event sourcery

keep the cost of change lower for longer

Sebastian von Conrad
James Ross

shameless self-promotion

Sebastian von Conrad - @vonconrad
James Ross - @jimmyjazz68

some important questions

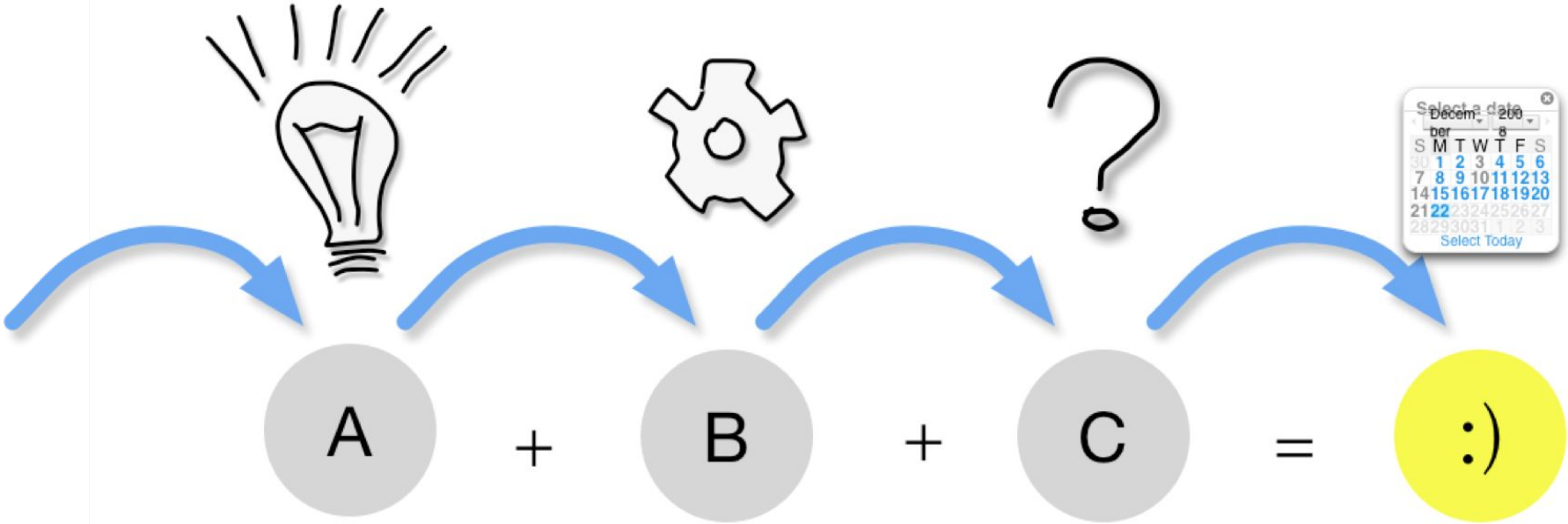
some important questions

how important is software engineering to our agility?

some important questions

hint: *very*

some important questions



some important questions

how agile is our software engineering?

some important questions

hint: *not very*

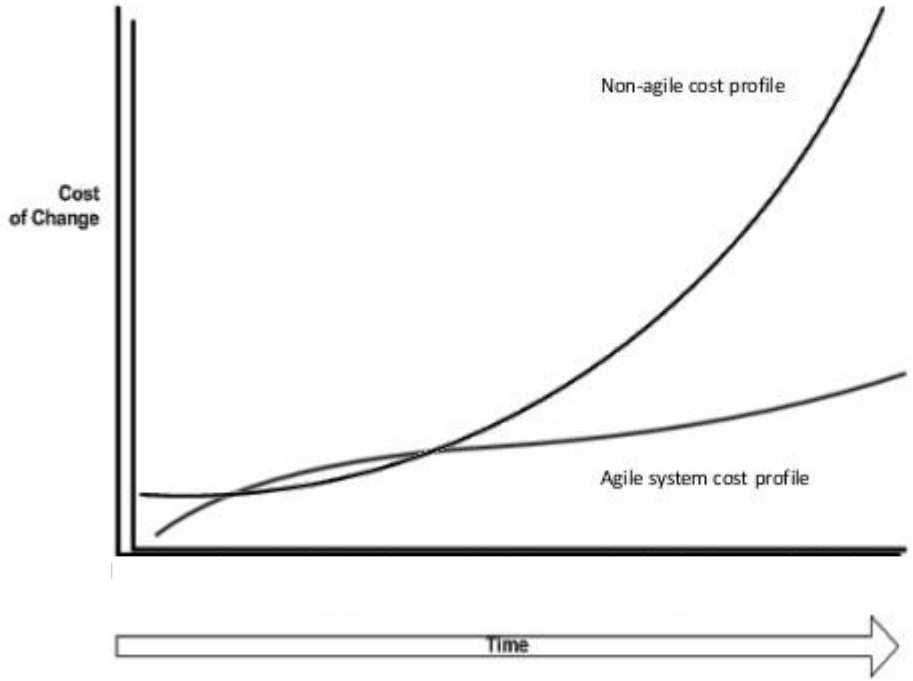
some important questions

an envato example

some important questions

we need to do better!

some important questions



some important assumptions

some important assumptions

business outcomes over technical outcomes

some important assumptions

functional slices over architectural layers

some important assumptions

ability to respond
over
ability to predict

some important assumptions

the long term
and
the short term

polarities to be managed
not options to be chosen

there’s one thing you can rely on *not* to change...

some important assumptions

the past :)

event sourcing 101

event sourcing 101

an event is a business fact that happened at a particular time

i.e. it represents something that changed in the past

examples from a calendar domain

- appointment scheduled
- appointment rescheduled
- appointment location moved
- appointment cancelled
- invitation extended
- invitation notification sent
- invitation accepted
- invitation declined

examples from the Envato domain

- item version submitted
- item version approved
- item added to cart
- item removed from cart
- item licence purchased
- item support purchased
- withdrawal request submitted
- withdrawal completed

event sourcing 101

event sourcing is using an append-only series of immutable events as the source of truth

i.e. put the only thing you can rely on at the core

...and deriving *everything else* from the events

...by *sourcing* current state by replaying events

event sourcing 101

...so *everything else* is completely disposable

being able to start over quickly when things change = agility

event sourcing 101

language agnostic

event sourcing 101

paradigm agnostic

functional, object oriented, SQL, NoSQL, whatever

...and it's the *only* approach used in financial domains

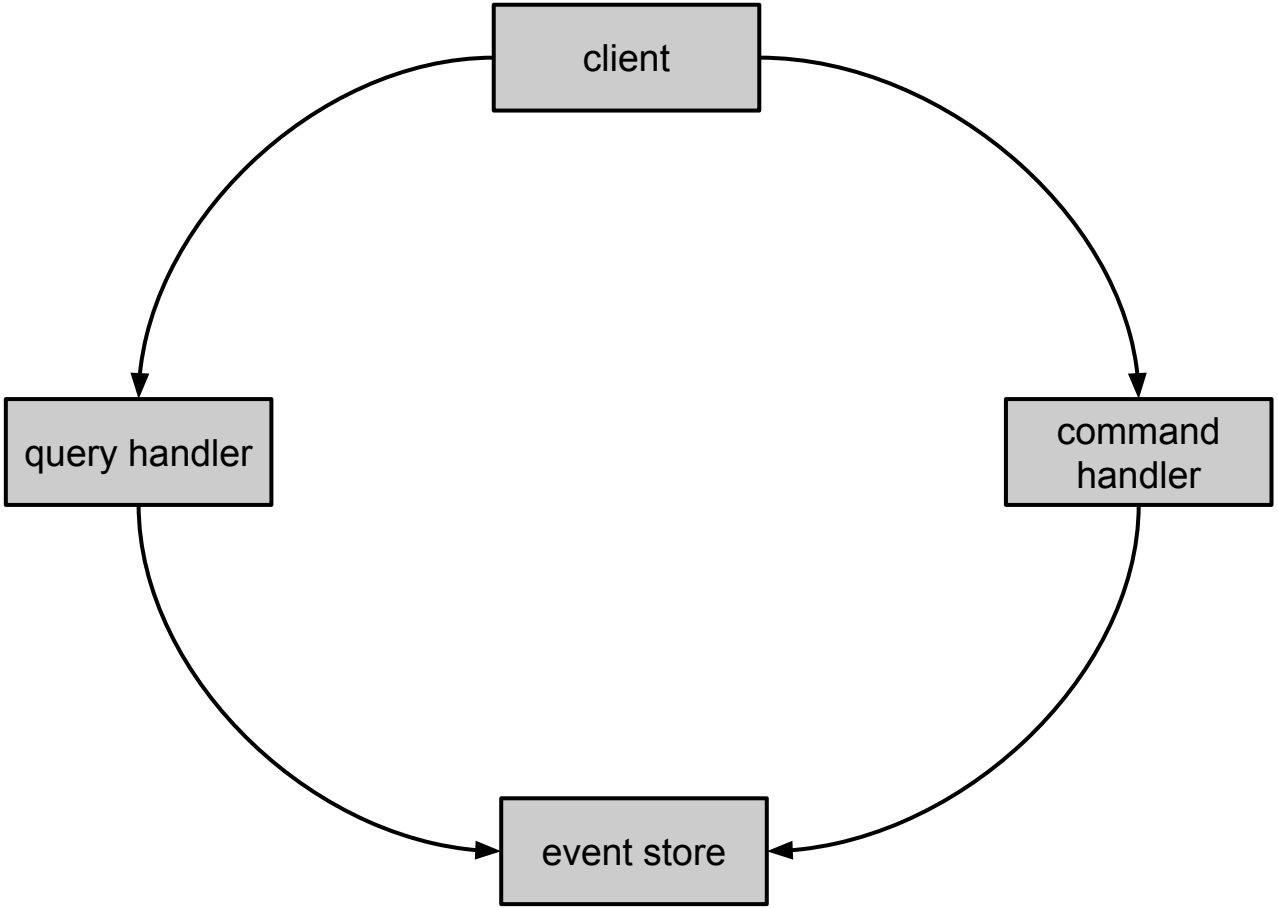
CQRS 101

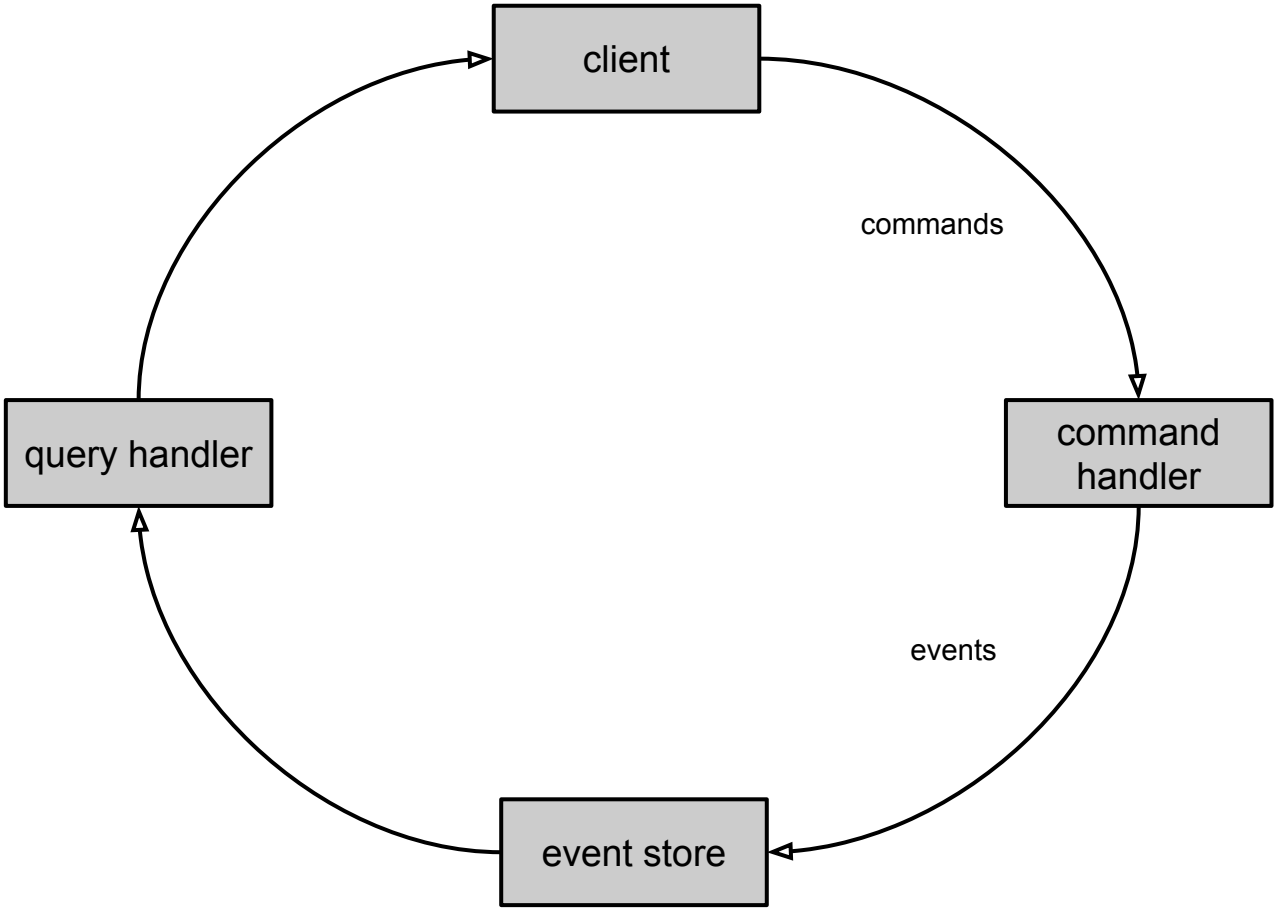
CQS: methods can read (queries) or write (commands) but not both

CQRS 101

CQRS: objects have only commands or queries but not both

we take it one step further and separate read/write subsystems


client
query handler
command handler
event store

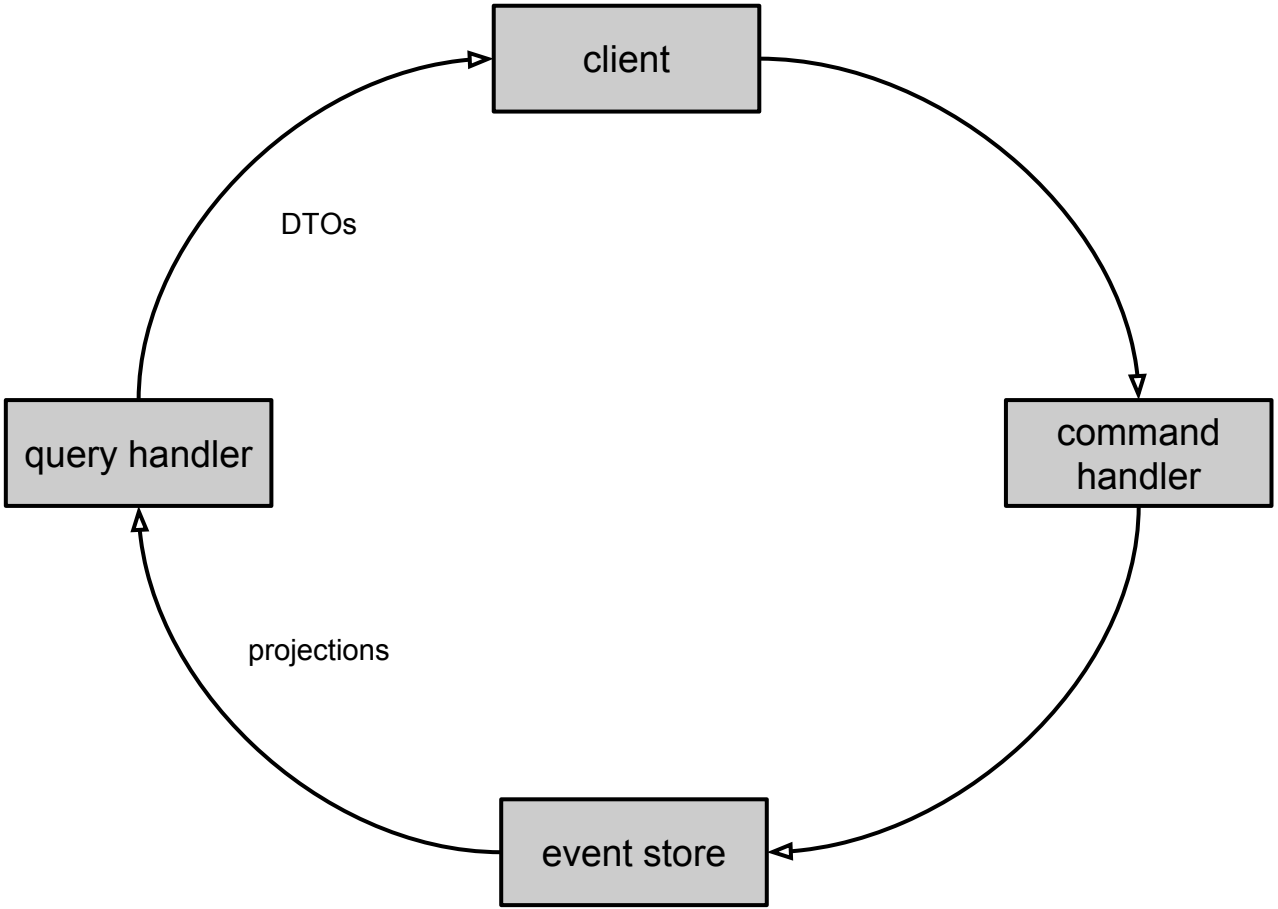

client
commands
command handler
events
event store
query handler

CQRS 101

commands represent user intent

commands can be rejected

events are produced when commands are accepted


client
DTOs
query handler
command handler
projections
event store

projectors process events in order

projectors maintain denormalised projections

projectors and projections are 1:1

one projection per screen/endpoint

CQRS 101

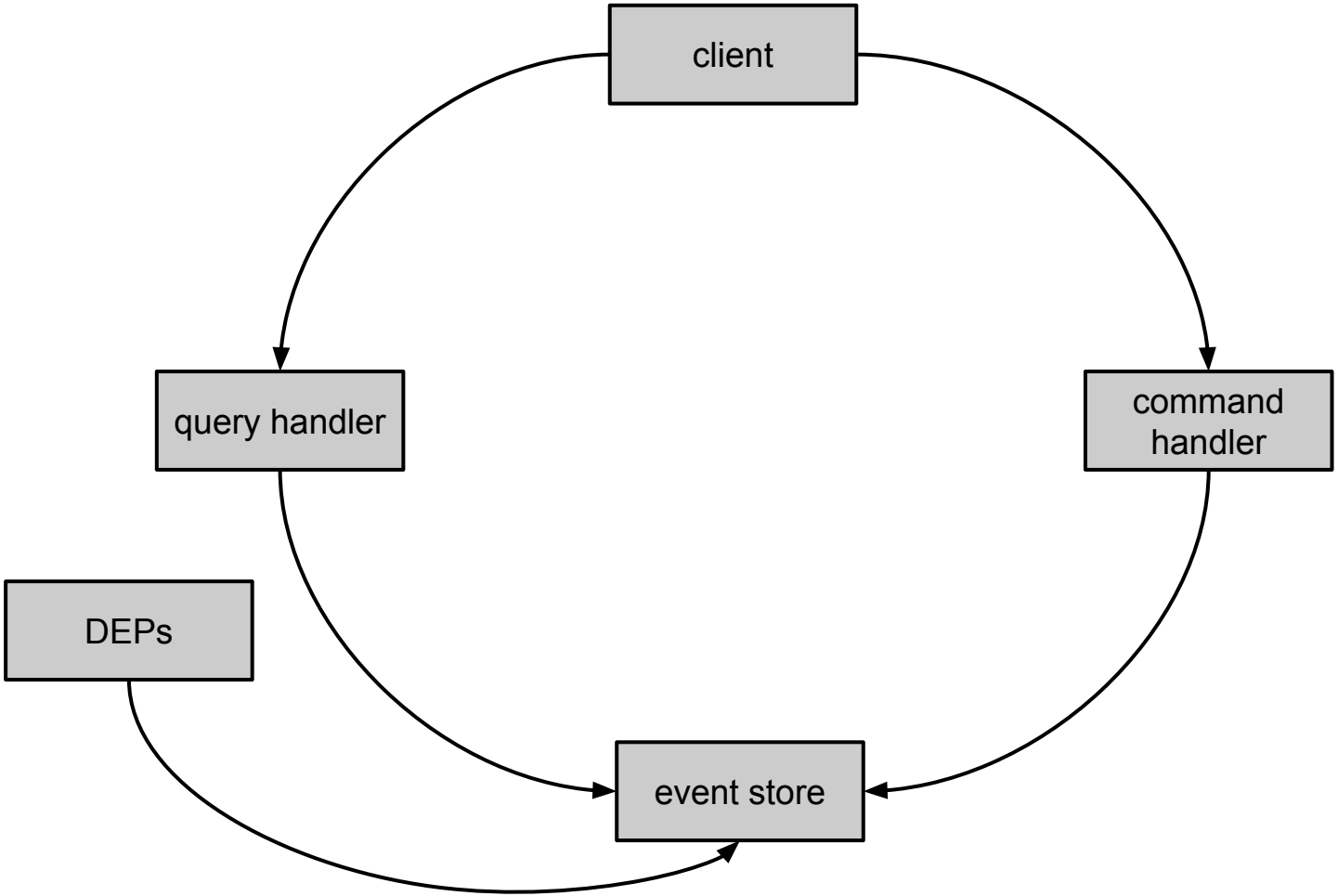
query handlers query projections

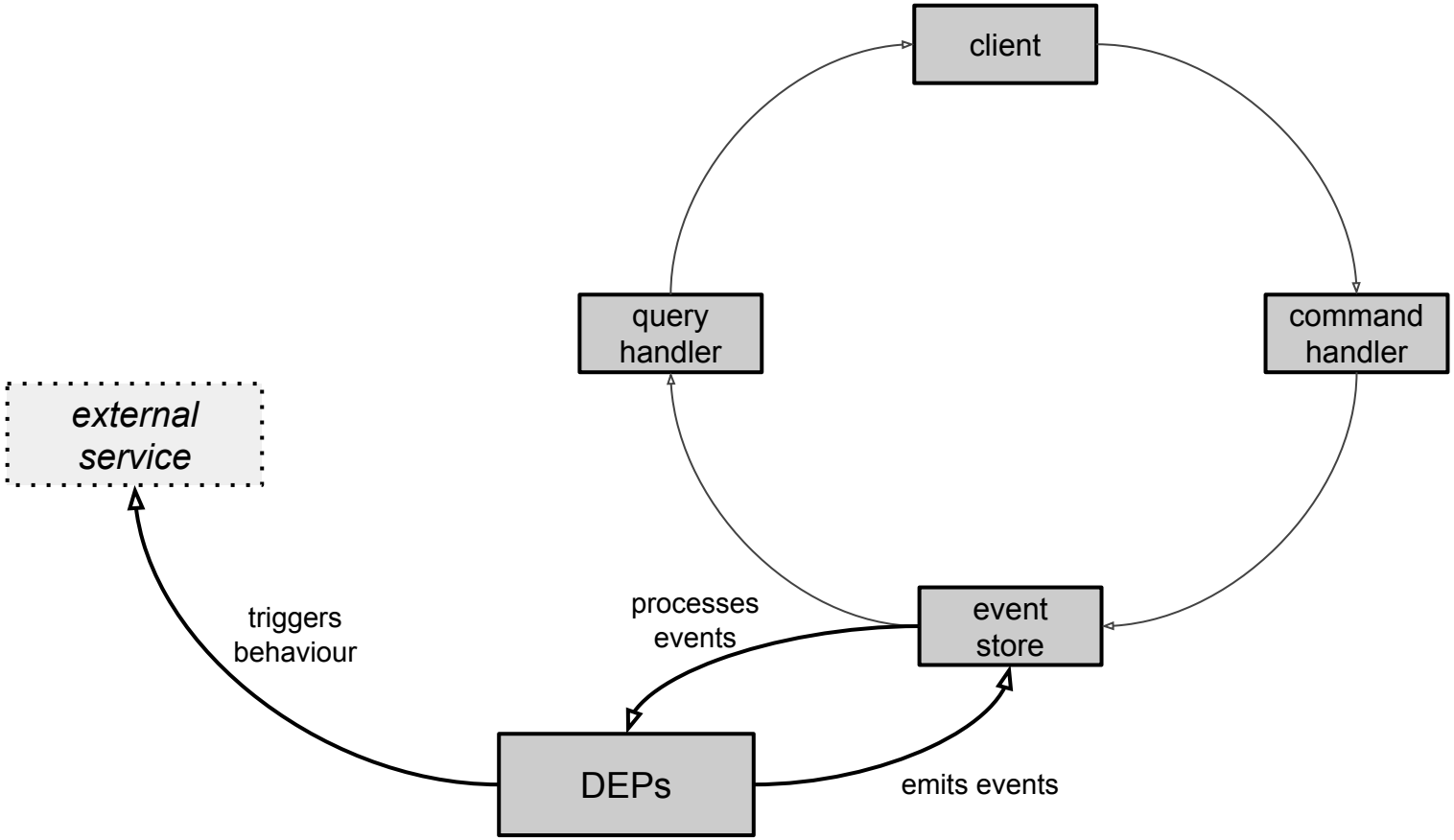
CQRS 101

query handlers return DTOs

CQRS 101

downstream event processors (DEPs)


client
query handler
command handler
DEPs
event store


client
query handler
command handler
external service
triggers behaviour
processes events
event store
DEPs
emits events

DEPs process events like projectors

DEPs can react by emitting events back to the event stream

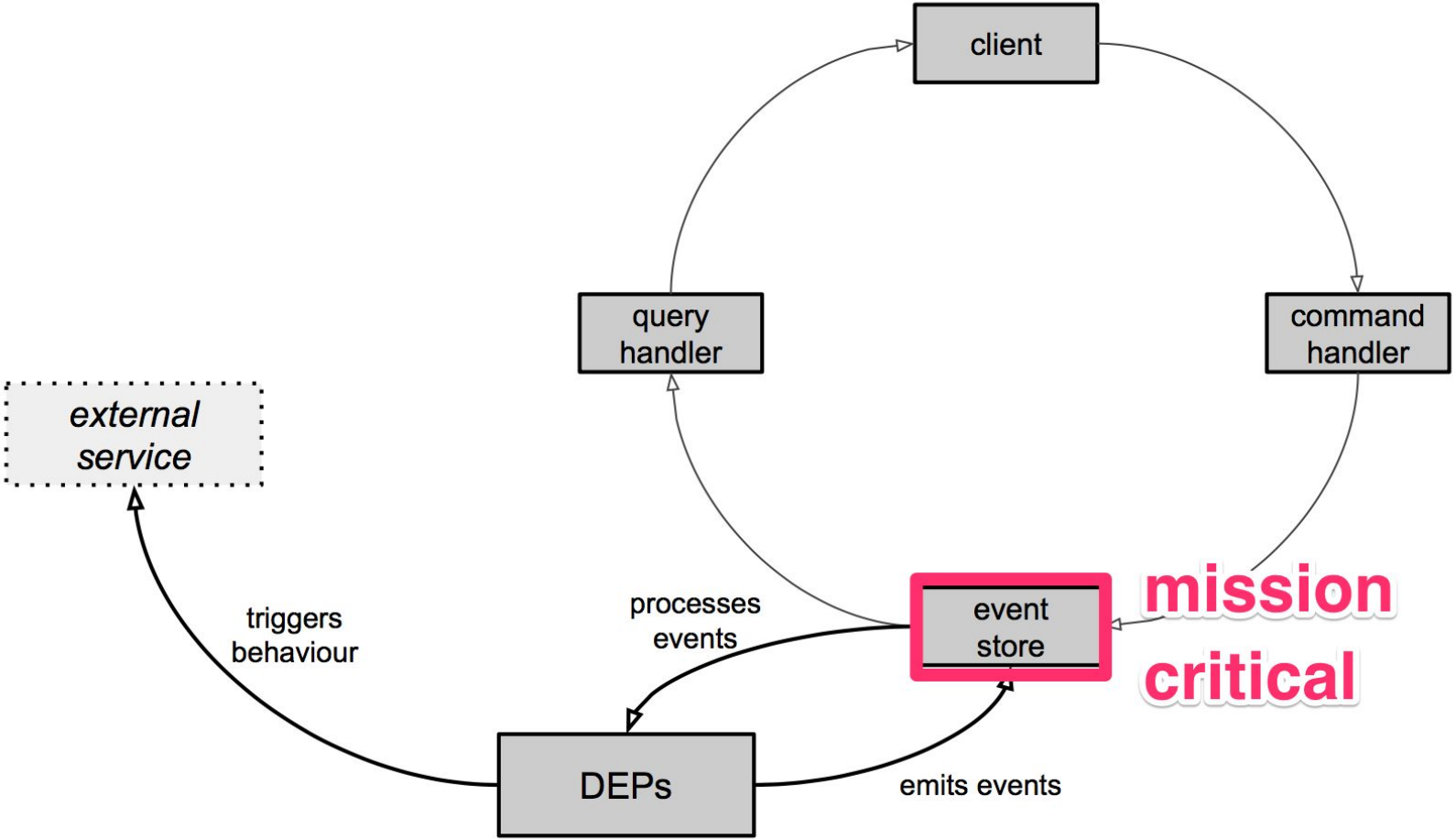
DEPs can react by triggering external behaviour

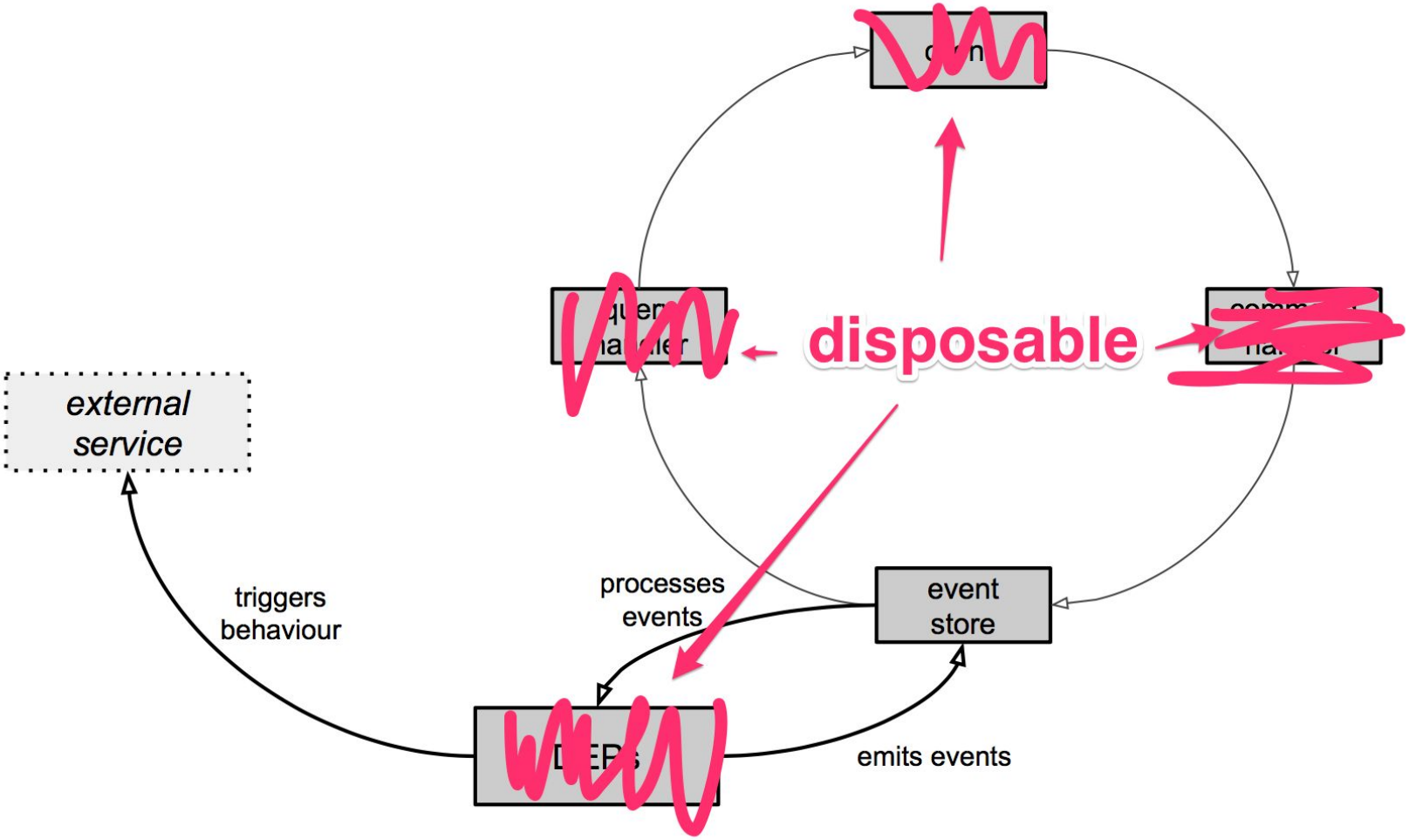
CQRS 101

DEPs encourage clean separation of concerns

responding to change

...and keeping the cost of change *lower for longer* with CQRS and event sourcing


client
query handler
command handler
external service
triggers behaviour
processes events
event store
mission critical
DEPs
emits events


disposable
external service
triggers behaviour
processes events
event store
emits events

separate recording of what happened from interpreting it

…so we can more easily re-interpret it

…even have multiple different interpretations at the same time

responding to change

we get a free time machine

responding to change

code organised by functionality in vertical slices

the core is far more stable than the edges

…and our edge systems are easier to build

responding to change

…and to get rid of

no destructive schema migrations

no schema migrations full stop

separation of concerns limits the blast radius of changes

responding to change

...and creates logical seams for microservices

responding to change

reading and writing scales independently

...allowing you to select whatever technology best suits a given subsystem

responding to change

all of this...

...reduces fear and enables more rapid change

...keeping the cost of change lower for longer

#nosilverbullets

#nosilverbullets

not appropriate for every problem

#nosilverbullets

a *serious* paradigm shift for developers

#nosilverbullets

a complex arrangement of simple things

#nosilverbullets

...can make monitoring and tracing interesting :)

#nosilverbullets

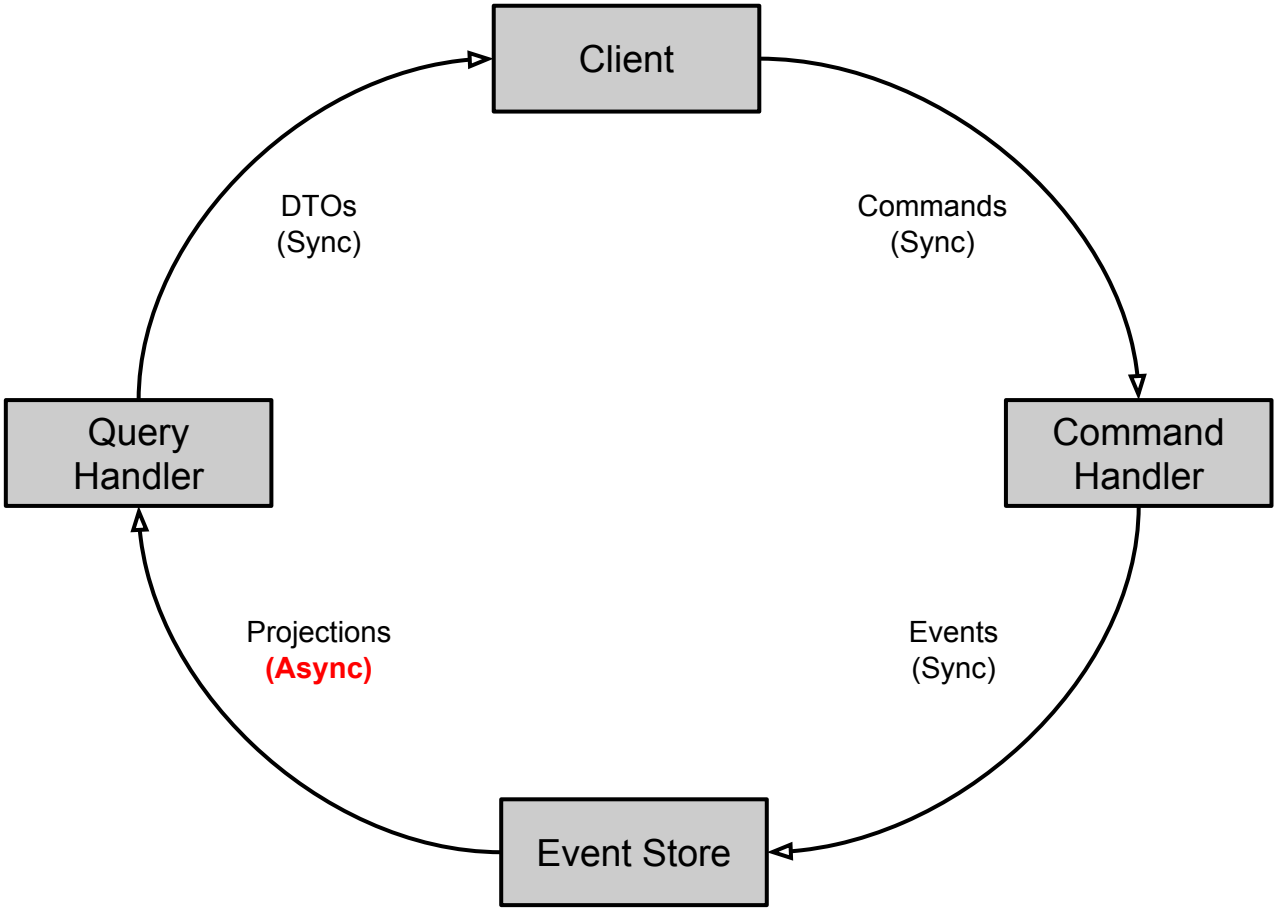
and then there’s that other thing

#nosilverbullets

that thing we’re not supposed to talk about

#nosilverbullets

eventual consistency


Client
Commands
(Sync)
Command
Handler
Events
(Sync)
Event Store
Projections
(Async)
Query
Handler
DTOs
(Sync)

#nosilverbullets

the world is eventually consistent

#nosilverbullets

and so is your current system :)

#nosilverbullets

is a nanosecond delay okay?

what about a month?

#nosilverbullets

risk is always a function of time

#nosilverbullets

...and accepting risk is a business decision not a technical one

Q & (maybe) A