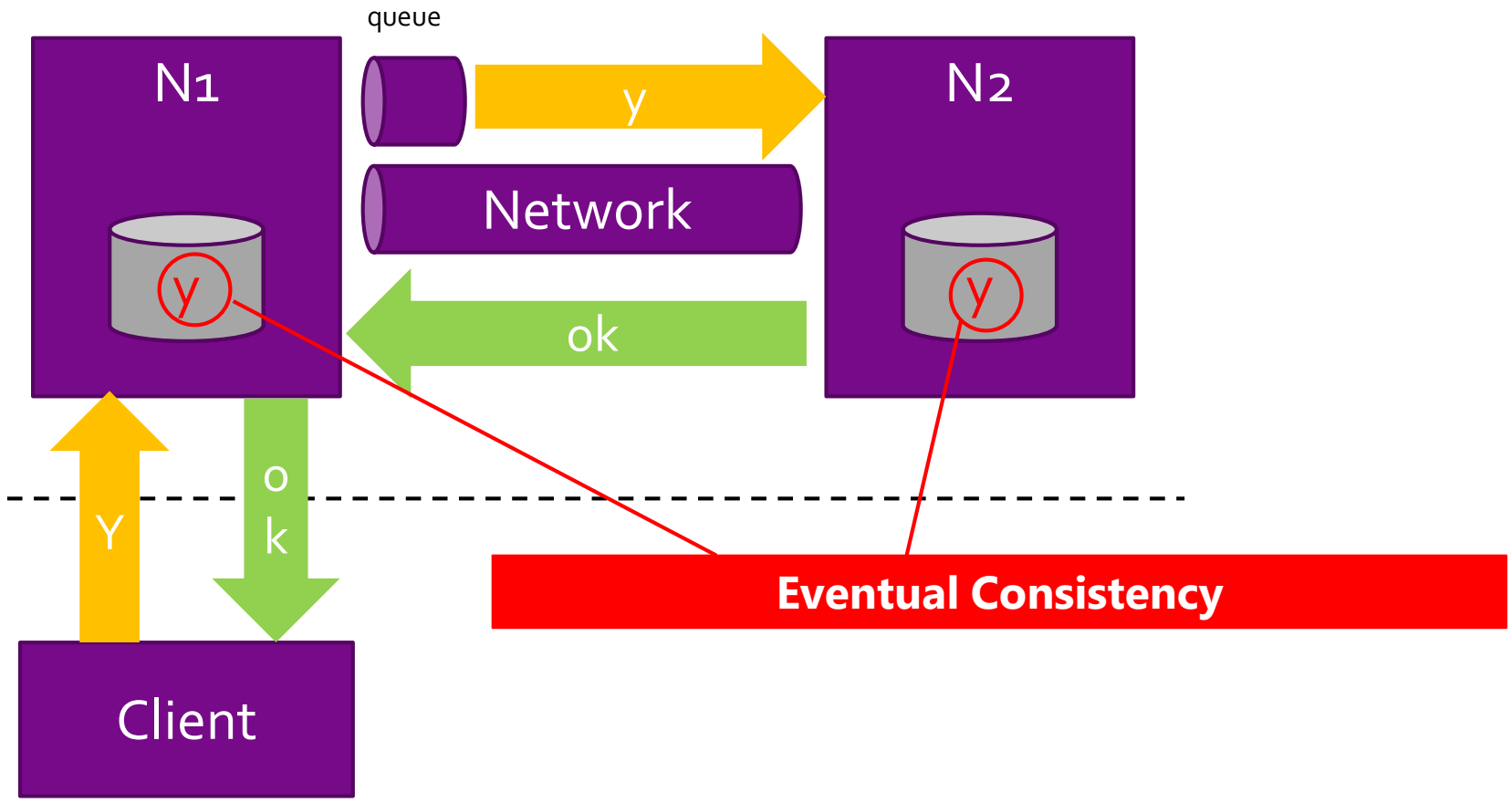
Eventual consistency - AP

Ensures services are always available



Eventual consistency - AP

Ensures services are always available



Eventual consistency

- EC is often not easily accepted
  - "And what about "ACID" and 2PC?"

- Yet, in the "real" world almost every proces is EC
  - Consider whether you really need full consistency when automating business processes
  - Users tend to "get" EC a lot better than we think
  - EC can save you a lot of complexity and trouble (and $)
  - Compensating actions vs. 2PC


Event-based architecture
Cloud computing
distributed computing
Consistency
Eventual consistency
Circuit breaker
Hexagonal architecture
Agile
Partition tolerance
DDD
CQRS
CAP
Domain driven design
Domain modelling
DevOps
Micro services
CQS
Ubiquitous language
CAP
Serverless computing
NOSQL
Bounded context
BASE
Actor model
Event stroming
Bulkhead
Event sourcing
Aggregate
Distributed systems
Web-scale IT
Continuous delivery
Design for failure
Polyglot X
Onion architecture

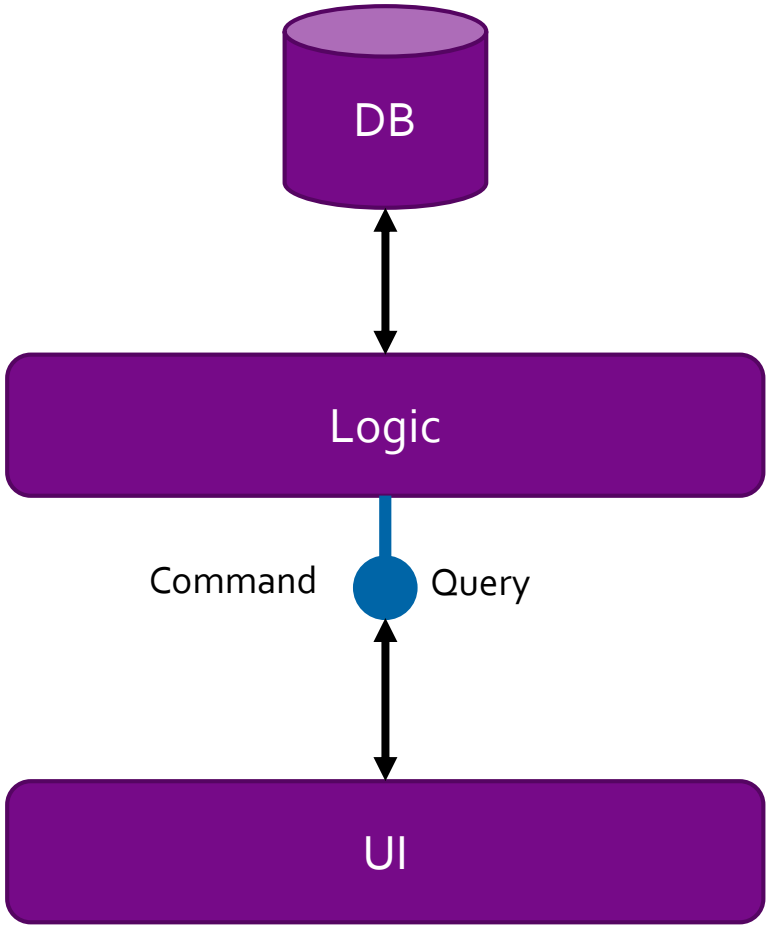
CQRS

- Command Query Responsibility Segregation

- Pattern that embodies strictly separating updates and queries in a system
  - Scale the update and query parts independently
  - Decreases coupling between systems
  - Enables a task oriented approach for your system (commands)

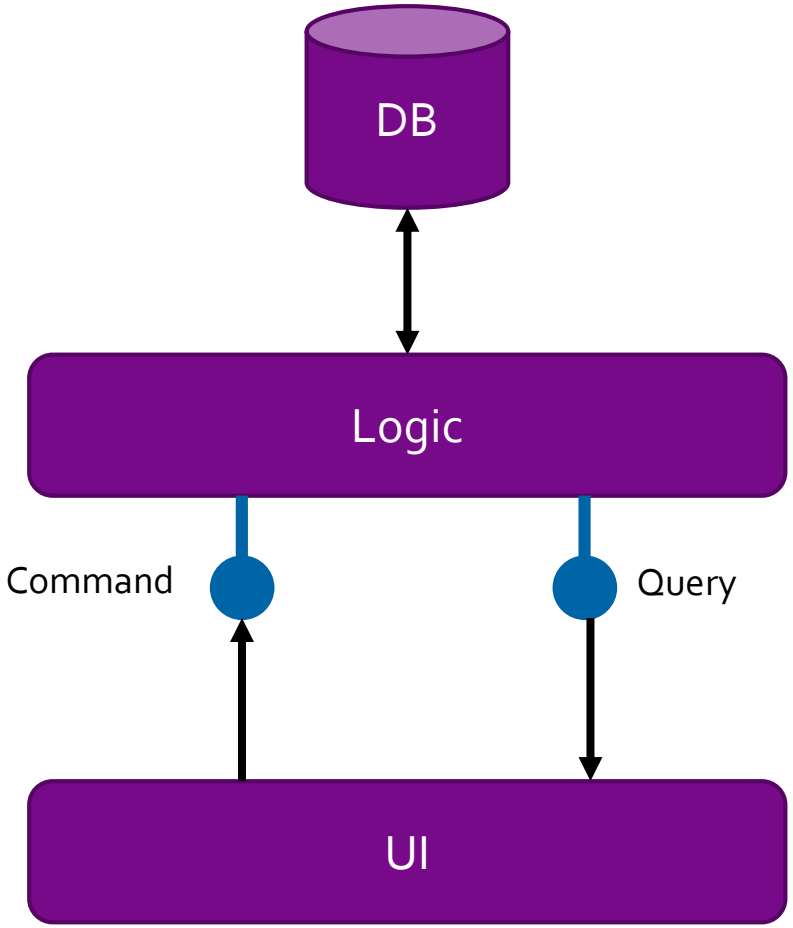
Evolution from SOA to CQRS

Traditional Architecture



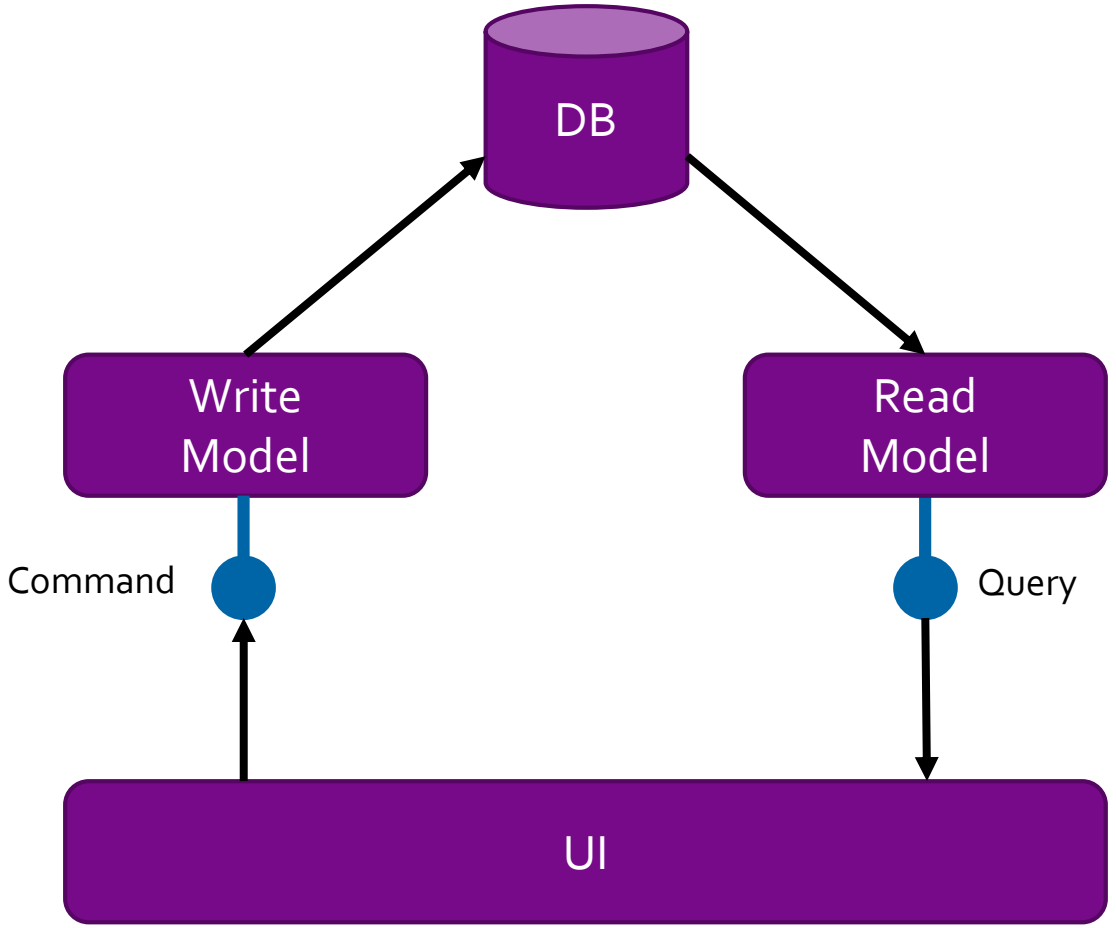
Evolution from SOA to CQRS

CQS

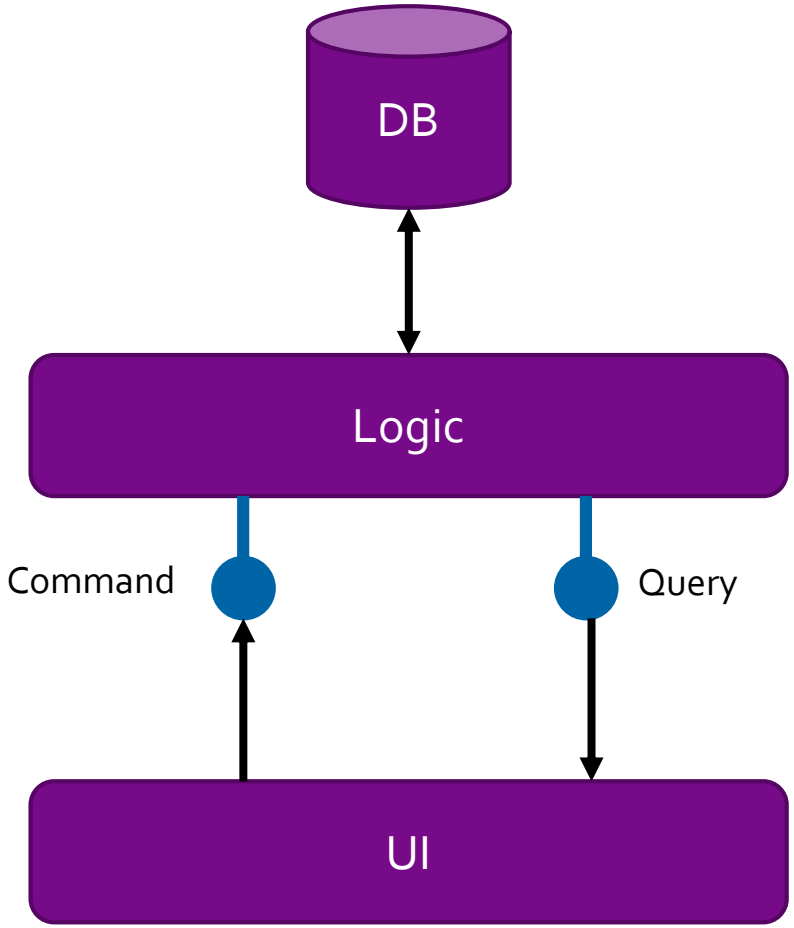


Evolution from SOA to CQRS

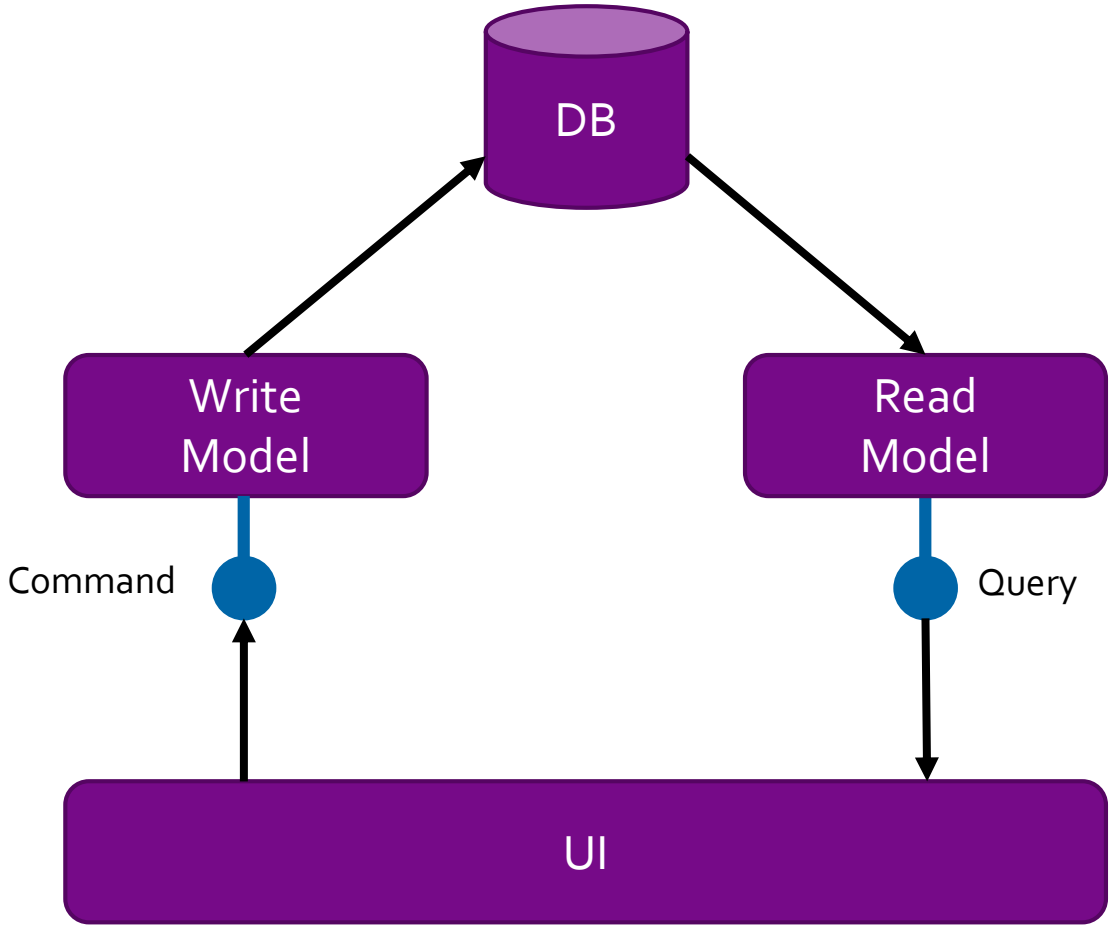
CQRS



Principle vs Pattern



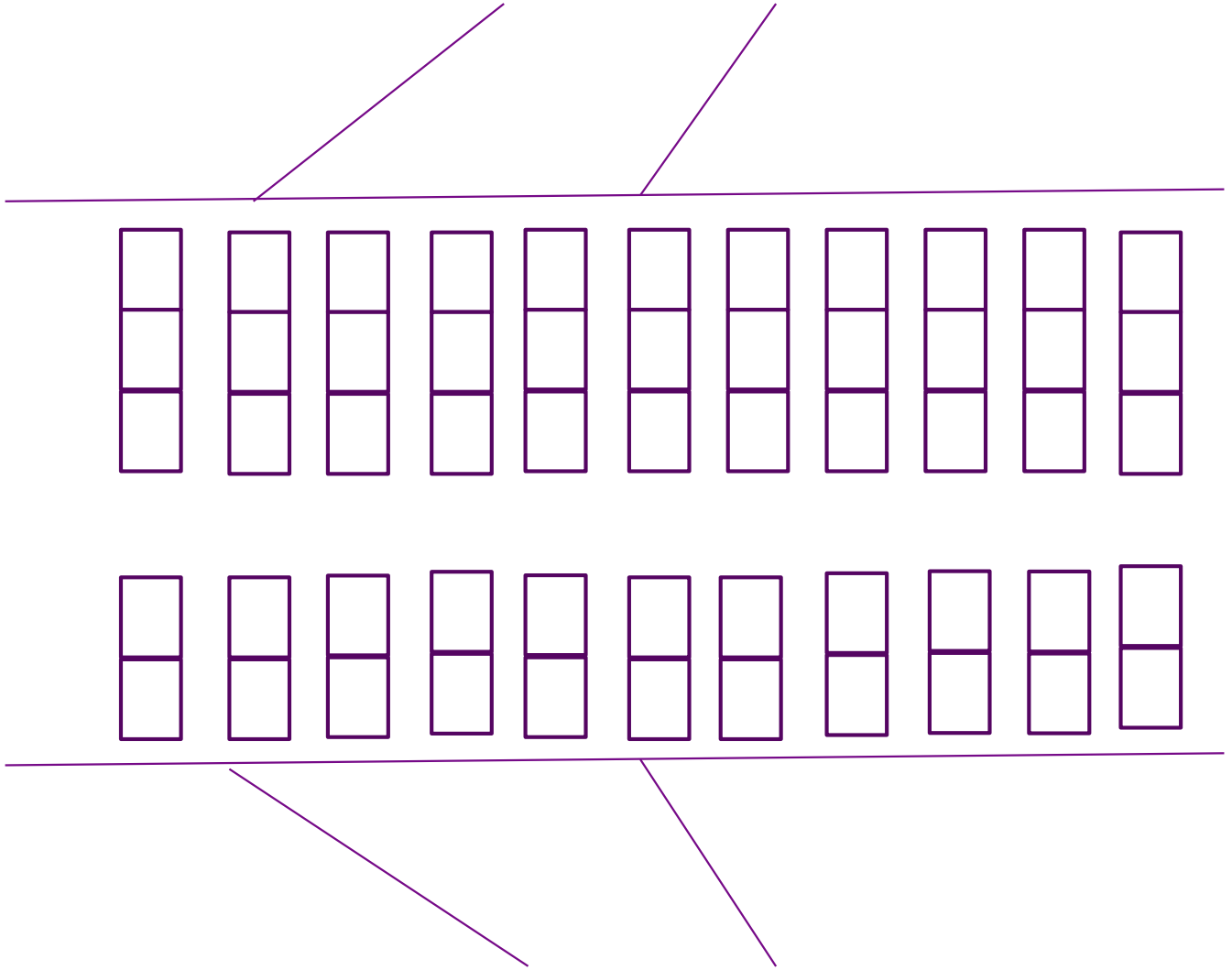
Vs.



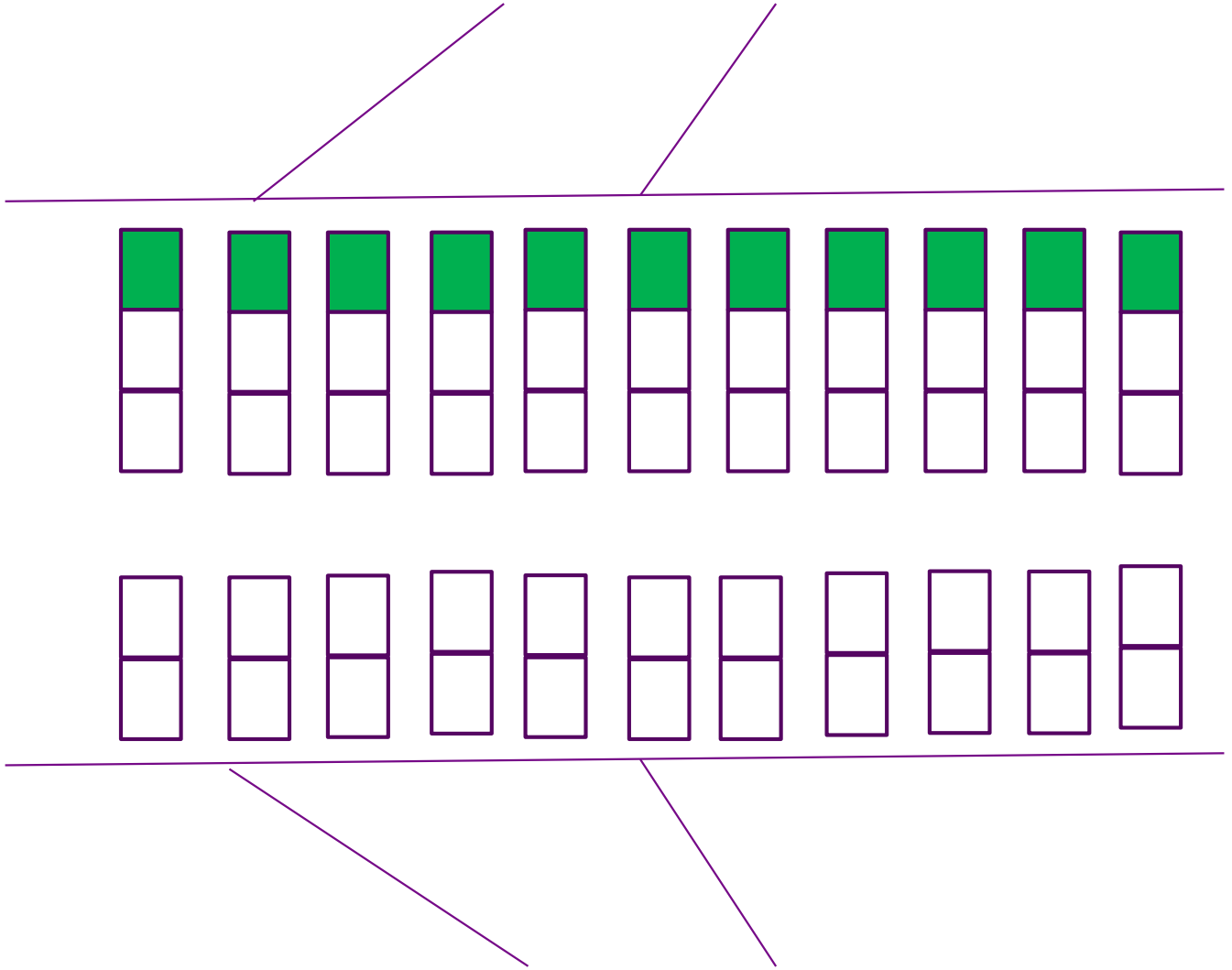
CQRS helps to

Blocking the user when locking the data.

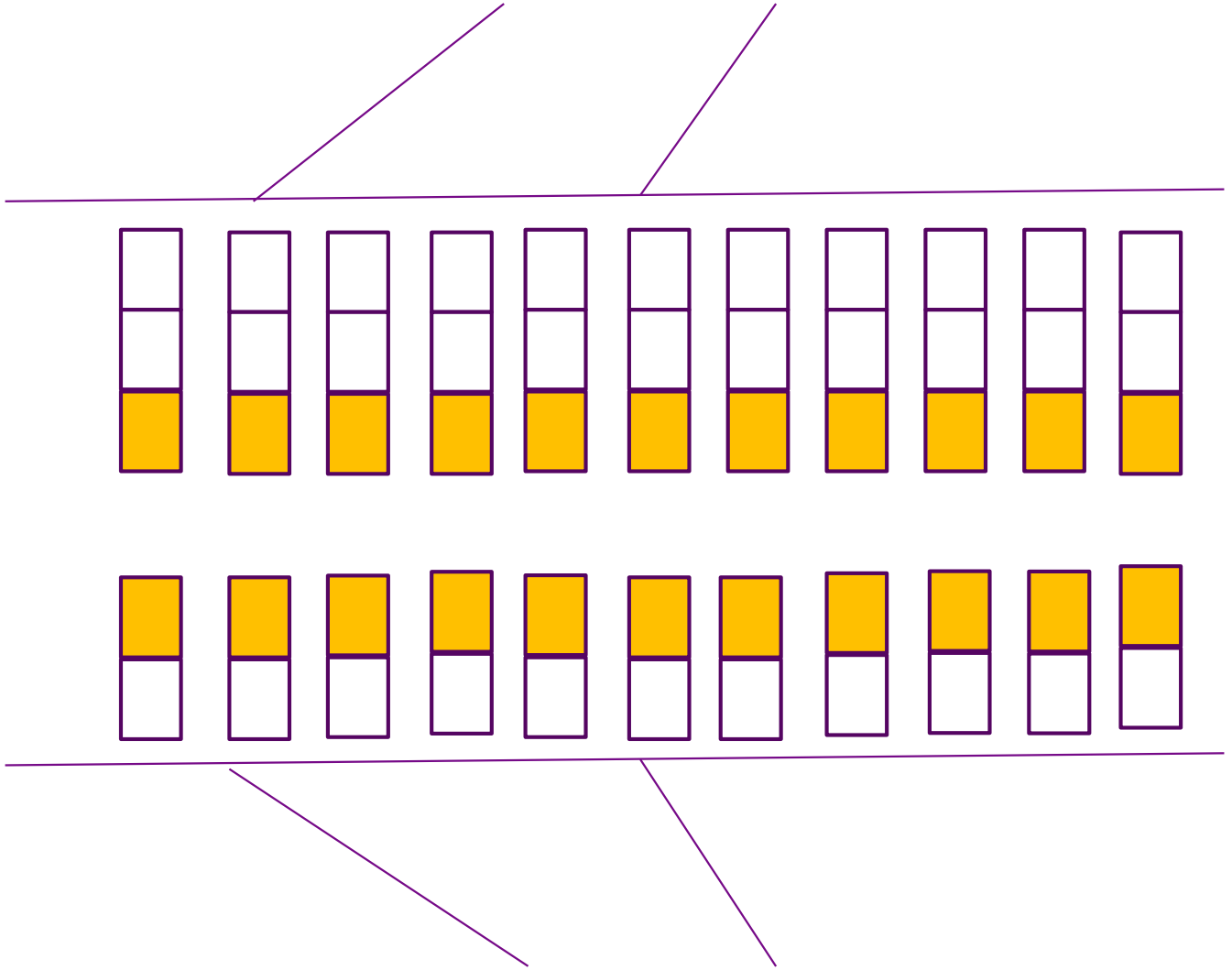
CQRS – Airline reservation



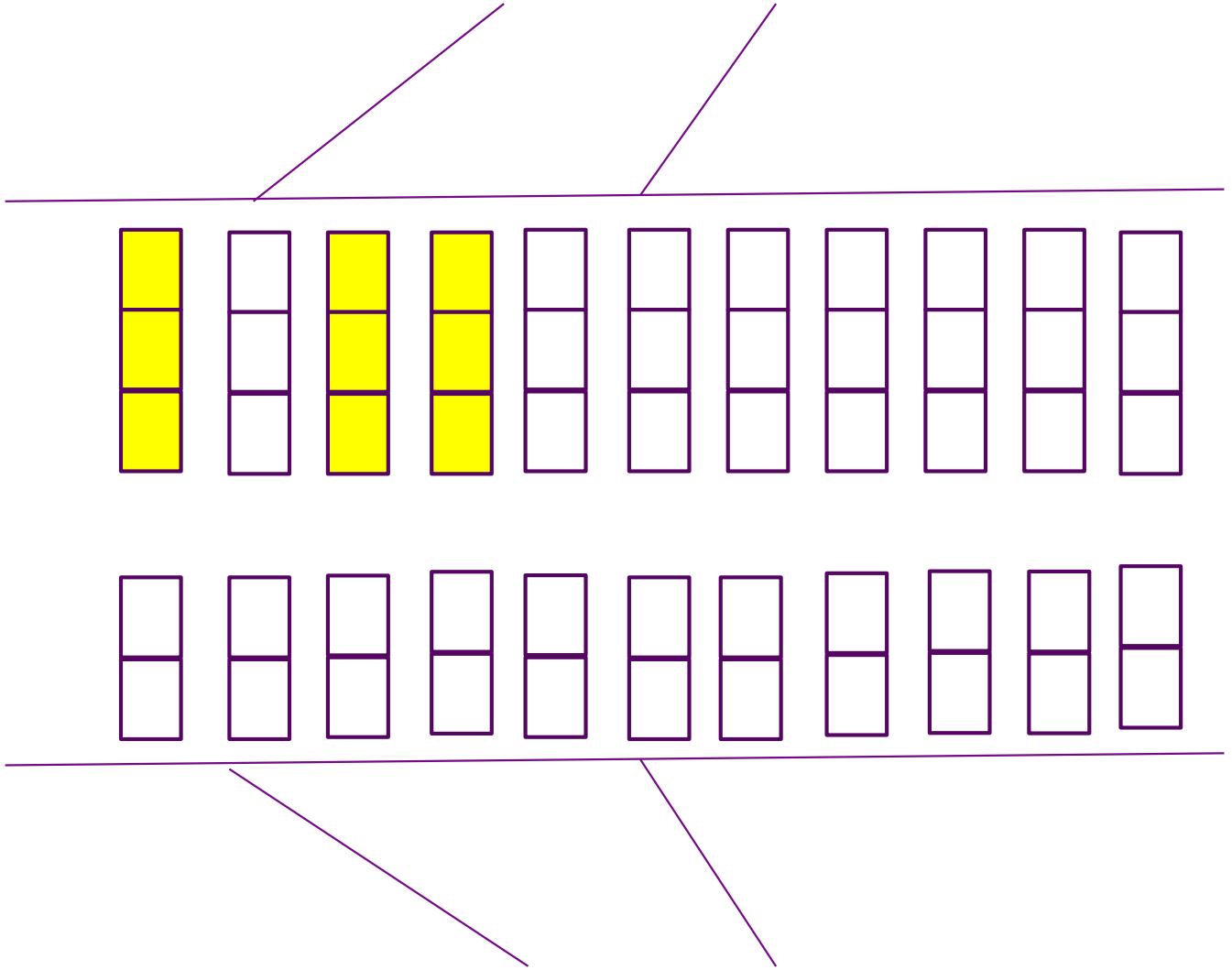
CQRS – Airline reservation



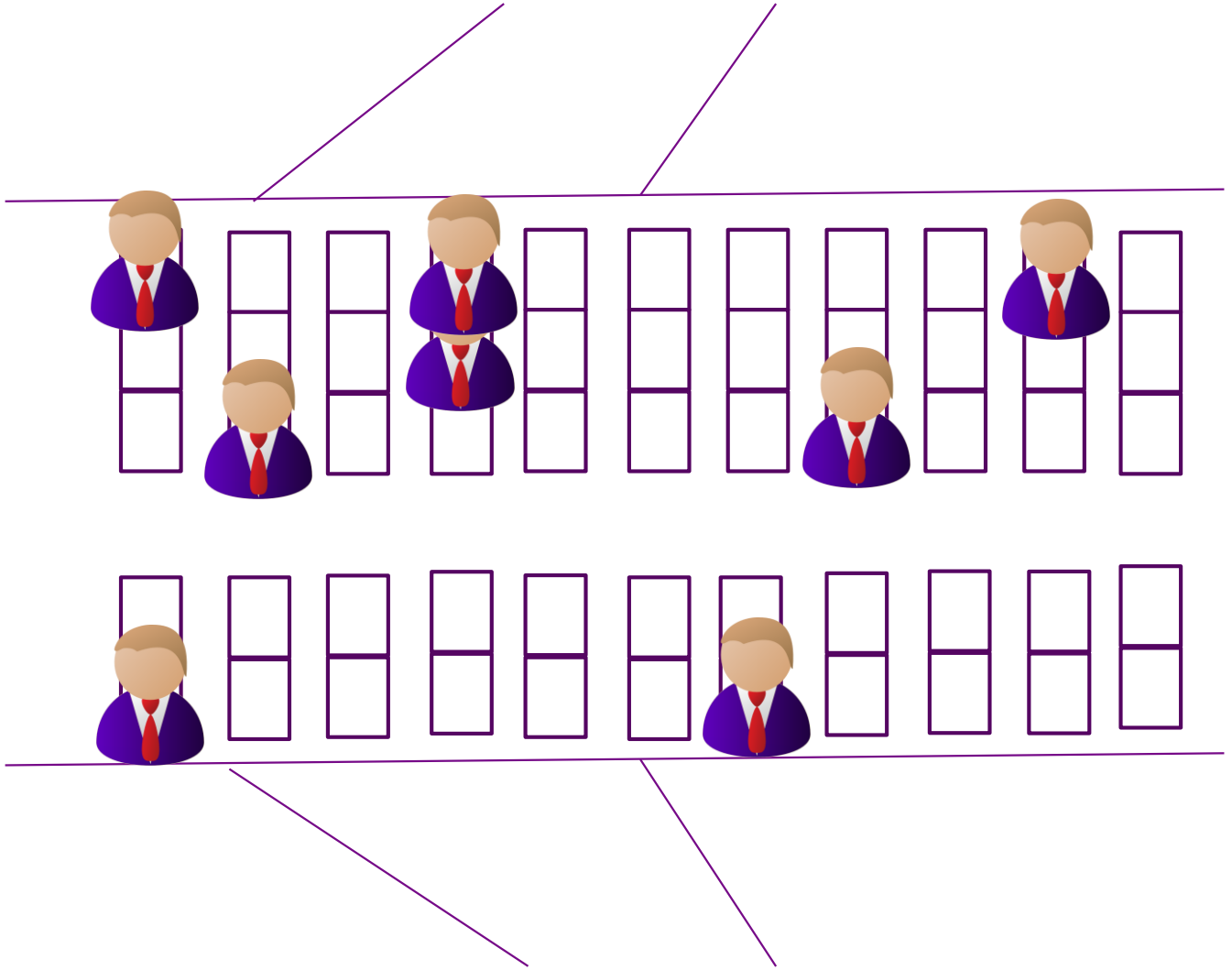
CQRS – Airline reservation



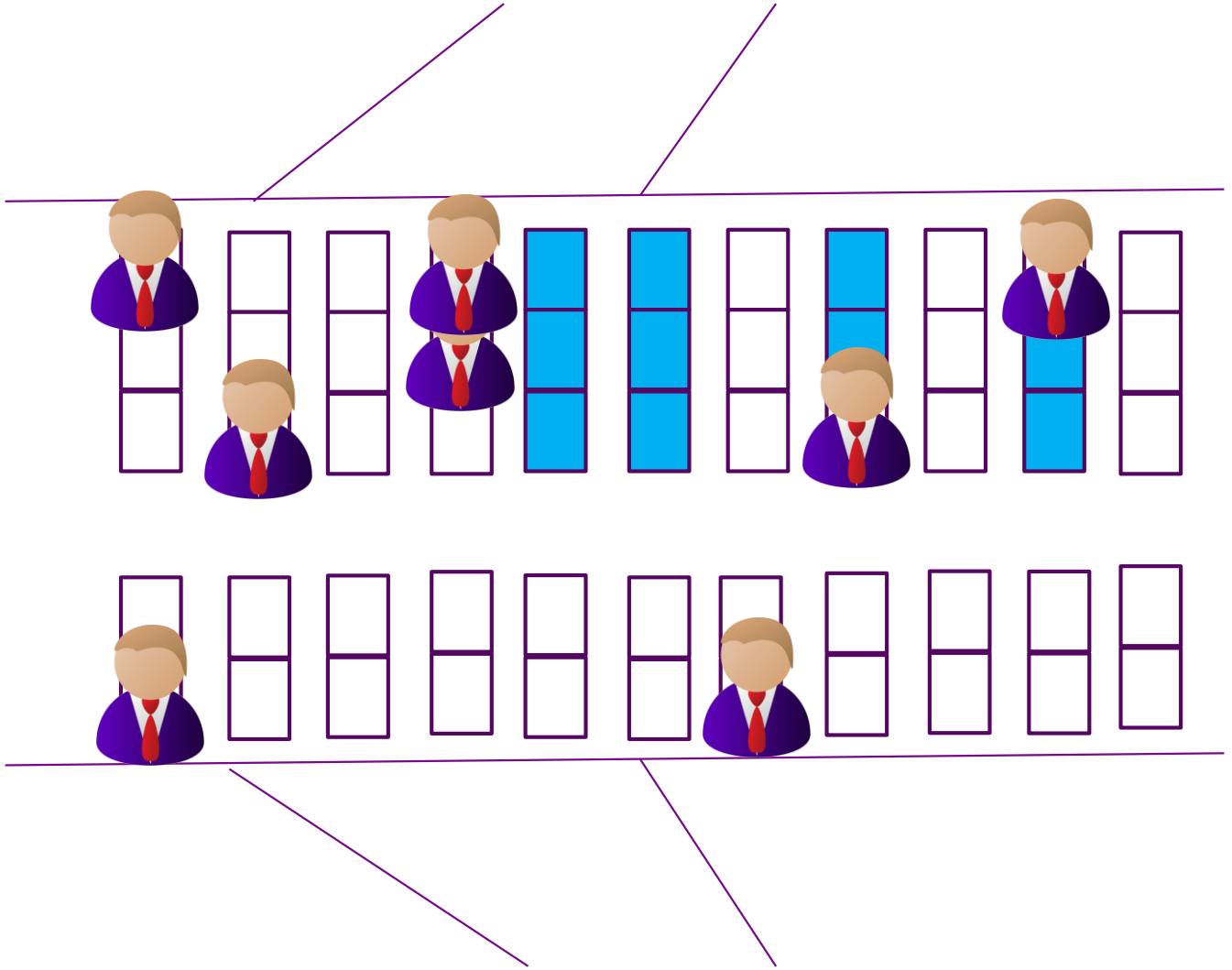
CQRS – Airline reservation



CQRS – Airline reservation



CQRS – Airline reservation



Collaborative domain

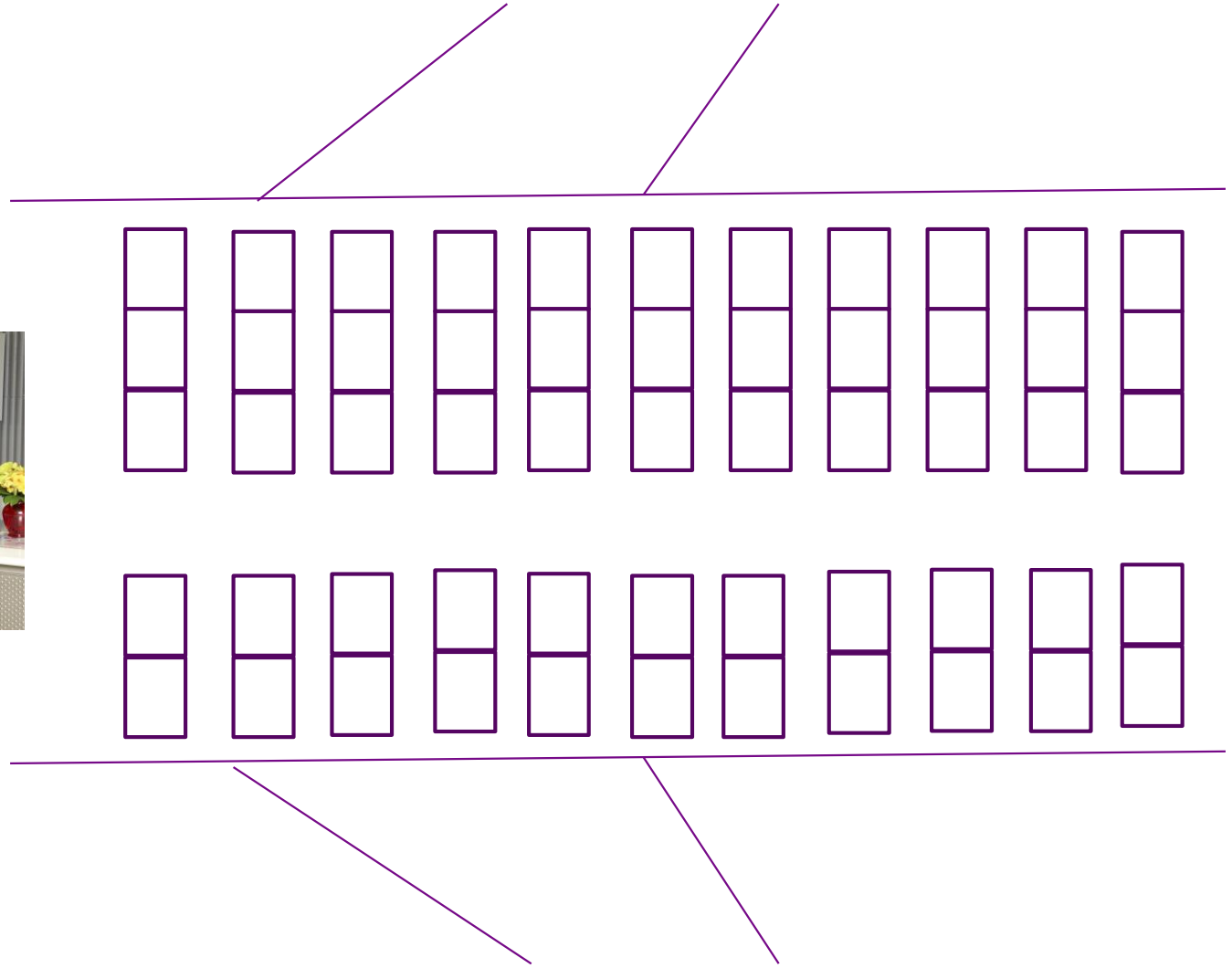
- Large set of people
- Small set of data
- Locking the data is necessary
- Blocking the user is not

CQRS – Ticketing agent

Aisle

3 seats together

Window

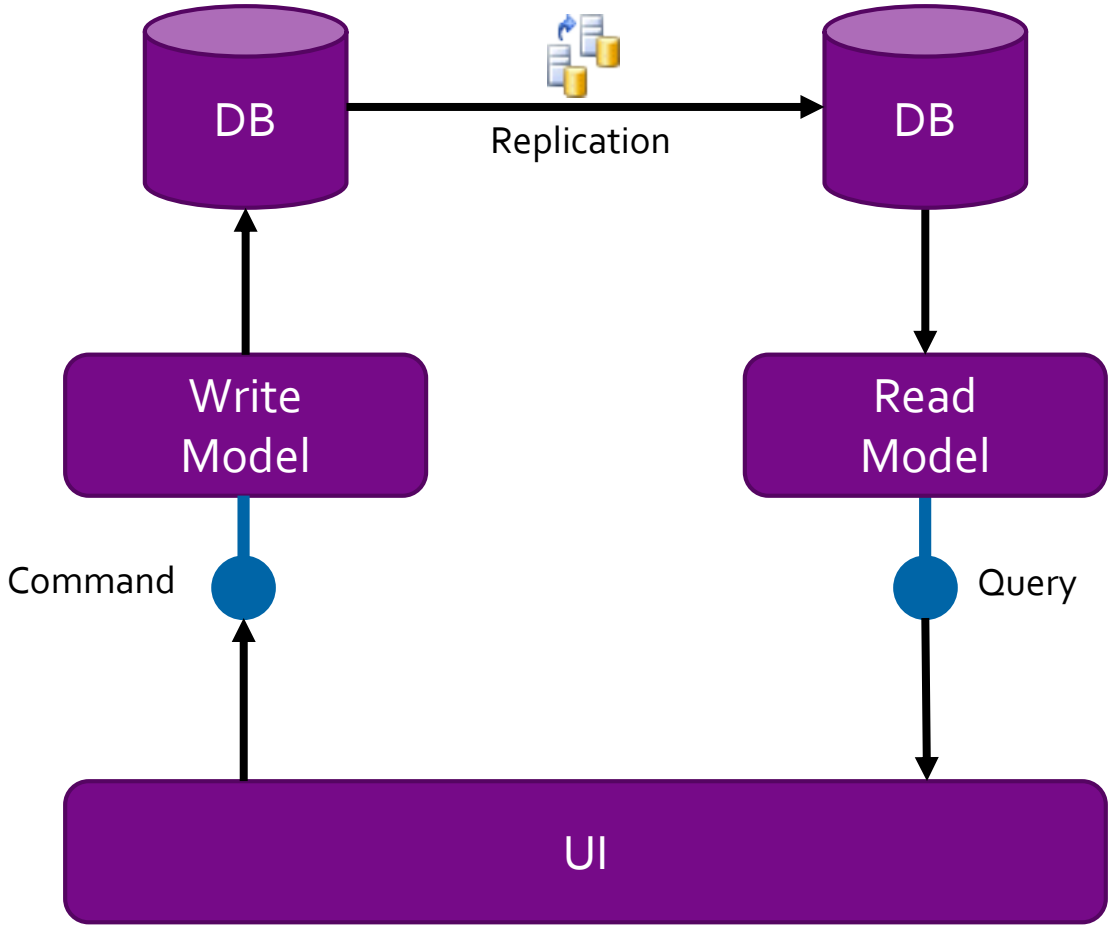


Tradeoff

- No immediate feedback
- Can’t/ don’t want to use everywhere.
- Only for collaborative domains.
  - Large number of people
  - Small set of data.

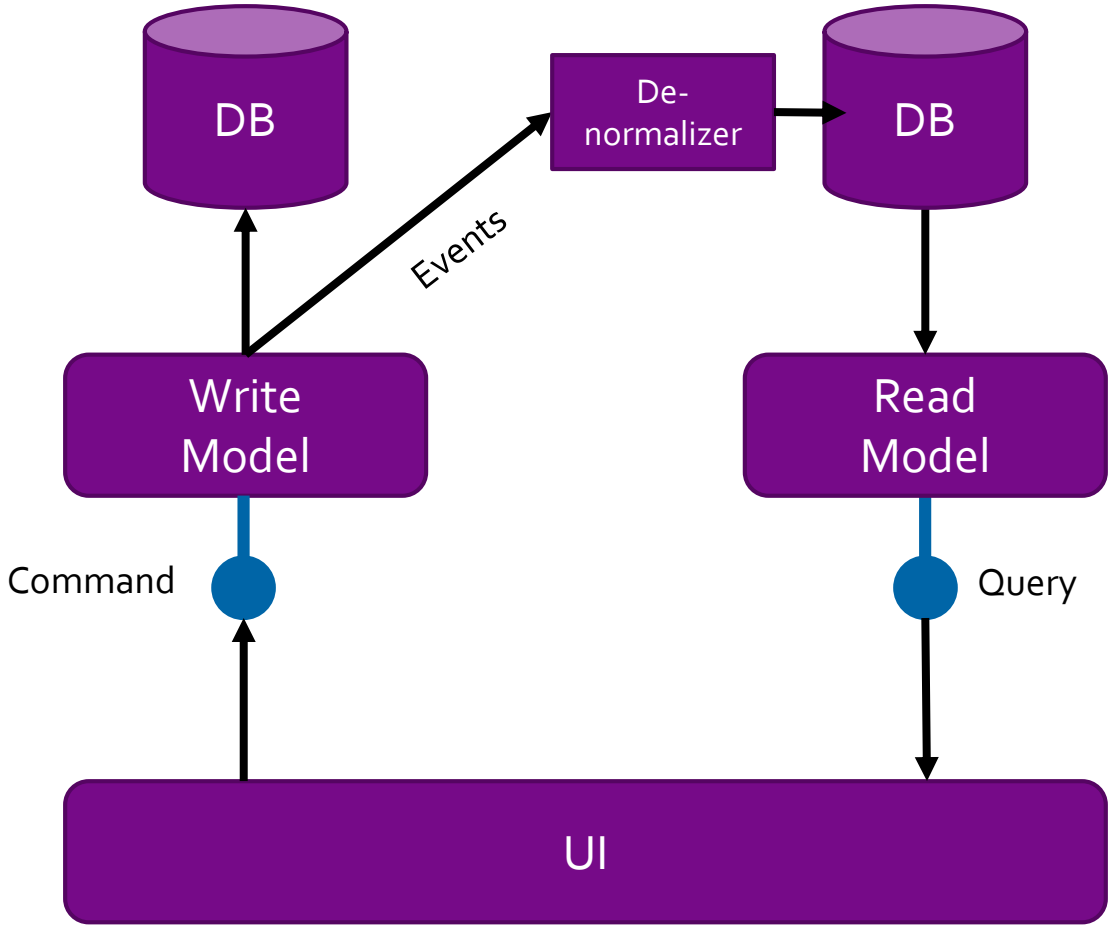
Evolution from SOA to CQRS

CQRS



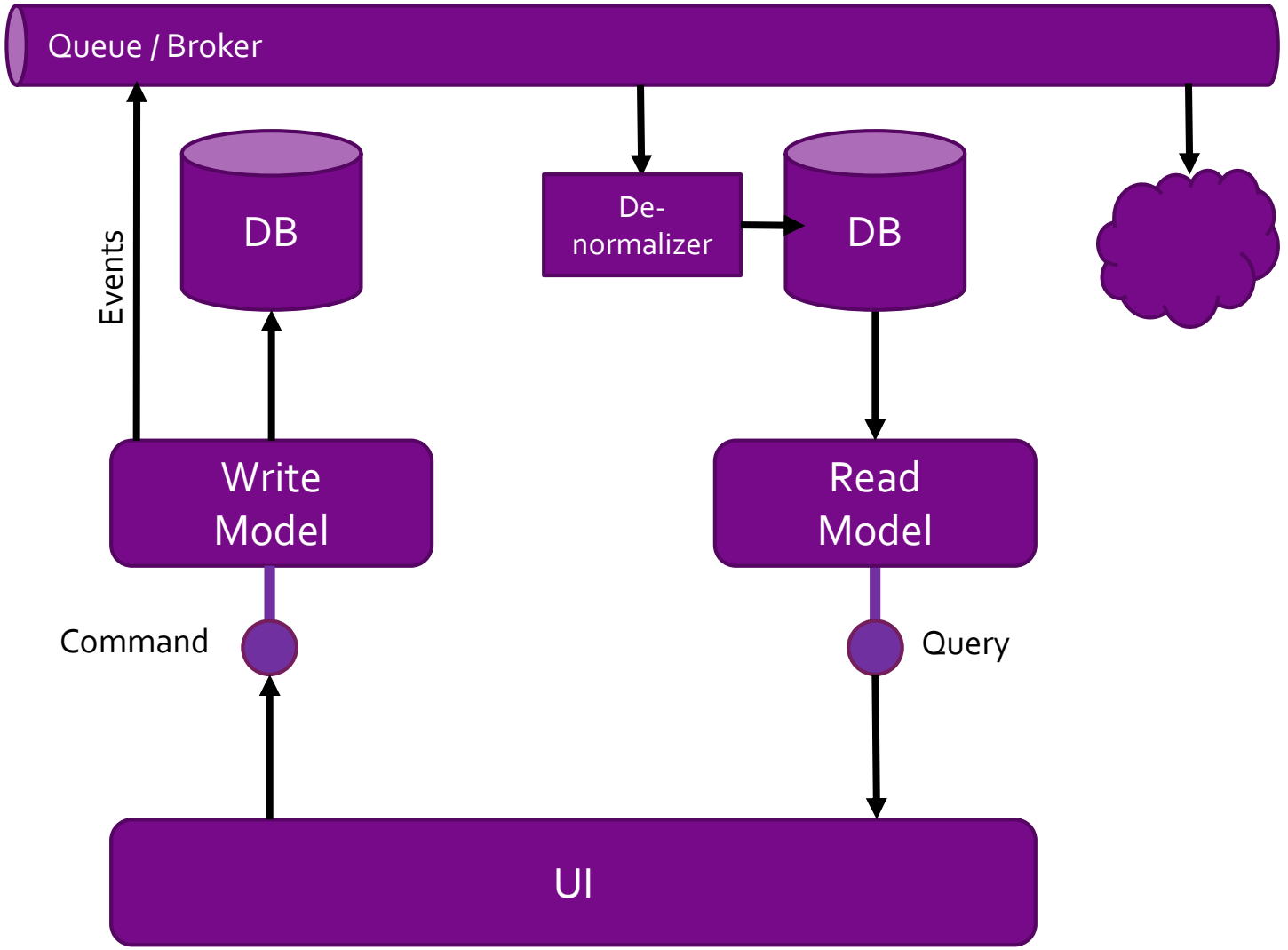
Evolution from SOA to CQRS

CQRS

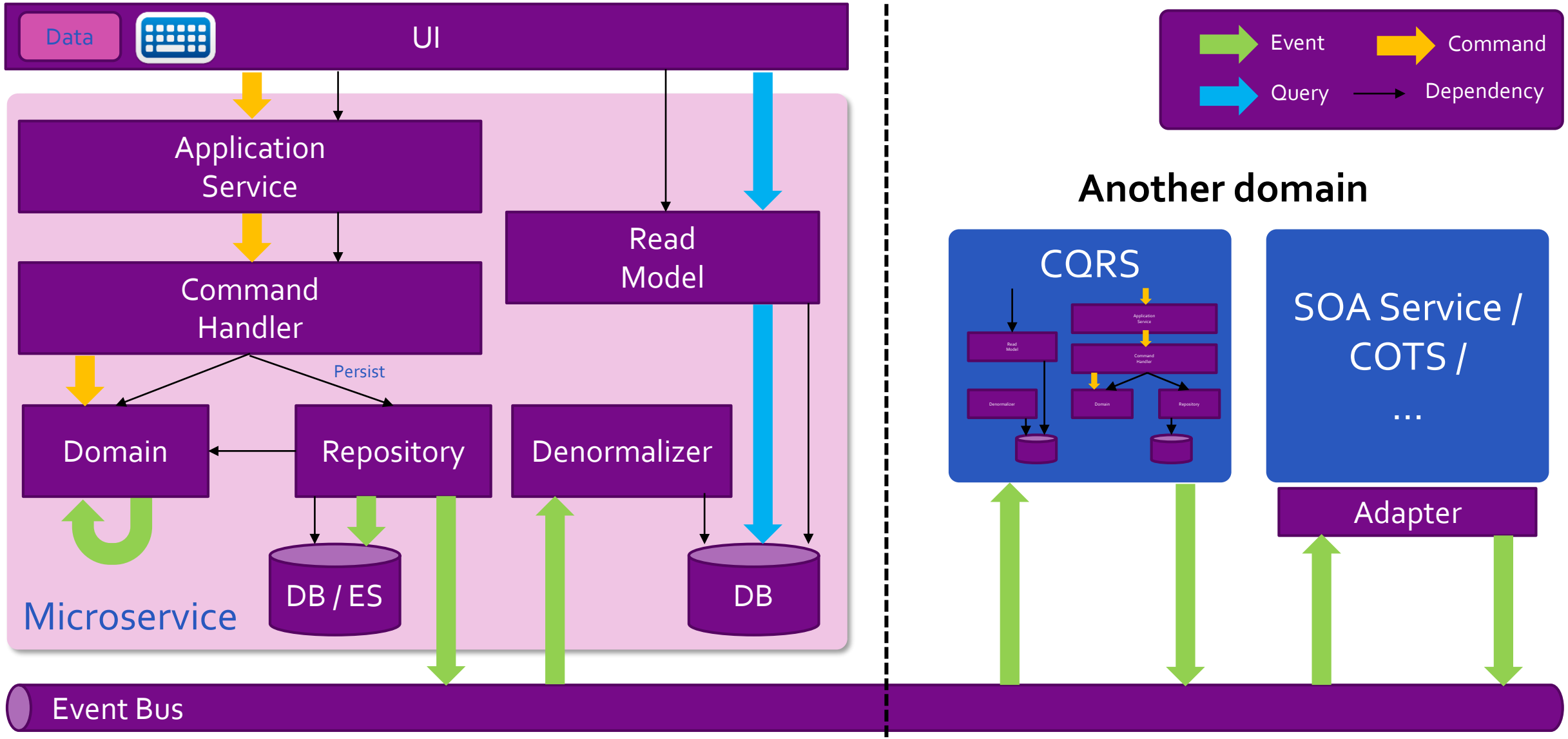


Evolution from SOA to CQRS

CQRS



CQRS - under the covers

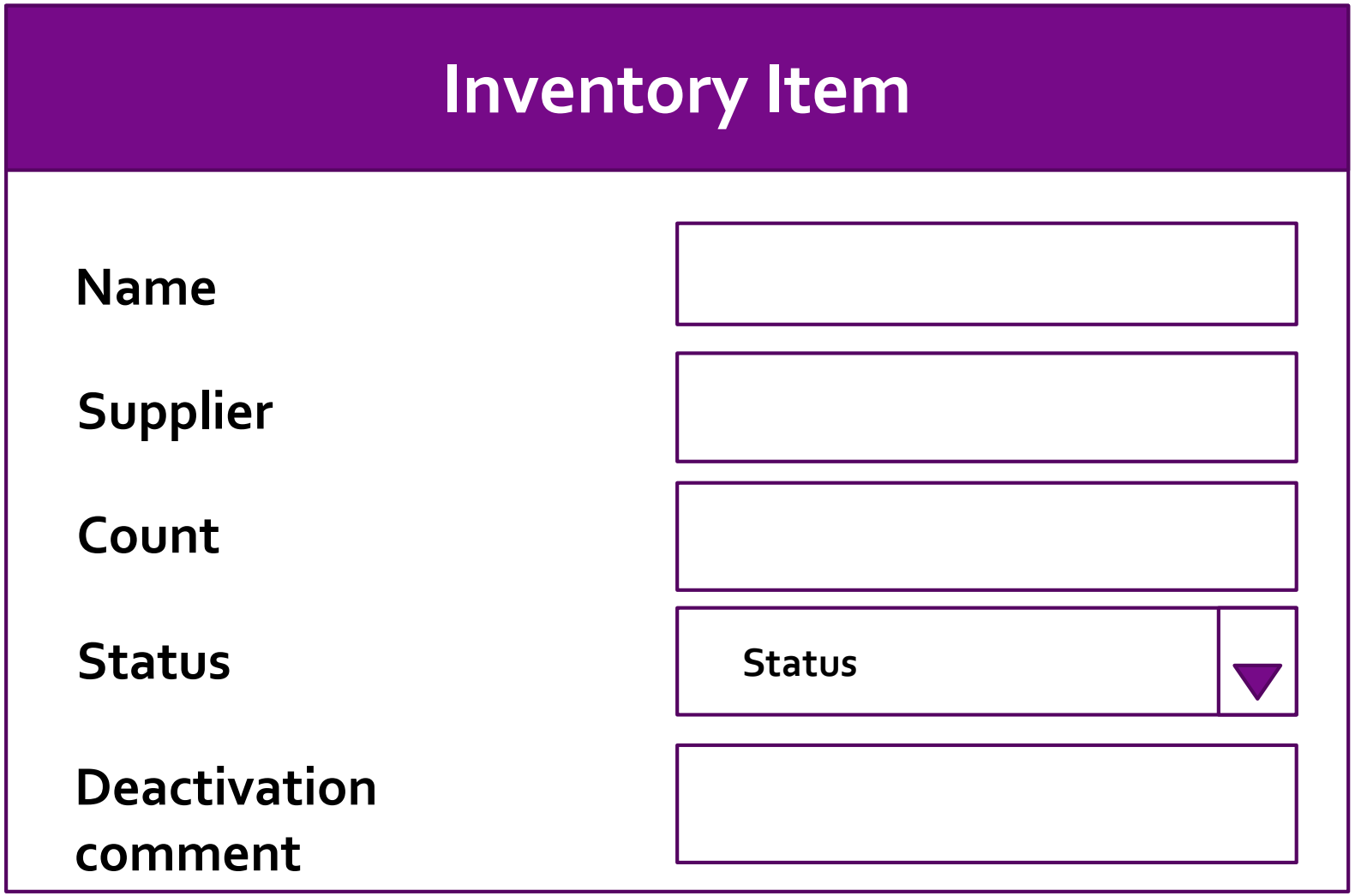


CQRS – CRUD vs. Task based

- Commands are the things that need to be executed
  - Must state business intent
    - So not “UpdateInventory” but “CheckOutItem”
  - Always in the form <Verb><Noun>
  - Can fail (because of business rule / invariants checks)

- Events are things that have happened
  - Always in the form <Noun><Verb (past tense)>
    - CustomerRegistered, ItemCheckedOut, AccountClosed, …

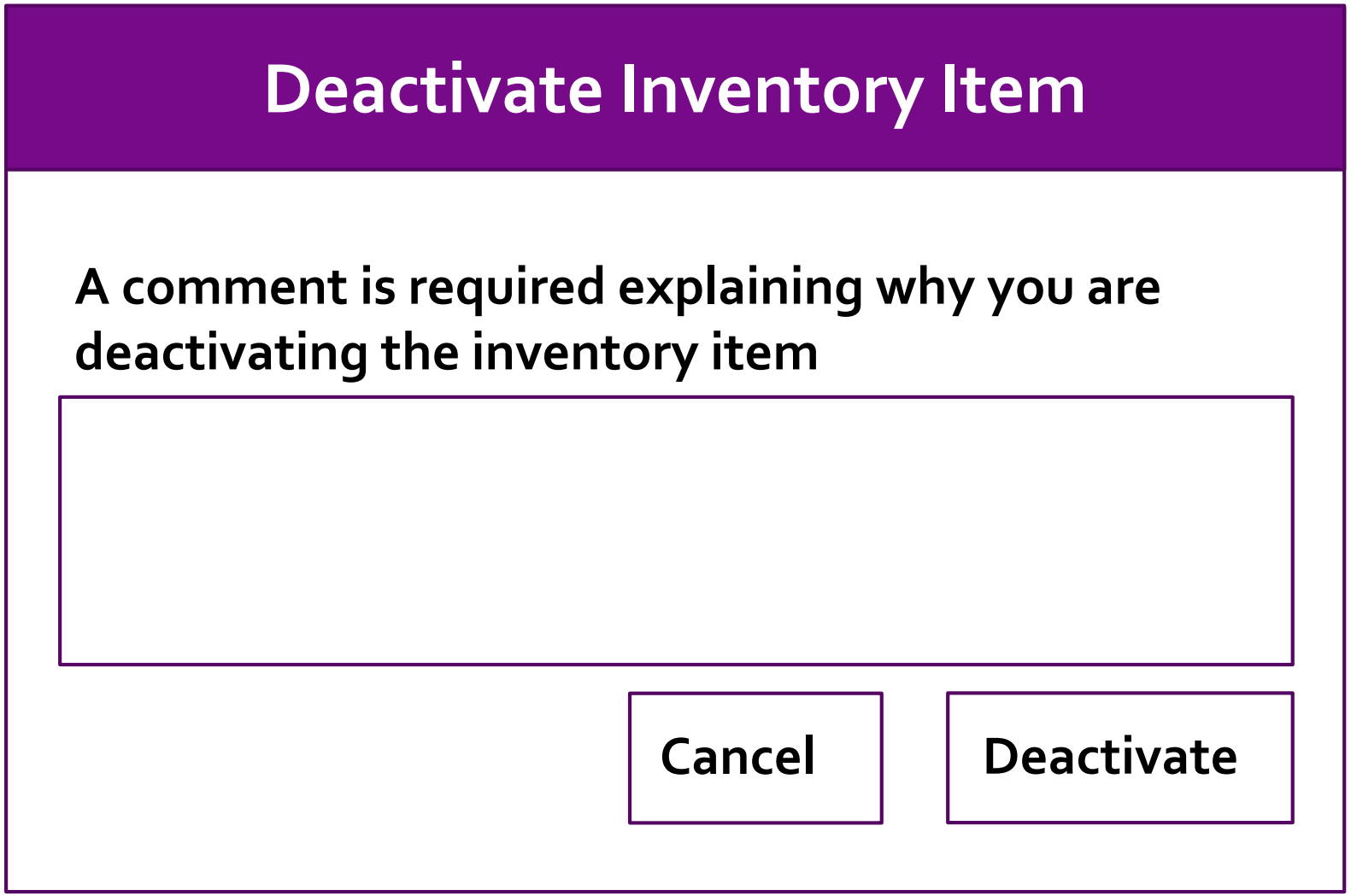
CQRS – CRUD vs. Task based



CQRS – CRUD vs. Task based

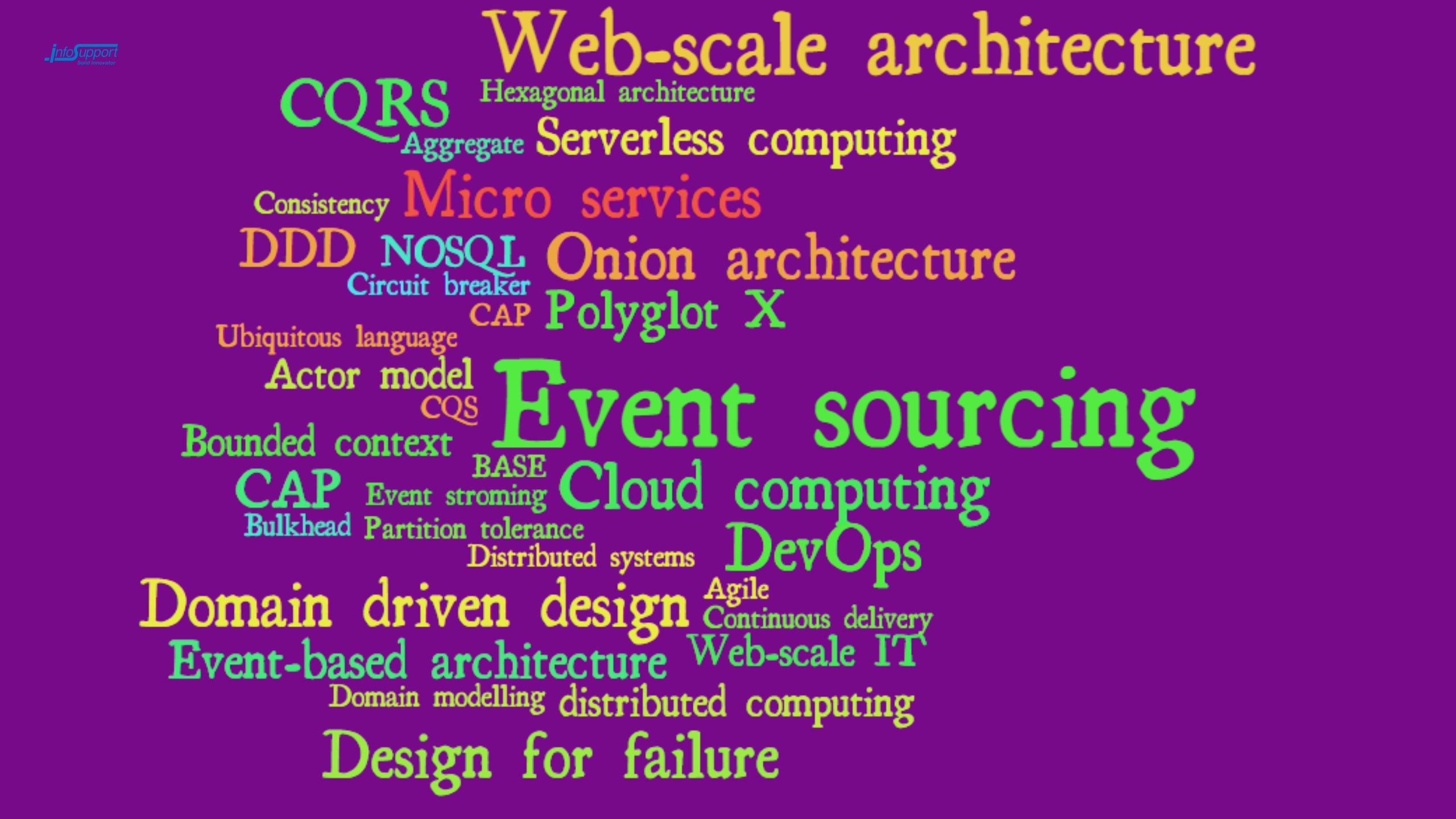
Name	Supplier	Active
TShirts	GAP	
Keyboards	Logitech	
Mouse	Microsoft	
Laptops	Dell	

CQRS – CRUD vs. Task based



CQRS

- Handling a command is a 2 phase process:
  - Check phase
    - Check all invariants and business-rules to make sure the command can be executed
    - External resources or services can be called in this phase
  - Execution phase
    - Update the state of the domain
    - No external resources or services can be called in this phase
    - Events are published
- This separation paves the way for *Event Sourcing*


Web-scale architecture
CQRS
Hexagonal architecture
Aggregate
Serverless computing
Consistency
Micro services
DDD
NOSQL
Onion architecture
Circuit breaker
CAP
Polyglot
X
Ubiquitous language
Actor model
CQS
Event sourcing
Bounded context
BASE
CAP
Event stroming
Cloud computing
Bulkhead
Partition tolerance
DevOps
Distributed systems
Domain driven design
Agile
Continuous delivery
Event-based architecture
Web-scale IT
Domain modelling
distributed computing
Design for failure

Event sourcing

- **This is not how the vast majority of applications work today.**
- Most applications work by storing the current state and using stored states to process business transactions.

Event sourcing – structural representation

Ordered list of items

Payment details

Shipping information

Event sourcing – event representation

Add item 1

Add item 2

Provide payment information

Update item 2

Remove item 1

Provide shipping information

Event sourcing – business decision

- Is it important to store what was added and removed ✓
- Is it important to store when an item was removed from the cart? ✓

Event sourcing

- Event-sourcing is an alternative way of persisting the state of your domain-objects
- Not necessarily normalized in an RDBMS, but as a immutable list of events that have occurred over time
- It's about ensuring that all changes made to the application state during the entire lifetime of the application are stored as a sequence of events.

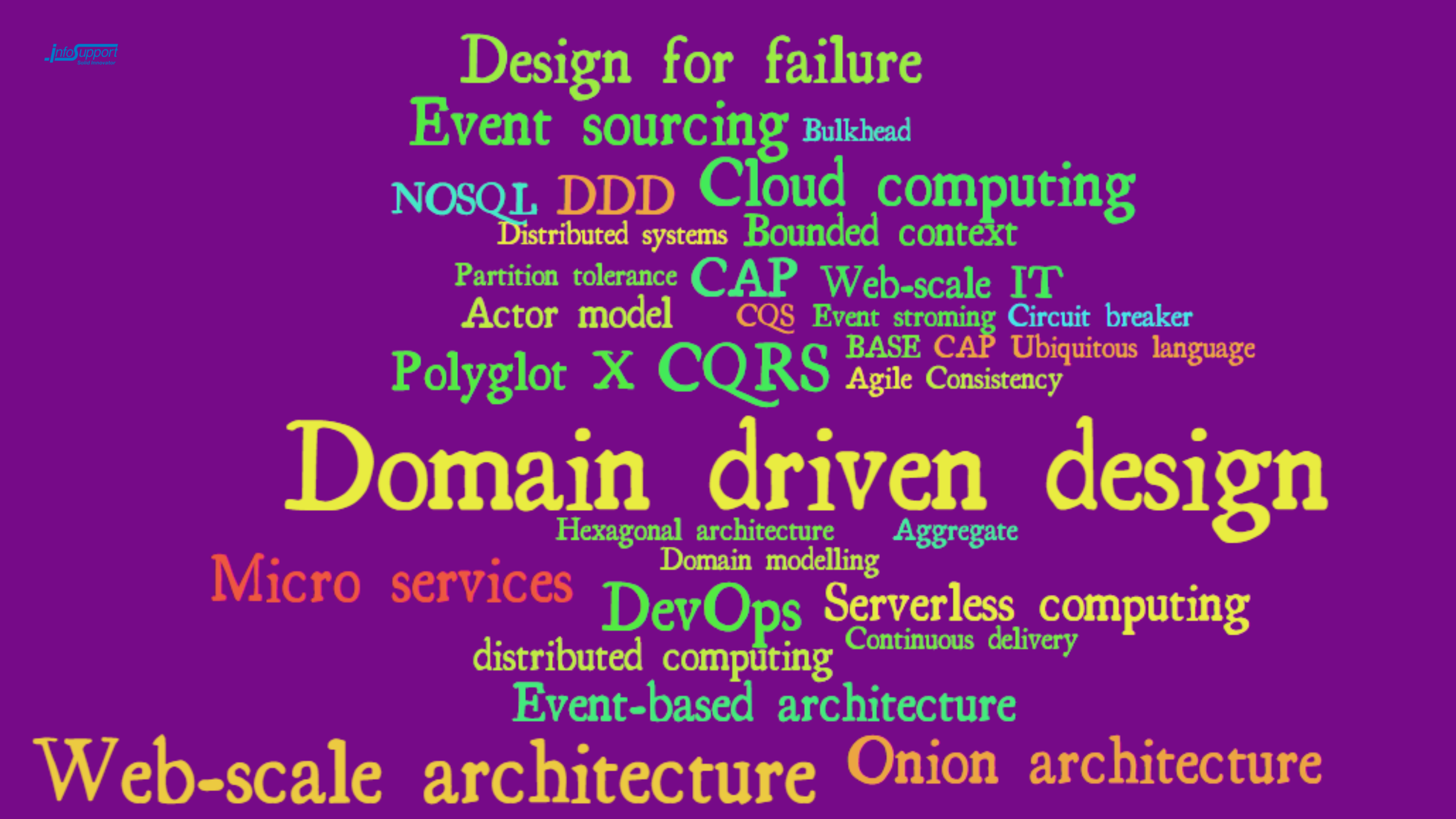
Event sourcing

- Events are immutable and new events only be appended (not be inserted in between)
  - Think accountant's ledger
  - Appending "Correction" events are allowed

- Snapshots can be used to boost performance
  - Only when absolutely necessary
  - Splitting up the domain can eliminate the need for snapshots

Event sourcing

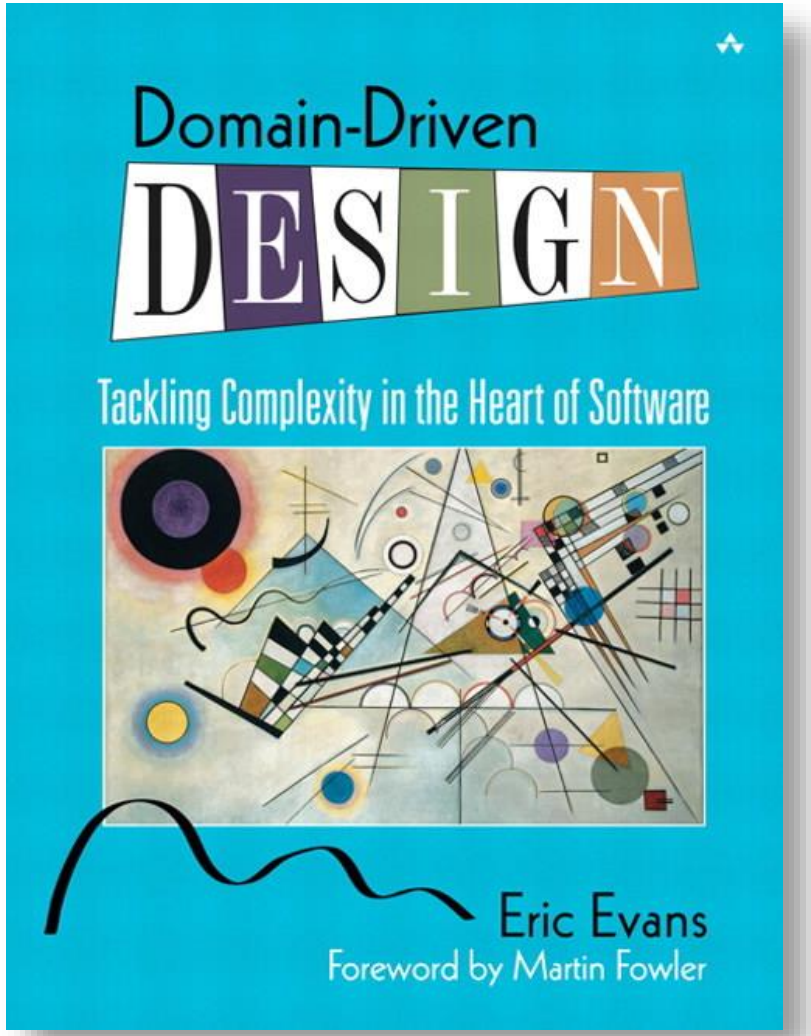
- Append only, so super fast (no locking etc.)
- Ability to completely rebuild the state based on event history
- Ability to analyze behavior that occurred in the past
  - Audit log for free
- State can be built-up by issuing events
  - Simplifies automated testing
- Ability to apply changes in retrospect


Design for failure
Event sourcing
Bulkhead
NOSQL
DDD
Cloud computing
Distributed systems
Bounded context
Partition tolerance
CAP
Web-scale IT
Actor model
CQS
Event stroming
Circuit breaker
BASE
CAP
Ubiquitous language
Polyglot
X
CQRS
Agile
Consistency
Domain driven design
Hexagonal architecture
Aggregate
Domain modelling
Micro services
DevOps
Serverless computing
Continuous delivery
distributed computing
Event-based architecture
Web-scale architecture
Onion architecture

Domain driven design

- Described in Eric Evans' book

Domain Driven Design – Tackling Complexity in the Heart of Software



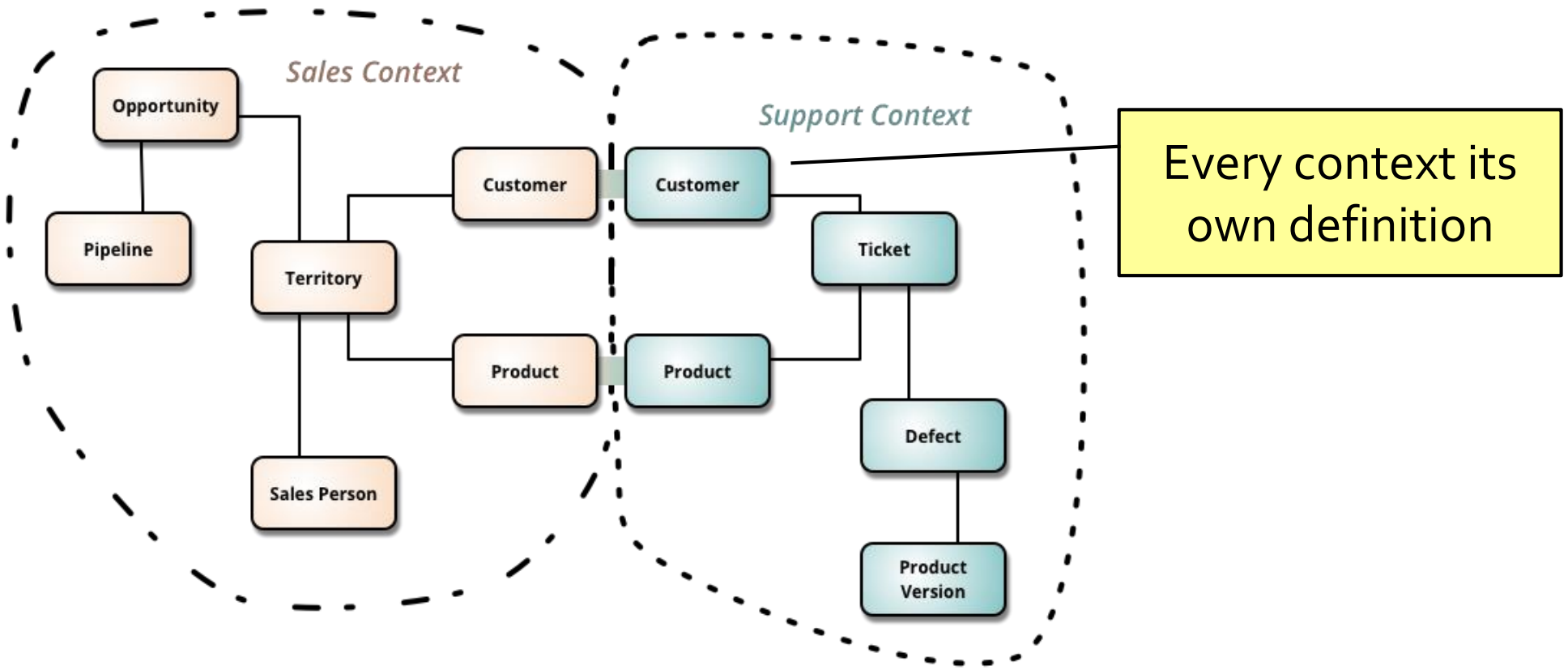
Domain driven design – Core concepts

- Ubiquitous language
- Bounded contexts
- Aggregate roots

Domain driven design – Ubiquitous language

- Building up a common, rigorous language between developers and users
- Used by developers and users for common specifications for the domain they are working on.
- Brings all the team members to same page

Domain driven design – Bounded contexts



Domain driven design - Aggregate

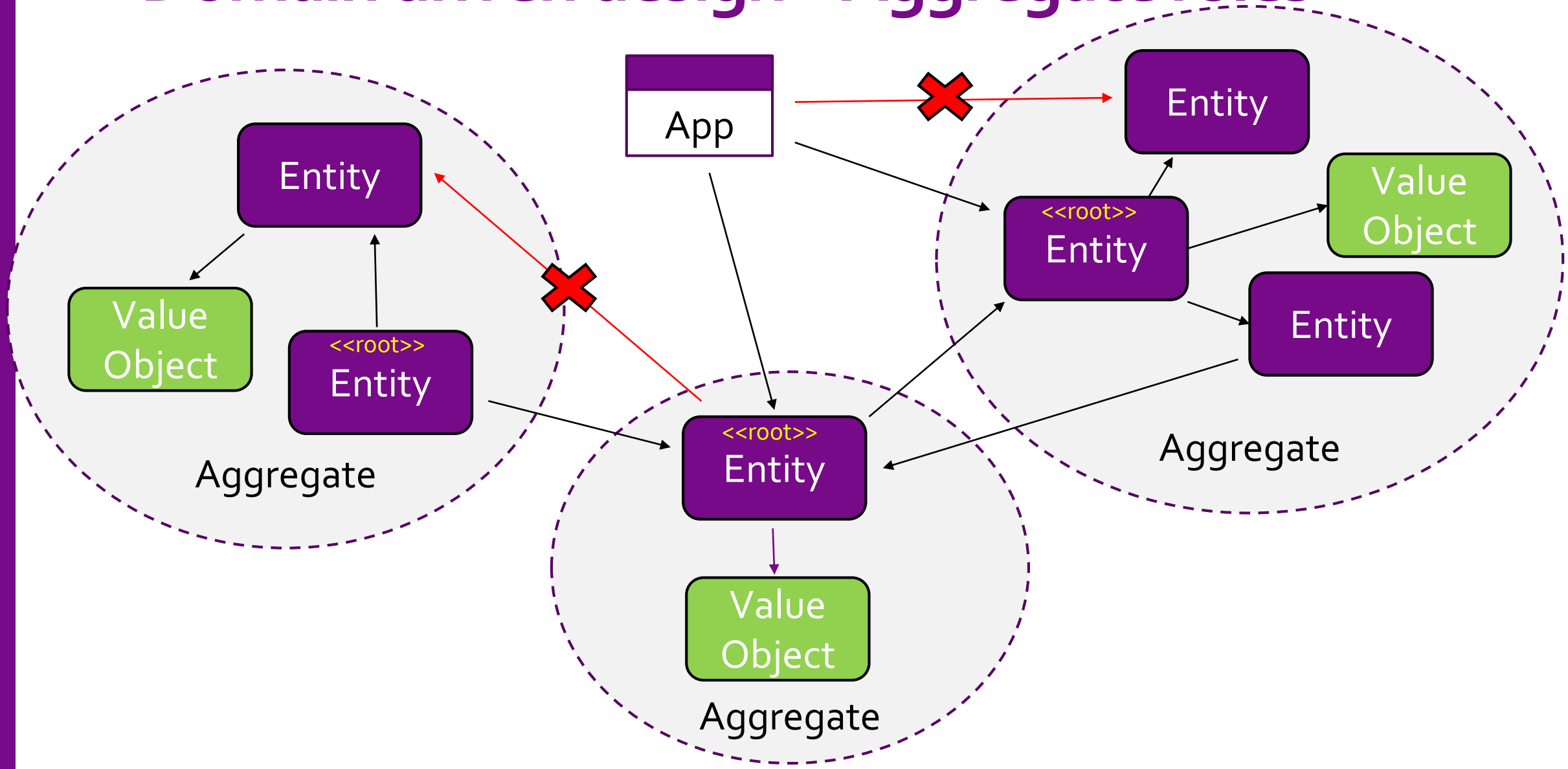
- Domain objects are divided into *Entities* and *Value objects*
  - Entities have an identity
    - Examples: Customer, Product, Contract, Car, …
  - Value objects don't have an identity and are immutable
    - Able to identify based on the value of its attributes
    - Examples: Address, Temperature, Amount ($), …

Domain driven design - Aggregates

- Group entities and value objects into *Aggregates*
  - Set of coherent objects with their relationships

- 1 object is the *Aggregate Root*
  - Only access-point for the aggregate
  - Ensures consistency for the entire aggregate
  - Has a single version# (optimistic concurrency control)

Domain driven design – Aggregate rules



Domain driven design

- Design your system around the business domain
- Do this together with subject matter experts (business analysts / end-users / …)
- Speak the same language in documentation, design and code
  - "Ubiquitous Language"
- Divide the business into *bounded contexts*
  - Autonomous part of the business domain
  - Local definitions of business entities (naming based on the ubiquitous language)
  - Make technical choices per bounded context
    - Polyglot 'X' (best fit for purpose)
    - Architecture style (layered / onion / monolith / CQRS / …)

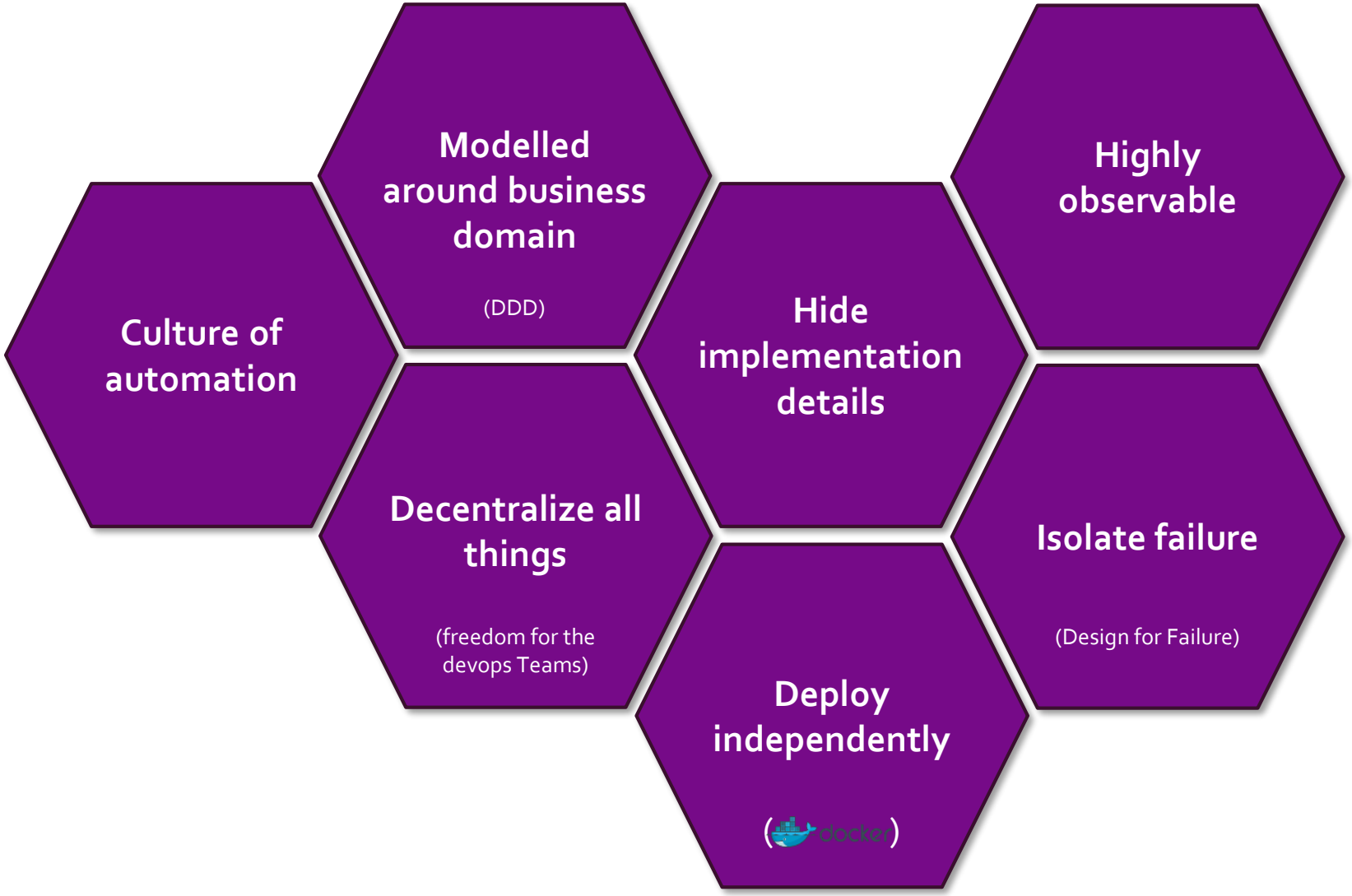

distributed computing
Onion architecture
Event-based architecture
Domain modelling
Actor model
CQS
Web-scale architecture
Domain driven design
CAP
Distributed systems
Serverless computing
Event stroming
Ubiquitous language
Web-scale IT
Micro services
CQRS
Continuous delivery
Bulkhead
Partition tolerance
CAP
Consistency
NOSQL
Aggregate
Hexagonal architecture
BASE
DevOps
Domain driven design
Bounded context
Cloud computing
Circuit breaker
Event sourcing
Polyglot
X DDD
Design for failure
Eventual consistency

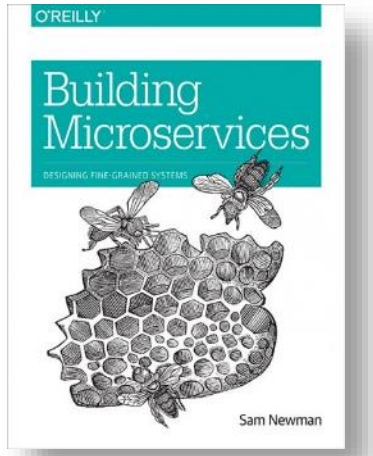
Microservices

- "Small" autonomous services that cooperate
  - Designed around business domains /capabilities (DDD bounded contexts)
  - Simple to scale-out
  - High cohesion / low coupling

- A Microservice is specialized in 1 thing
  - Single responsibility principle

Microservices principles

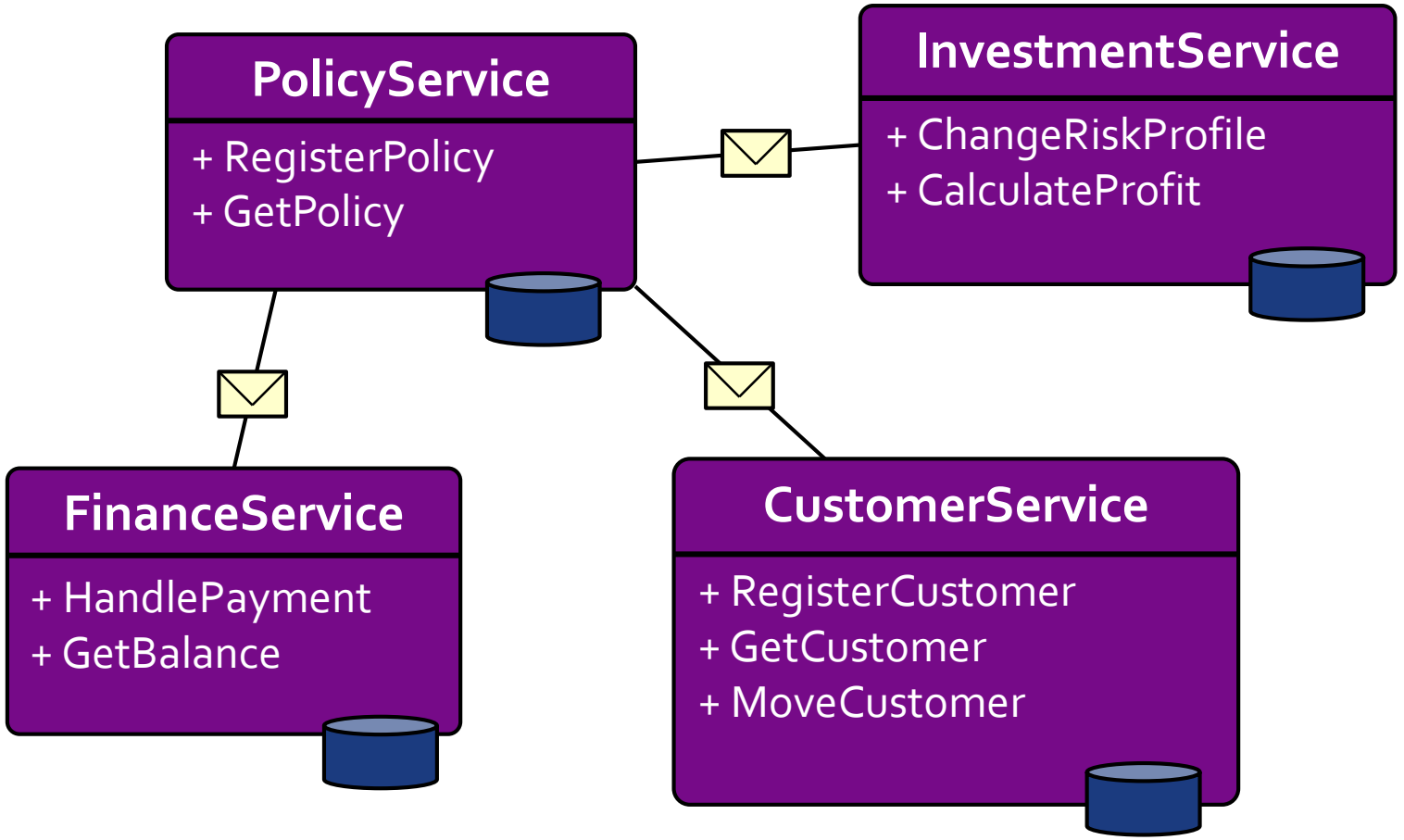




Microservices

Traditionally (SOA)

Microservices



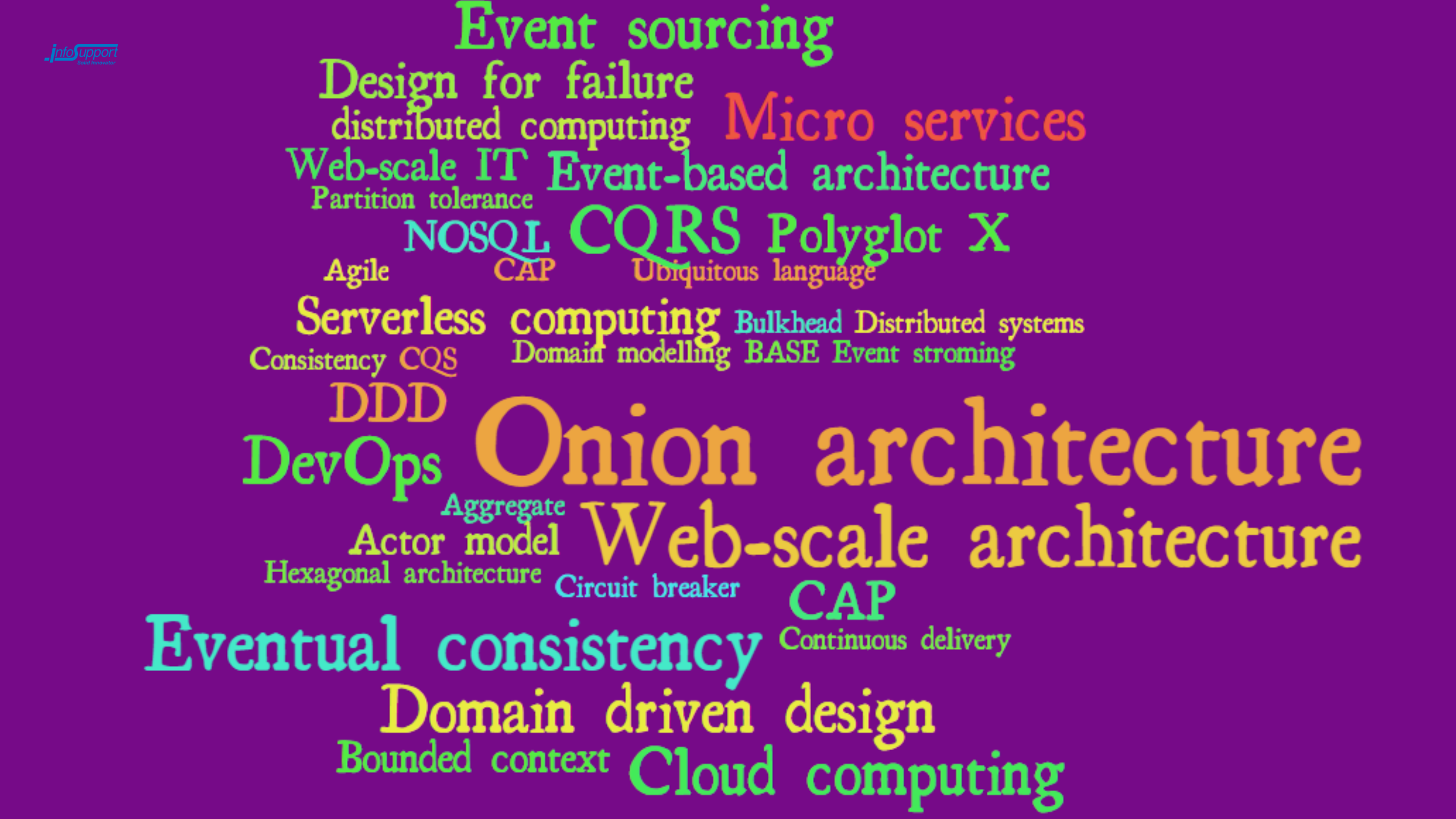
Microservices

- Microservices communicate using “lightweight” protocols
  - HTTP (Rest API + JSON) / TCP + ProtoBuf / Own implementation
  - Choose between *open* or *fast*
- Primarily asynchronous communication

Microservices

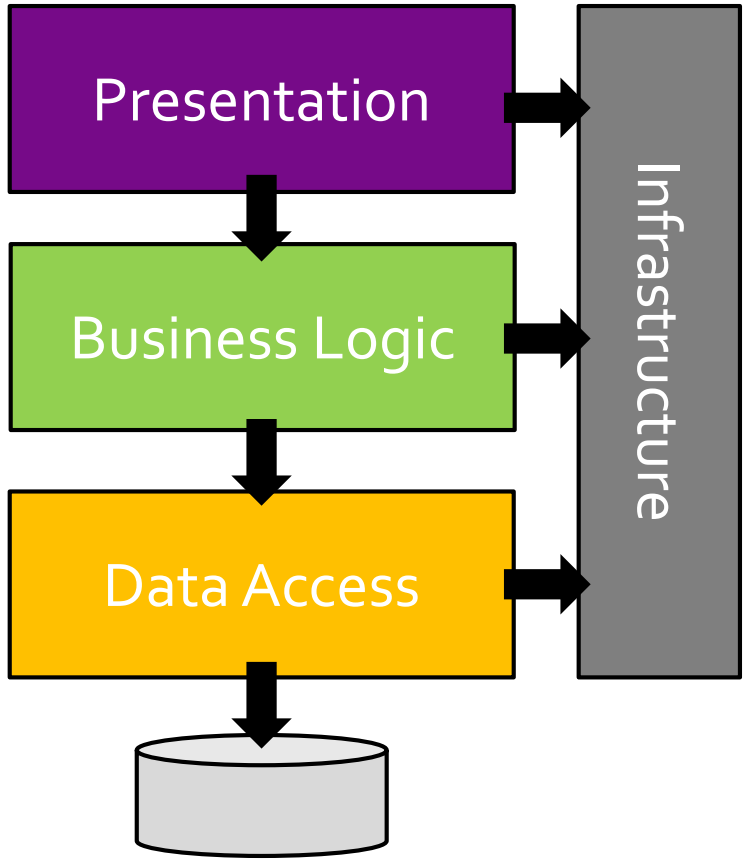
- Because of Microservice autonomy, certain technology-choices can be made per service:
  - **Architecture:** 3-tier | CQRS | Monolith | …
  - **Language/runtime:** Java | C# | Scala | NodeJS | …
  - **Persistence:** File System | MongoDB | Cassandra | …

- This increases flexibility and makes sure you can choose the best solution per service (polyglot)


Event sourcing
Design for failure
distributed computing
Micro services
Web-scale IT
Event-based architecture
Partition tolerance
NOSQL
CQRS
Polyglot
X
Agile
CAP
Ubiquitous language
Serverless computing
Bulkhead
Distributed systems
Consistency
CQS
Domain modelling
BASE
Event stroming
DDD
DevOps
Onion architecture
Aggregate
Actor model
Web-scale architecture
Hexagonal architecture
Circuit breaker
CAP
Eventual consistency
Continuous delivery
Domain driven design
Bounded context
Cloud computing

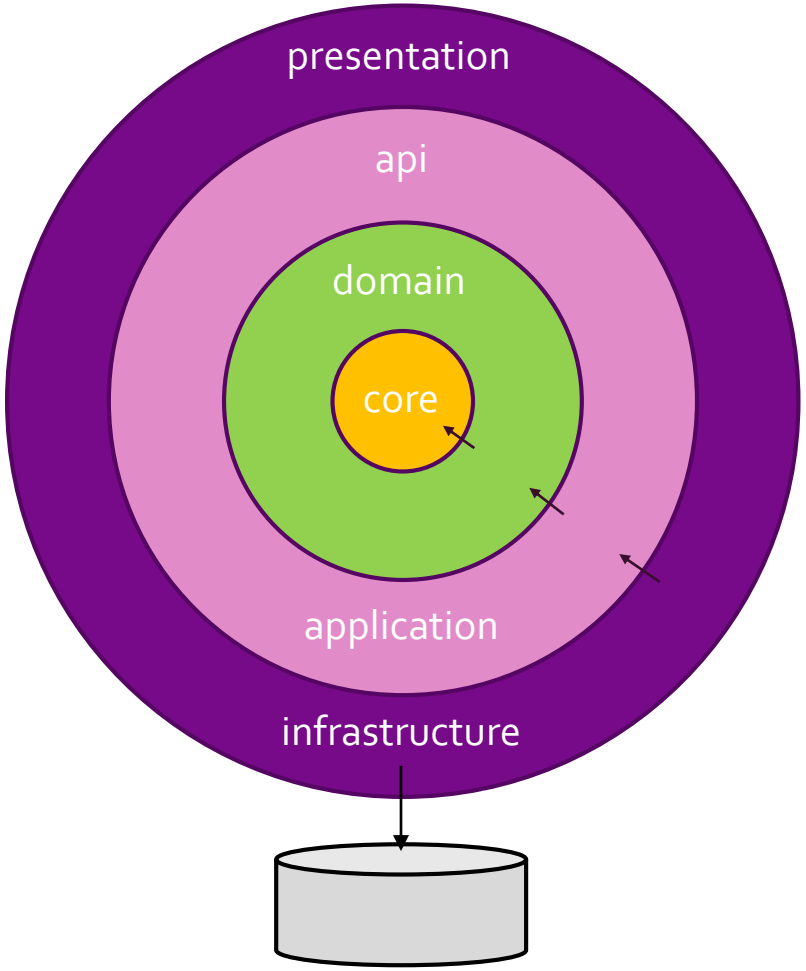
Onion architecture

- Often we see a layered model being used
  - Presentation coupled to data-model
  - Business logic dependent on data-access and infrastructure layer
  - Changes resonate throughout multiple layers



Onion architecture

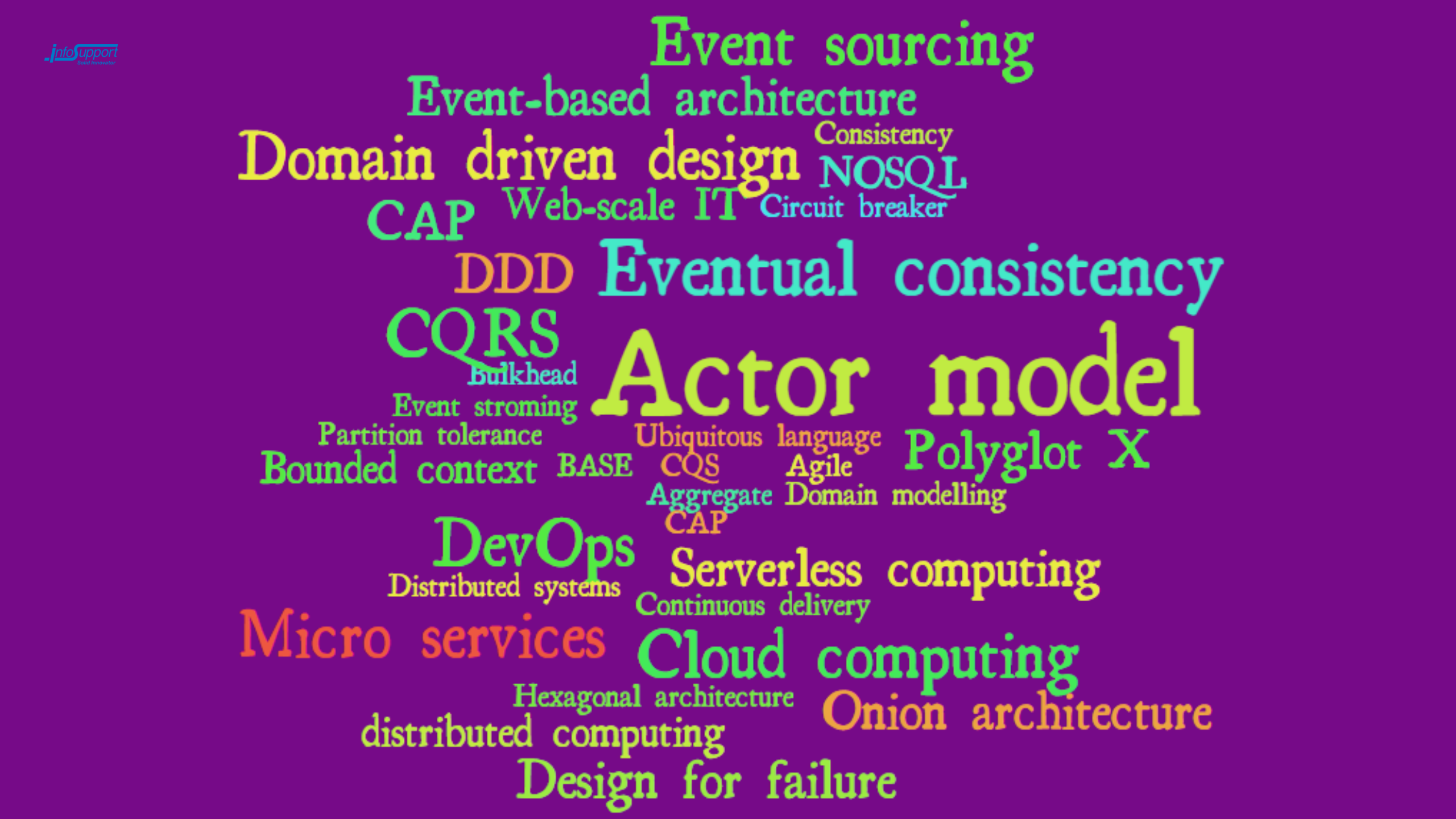
- A.k.a. “Hexagonal” architecture
  - Complexity and domain-logic lives in the center
  - Dependencies point “inwards” Domain-logic does not depend on infra / database



Onion architecture

- Everything is decoupled using interfaces
  - Interfaces defined in the domain layer
  - No tight coupling with concrete implementation
  - Use of a dependency injection framework possible

- Unit-testing of business logic is simpler because of the lack of dependencies (easy mocking / stubbing)


Event sourcing
Event-based architecture
Domain driven design
Consistency
NOSQL
CAP
Web-scale IT
Circuit breaker
DDD
Eventual consistency
CQRS
Actor model
Bulkhead
Event stroming
Partition tolerance
Ubiquitous language
Polyglot X
Bounded context
BASE
CQS
Agile
Aggregate
Domain modelling
CAP
DevOps
Serverless computing
Distributed systems
Continuous delivery
Micro services
Cloud computing
Hexagonal architecture
Onion architecture
distributed computing
Design for failure

Actor model

- Pattern for building highly scalable distributed systems
- Particularly well suited for handling large concurrent workloads
- Several implementations exist:
  - Erlang
  - Scala.Akka
  - Akka.NET
  - Azure Service Fabric
  - …

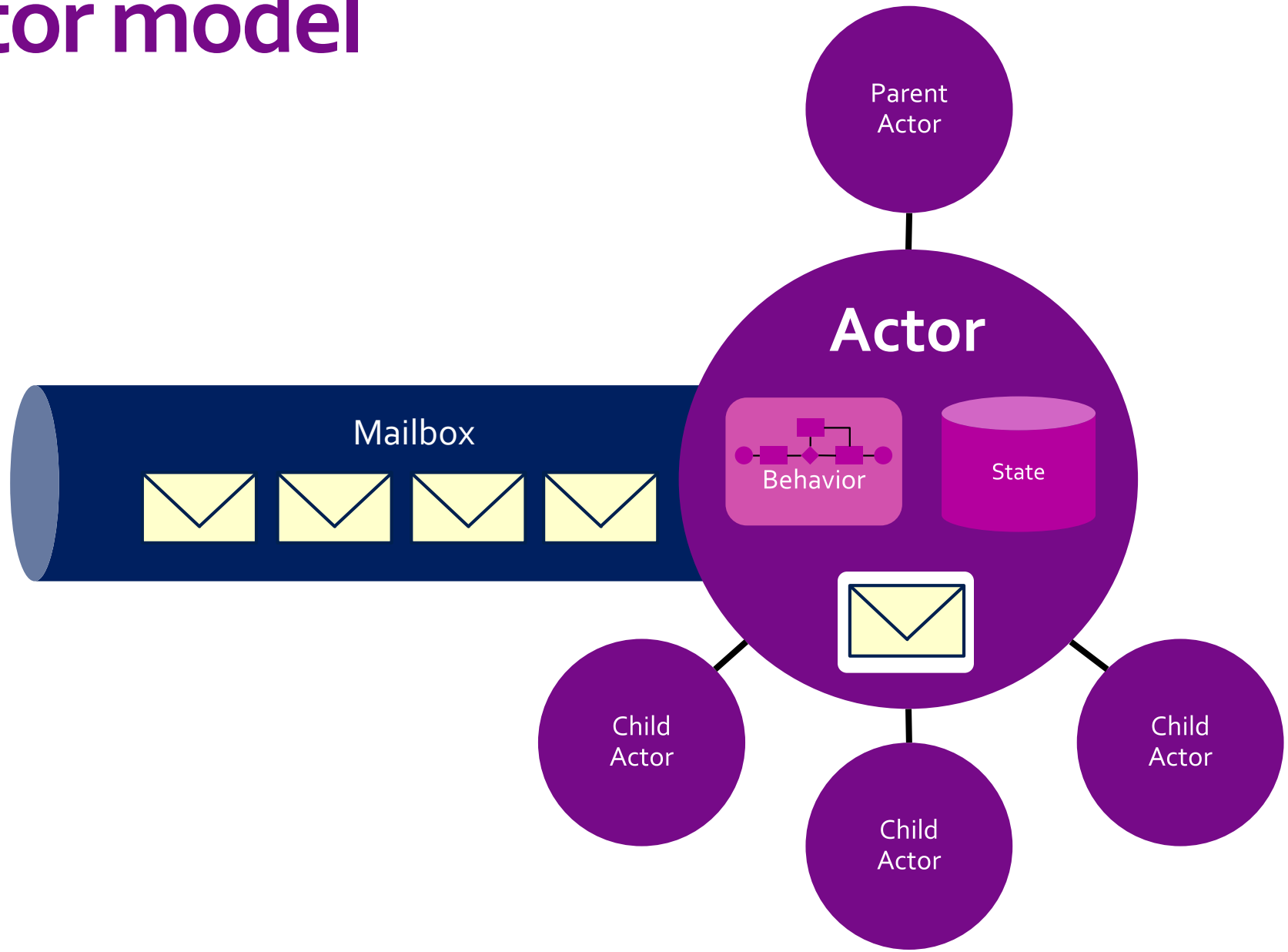
Actor model

- Communication with an actor is done solely using asynchronous messages
- Messages arrive in the mailbox (queue) of the actor
- An actor handles exactly 1 message at a time
  - No concurrency issues when updating state

Actor model

- An actor can handle a message in several ways
  - Execute certain (business-)logic
  - Update its own private state
  - Send messages to other actor(s)
  - Create child actors to offload work
  - Change its behavior
    - Like a finite state machine
    - Each state handles different message types

Actor model


Parent Actor
Actor
Mailbox
Behavior
State
Child Actor
Child Actor
Child Actor

Actor model - characteristics

- Actors are location-transparent
  - No difference in calling a local or a remote actor
  - Each actor has a unique address

- Actors are scalable
  - Scale-out by creating multiple actors
  - Clustering over multiple machines possible

Actor model

- Not every problem can be solved using an Actor Model approach

- Sometimes looking at your problem differently can help
  - Partitioning of the work-load

Thank you

Prajeesh Prathap

IT Architect

CONTACT

InfoSupport