Problemen

In Professional
Software Development


Concurrentie voorblijven
Bruikbaar / User friendly
Korte Time to Market
Performance
Build the product Fast
Privacy
24/7 beschikbaar
Productief Team


Veranderende wereld
Flexibel
“made for change”
Onderhoudbaar
Hoge kwaliteit
Goed design
High Cohesion
Low Coupling

Web-Scale Architecture kan helpen:

Korte
Time to Market

Flexibel
“made for change”

24/7
beschikbaar

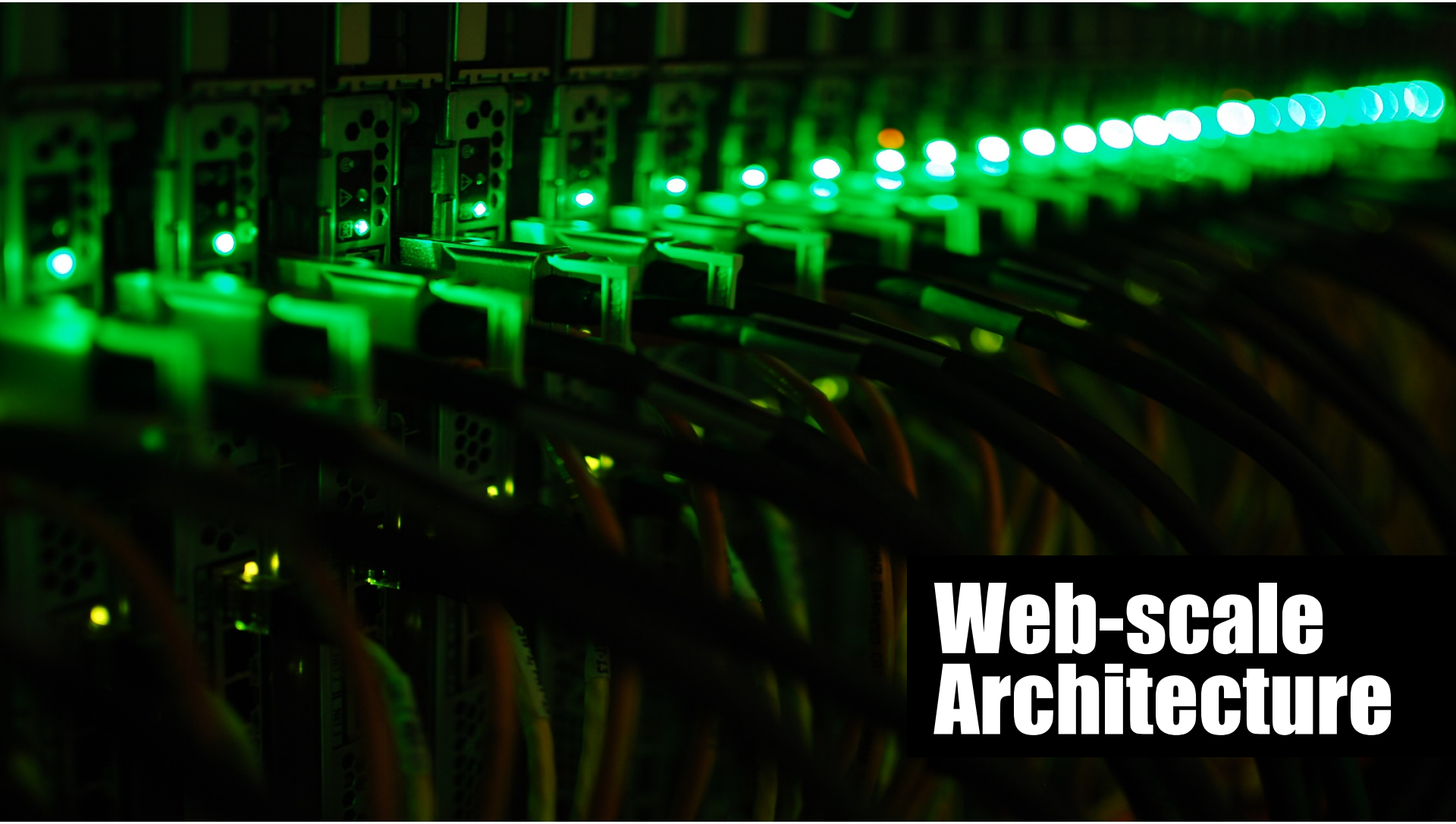
Complexe
software

IT Challenges today

- **Time to Market of new features is long**
- **Software has become too complex**
- **The “shop” is closed periodically**
- **SOA is a step in the right direction, but not enough**
- **Attracting and keeping skilled IT personnel is hard**

IT Challenges today

Web-scale architecture can help to overcome these challenges!


Web-scale
Architecture

Term “web-scale IT”

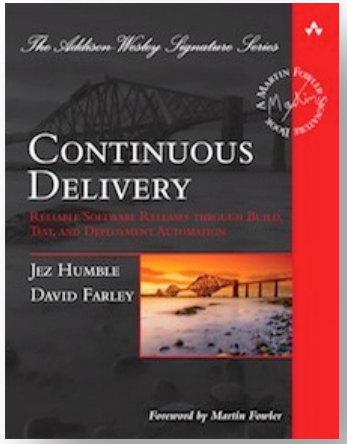
- Coined by Gartner

- Describes how large cloud services firms achieve
  - extreme levels of service delivery
  - Speed
  - agility

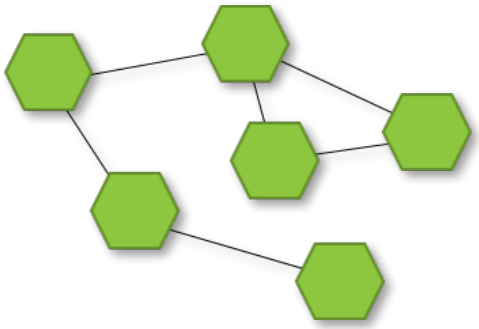
- Also applicable to enterprises and smaller companies

Achieving Web-scale IT

Agile Approach



Continuous Delivery

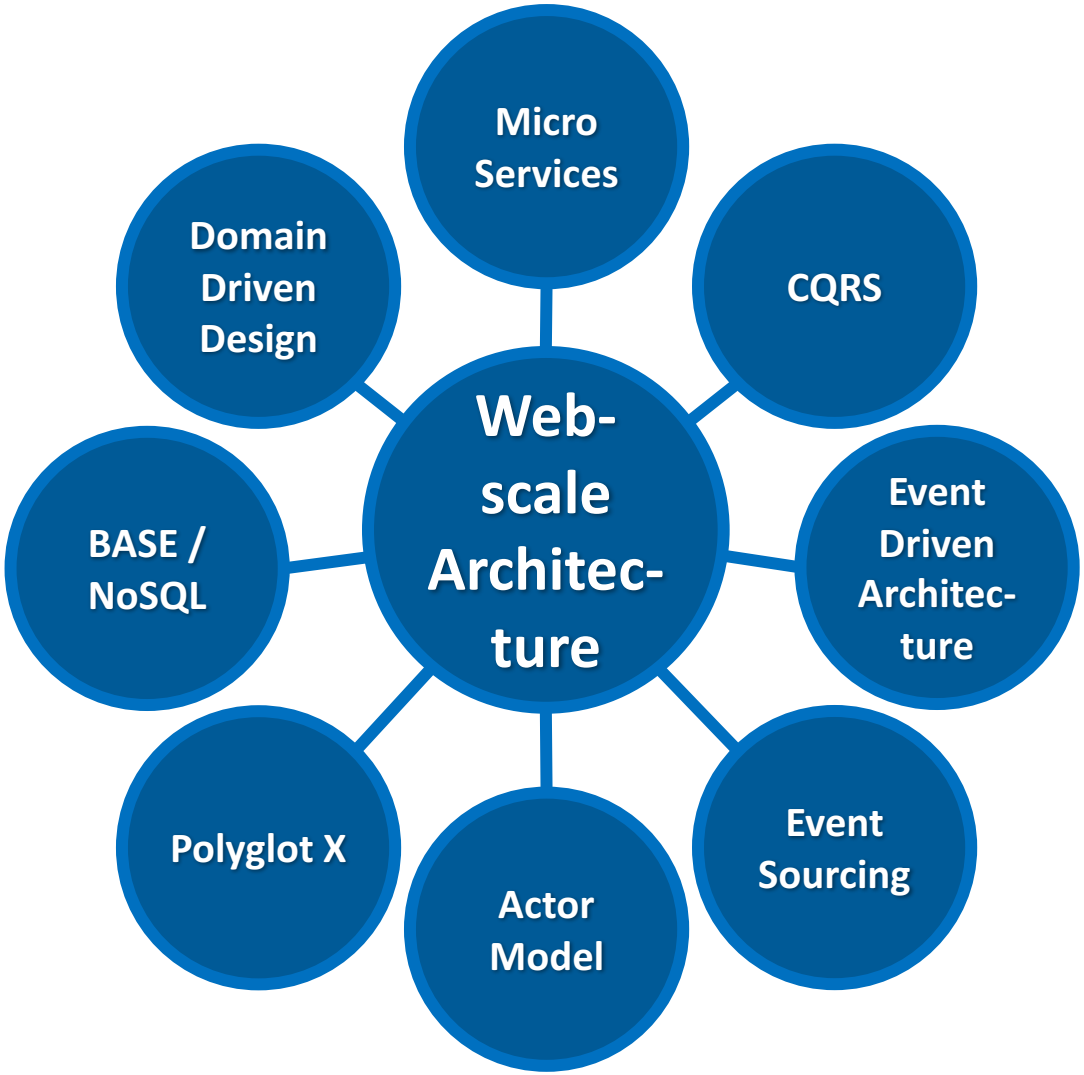


Modern Web-scale Architecture

What is Web-scale Architecture?

- A modern way of designing software that is flexible, scalable and available
- It is not a single top level architecture, but a collection of patterns and practices
- Info Support combined several patterns and practices into a Web-scale Architecture “toolbox”

Web-scale Architecture toolbox



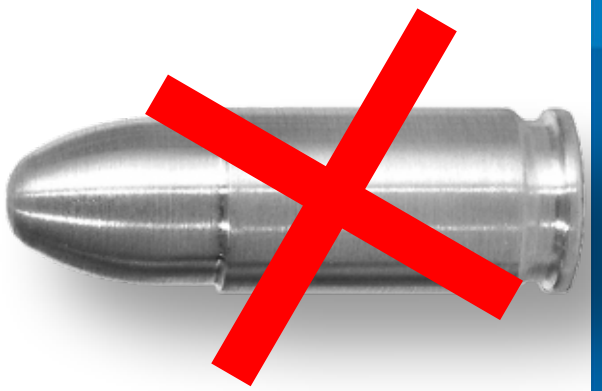
Are these all new concepts?

- No!
  - Actor Model : 1973 [Carl Hewitt]
  - CQS : 1988 [Book by Bertrand Meyer]]
  - DDD : 2003 [Book by Eric Evans]
  - CQRS : 2009 [Blog post by Greg Young]

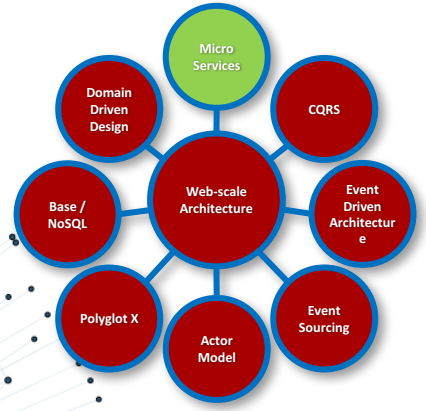
No silver bullet

KISS,
common sense
and craftsmanship

are the most
important tools
you own!



Micro services



Microservices introduction

- Architectural style in which systems are built as a collection of small services
- Each service is fully autonomous and specialized in a certain business capability
- Services are identified by analyzing business behavior as apposed to analyzing business data
- Services communicate (mostly asynchronously) using lightweight protocols
  - HTTP / REST

Microservices vs. SOA

- Microservices architecture is an evolution of SOA
- Most SOA principles are still valid
- SOA services handle an entire domain whereas Microservices handle a specific business capability
  - Focus on behavior in stead of data
- Data duplication is not evil in Microservices
  - No canonical schema

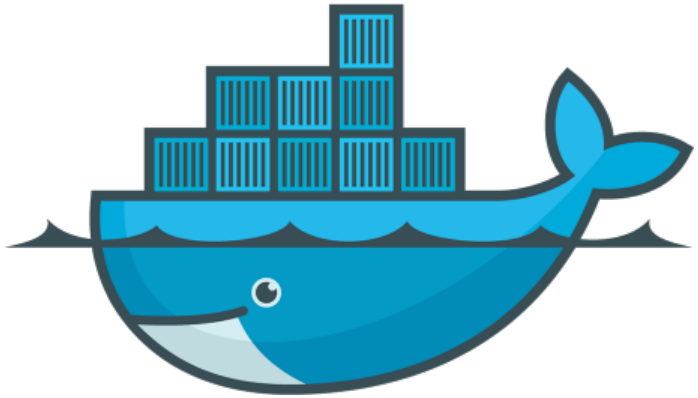
Microservices advantages

- The strict loosely coupled approach offers the ability to optimize per service
  - Polyglot X
- The strict loosely coupled approach enables teams to update each service independently
  - Continuous Delivery & Feature Teams
- Microservices' focus on a single business capability reduces the complexity of replacing a service with another implementation

Microservices and Containerization

- Container technologies as Docker are ideal for deploying Microservices

- Building the environment is part of the deployment pipeline



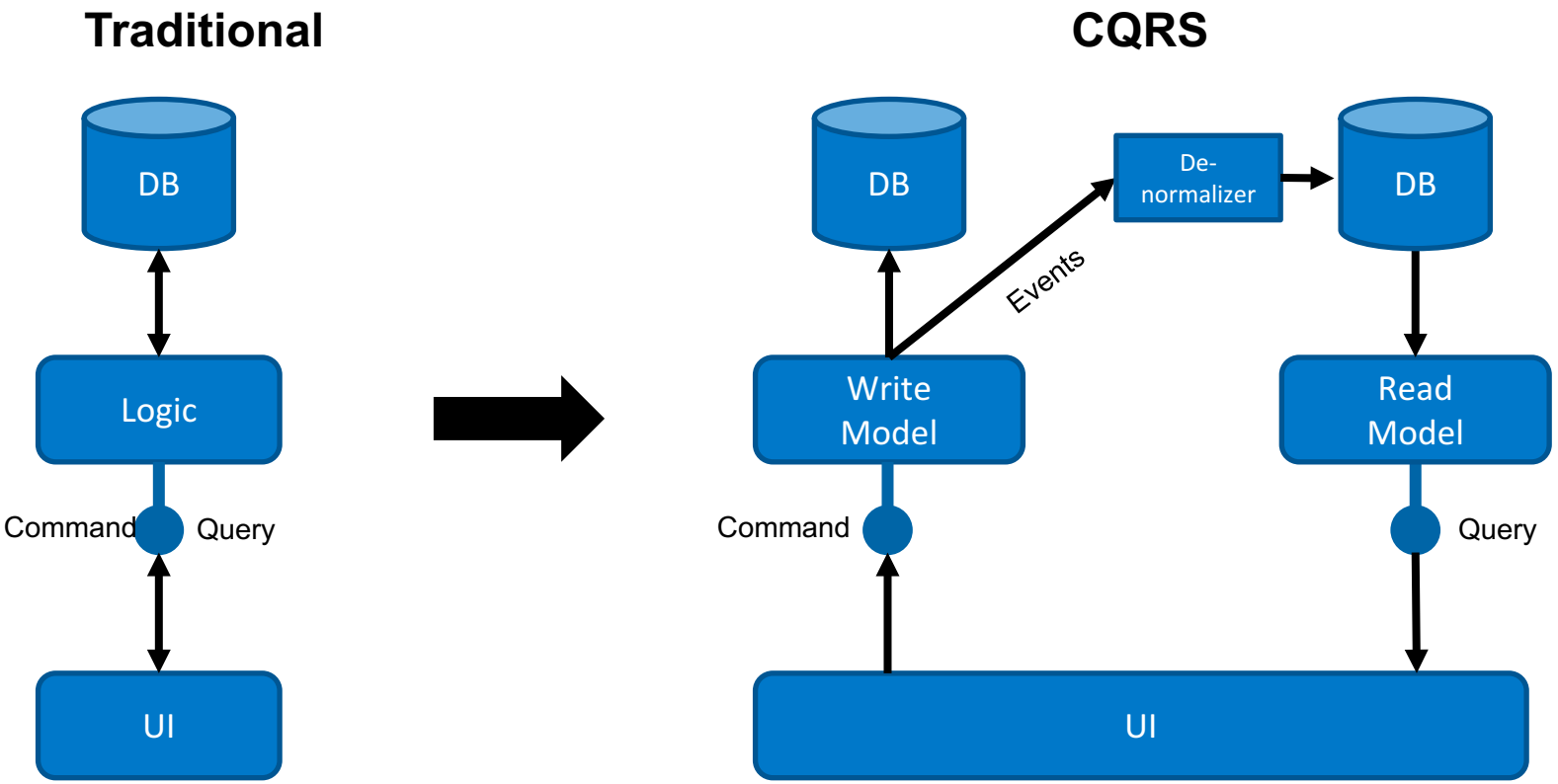
Microservices challenges

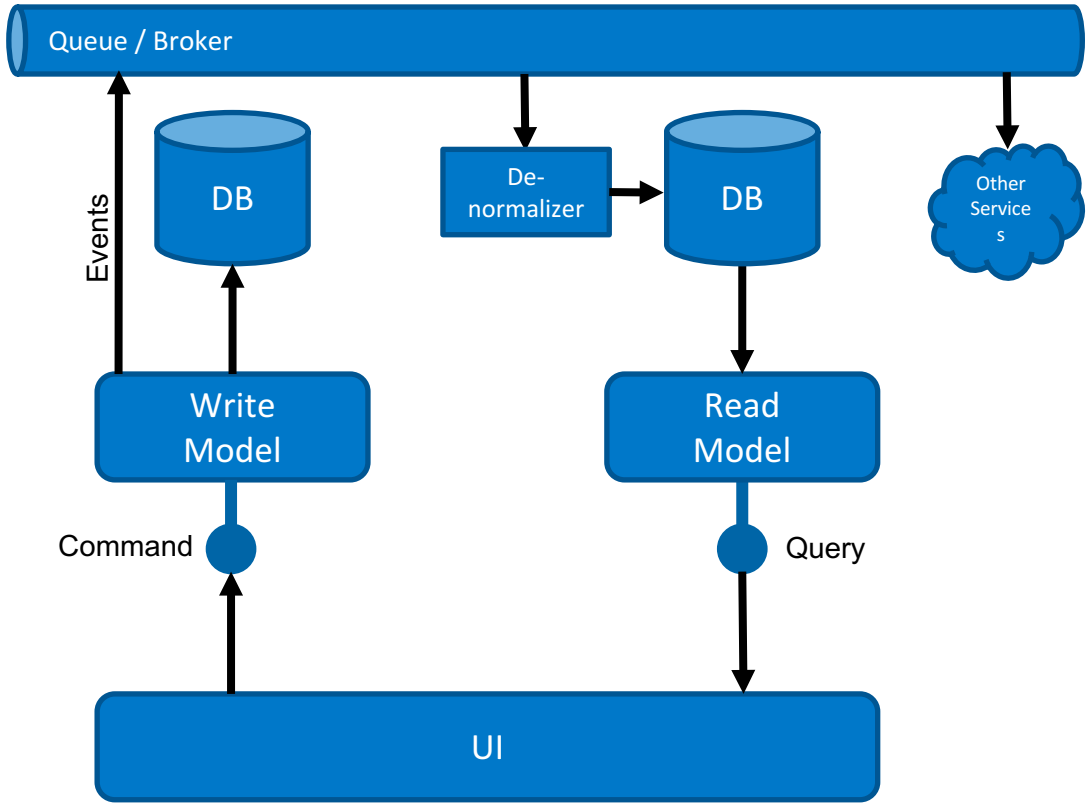
- Building a Microservices system is more complex than with a traditional approach
- Splitting a system up increases the effort needed for handling data consistency
  - Eventual Consistency
- Need to cope with eventual consistency
- Need to design for failure


Micro Services
Domain Driven Design
CQRS
Web-scale Architecture
BASE / NoSQL
Event Driven Architectur e
Polyglot X
Actor Model
Event Sourcing
CQRS

CQRS

- Application Architecture pattern: Command Query Responsibility Segregation
- Strict separation of updates (commands) and reads (queries) in a system
- Introduced by Greg Young as an evolution of CQS (by Bertrand Myer)


Traditional
DB
Logic
Command
Query
UI
CQRS
DB
De-
normalizer
DB
Events
Write
Model
Read
Model
Command
Query
UI


Queue / Broker
Events
DB
De-
normalizer
DB
Other
Service
s
Write
Model
Read
Model
Command
Query
UI

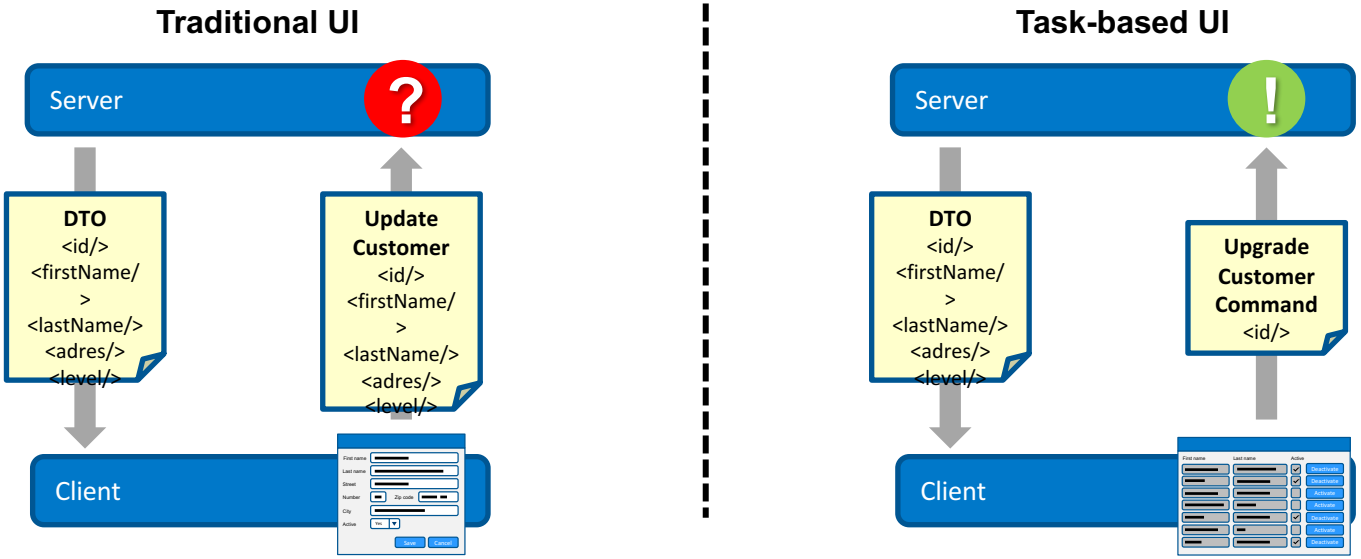
CQRS - Commands and Events

- Commands are the things that need to be executed
- Events are the things that have occurred in a certain point in time
- Events can be used internally within a service or published to the outside world
  - Event Driven Architecture
- CQRS often implies Eventual Consistency

CQRS - Advantages

- Scale the update and query parts independently
- Improves performance and can decrease coupling between systems
- Write-model and read-model (can) differ
- Multiple read-models possible
- Using commands can promote a more task-based UI approach

CQRS - Task-based UI



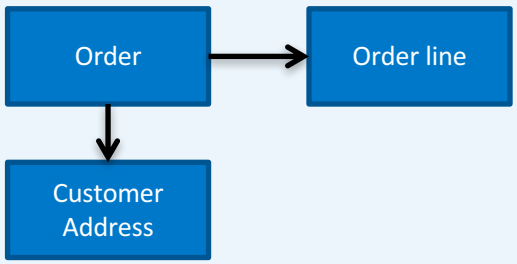
Info Support | Solid Innovator


Micro Services
Domain Driven Design
CQRS
Web-scale Architecture
BASE / NoSQL
Event Driven Architecture
Polyglot X
Actor Model
Event Sourcing
Event Sourcing

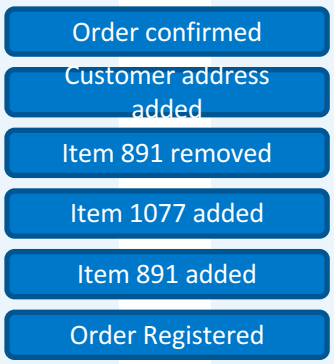
Event Sourcing

- **Event-sourcing is an alternative way of persisting the state of your domain-objects**

Not in a normalized RDBMS …

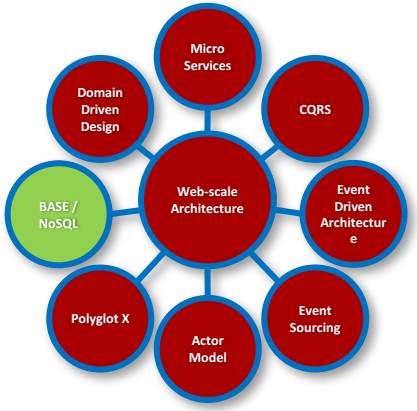


… but as a list of events that occured in time



Event Sourcing

- **Time becomes a first class concept in your software which is great for historical analysis**
- **Events are immutable and append only so super fast**
- **Event Sourcing is primarily used for rebuilding state when executing commands**
  - Queries are executed on view-model(s)
- **Ability to rebuild state from events is very useful for troubleshooting and (unit-)testing**



BASE - Eventual Consistency

Eventual Consistency

Traditional systems offer ACID characteristics

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Modern systems are more focused on BASE characteristics

- **B**asic **A**vailability
- **S**oft-state
- **E**ventual consistency

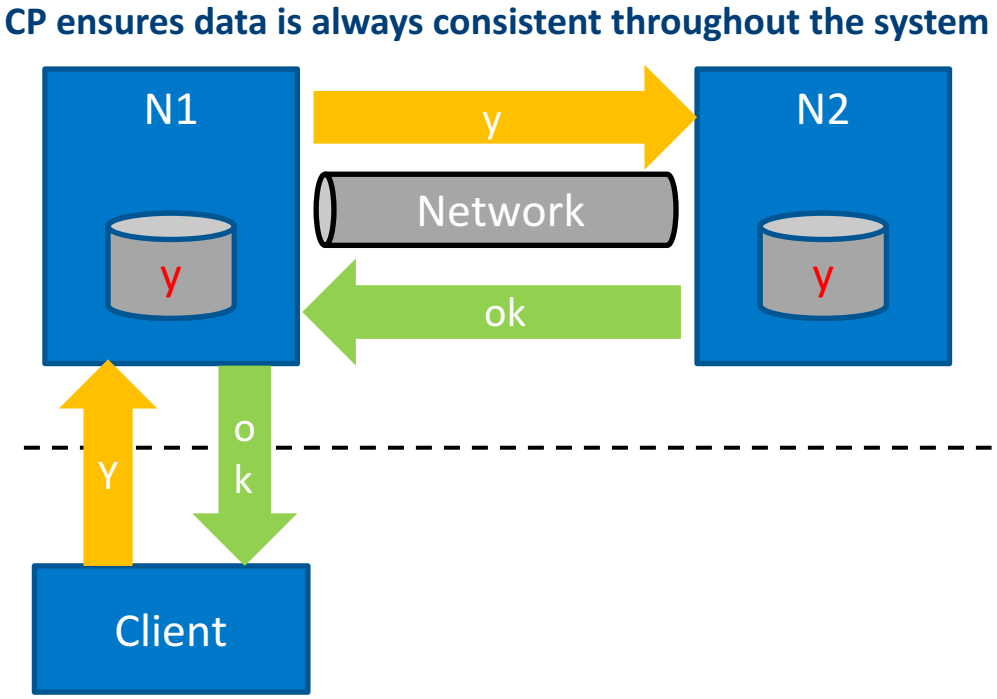
Eventual Consistency

- **EC states that when data is changed, it can take some time for all systems to process this change and contain the same data**
- **Must be part of your software architecture and design**
- **Events are a good way to implement eventual consistency**
- **EC is often not easily accepted**
  - Yet, in the “real” world almost every process is EC
- **EC can save you a lot of complexity and trouble (and $)**

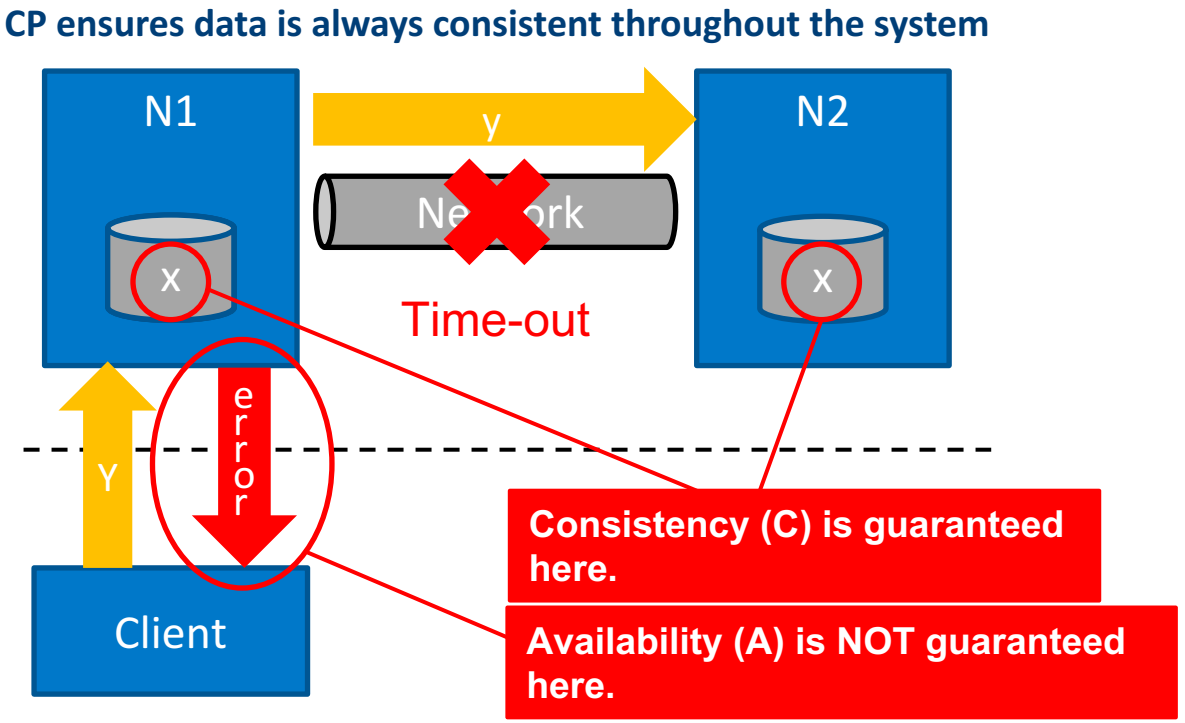
Eventual Consistency - Cap Theorem

- **For distributed systems the CAP theorem applies:**
  - **C**onsistency, **A**vailability and **P**artition Tolerance
  - **In a distributed system its not possible to adhere to C,A & P at the same time**
    - Networks are not reliable by nature
    - We MUST be “partition tolerant”
  - **We need to choose between *Consistent* and *Partition Tolerant* (CP) or *Available* and *Partition Tolerant* (AP)**

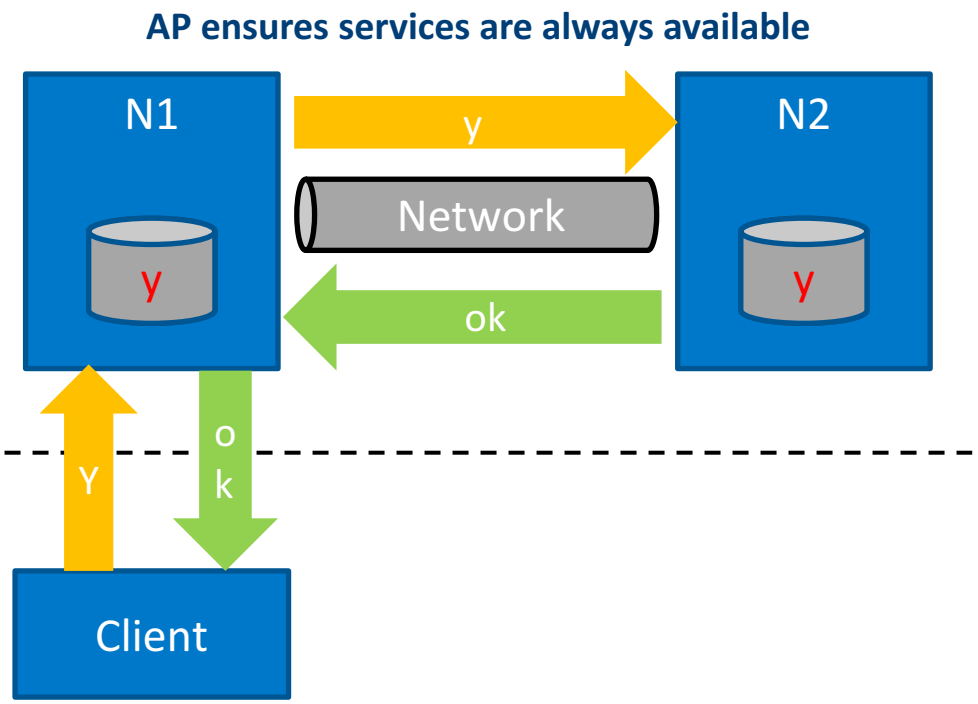
Eventual Consistency - CAP theorem



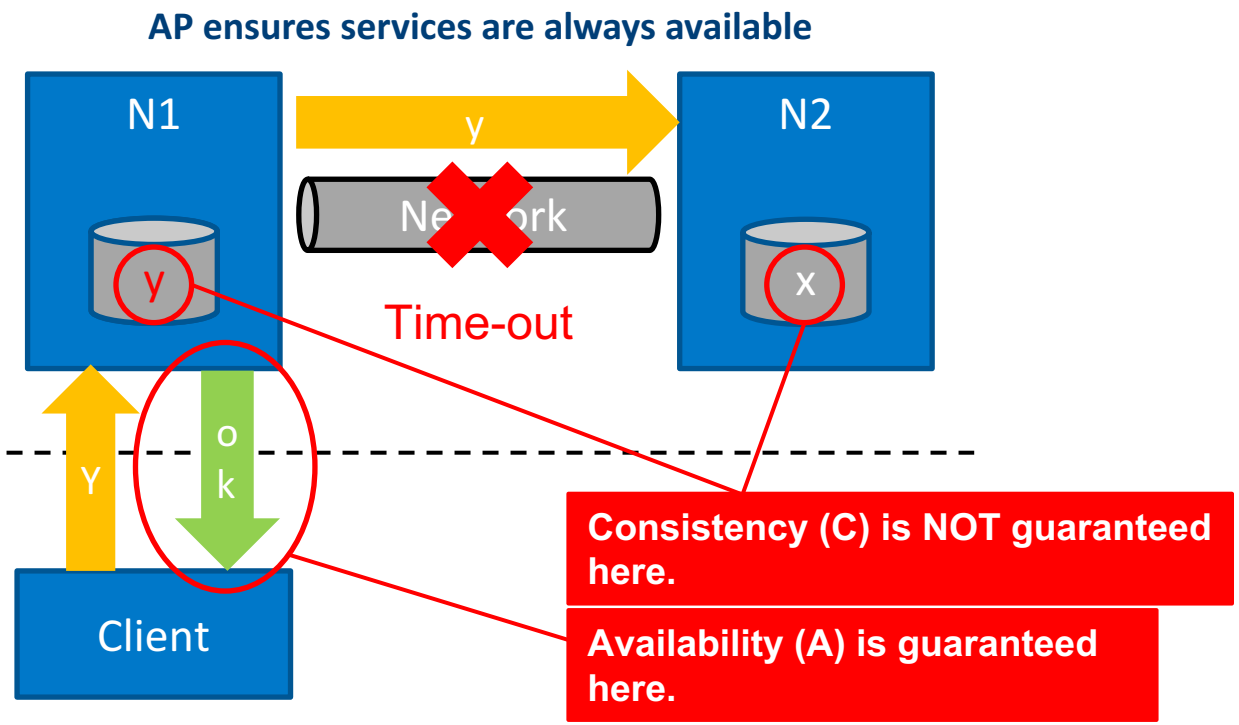
Eventual Consistency - CAP theorem



Eventual Consistency - CAP theorem



Eventual Consistency - CAP theorem



Design for failure

Design for Failure

- **Errors WILL occur, so make sure you can recover fast!!**

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

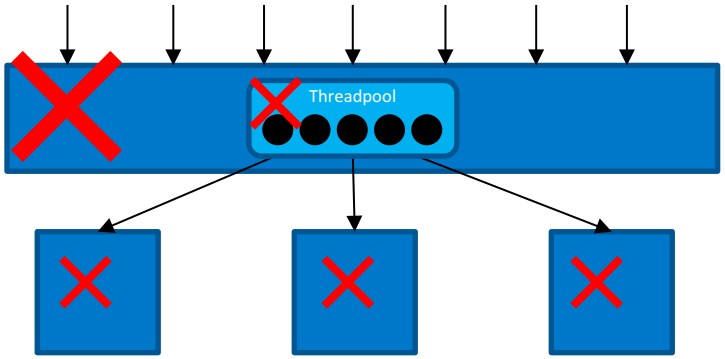
No influence
(remember the “8 falacies”)

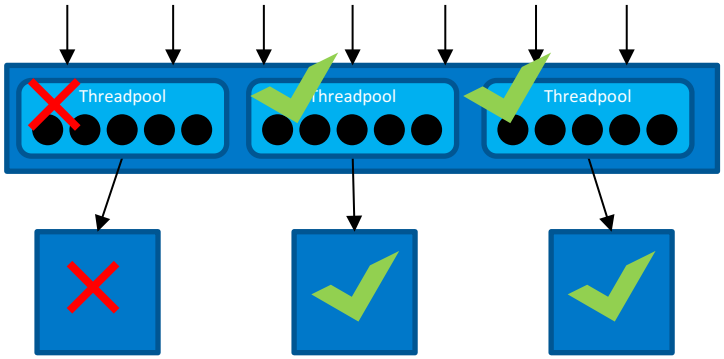
Lots of influence
(we write the code)

MTTF: Mean Time To Failure
MTTR: Mean Time To Recovery

Design for Failure

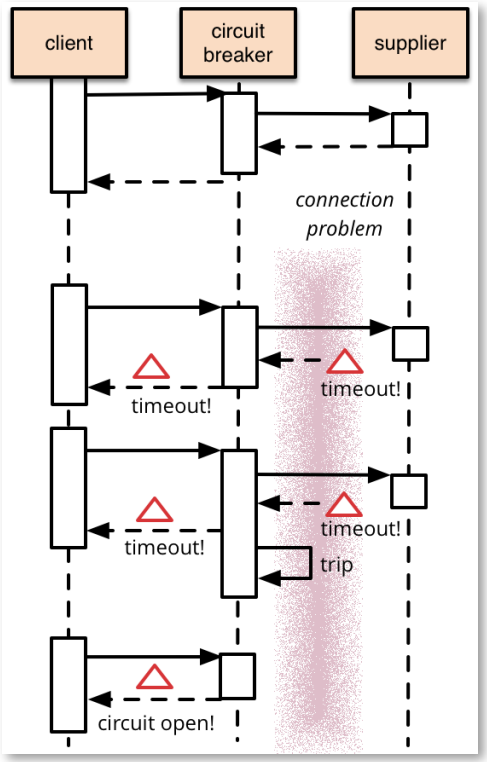
- **Introduce fault domains in your system**
  - Bulkhead pattern (nautical term)
  - Make sure if something breaks, the system only breaks partially

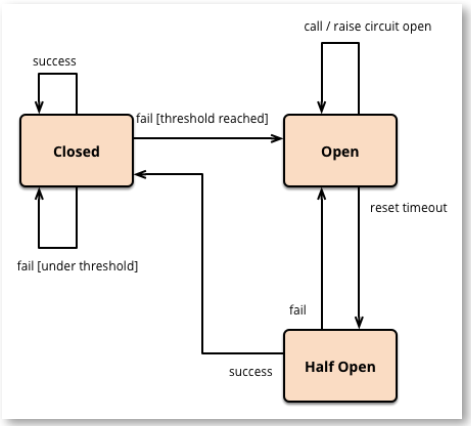




Design for Failure

- **"Fail fast"**
  - Circuit-breaker pattern
  - Make sure you don't keep waiting on time-outs from an unhealthy service
  - This bogs down performance and starves thread pools





http://martinfowler.com/bliki/CircuitBreaker.html