The 7 quests of resilient software design

A guide for the adventurous software engineer

Uwe Friedrichsen (codecentric AG) – GOTO Berlin – Berlin, 2. November 2018

Uwe Friedrichsen

IT traveller.

Dot Connector.

Cartographer of uncharted territory.

Keeper of timeless wisdom.

CTO and Fellow at codecentric.

https://www.slideshare.net/ufried

https://medium.com/@ufried

@ufried

You want to do resilient software design …

... and you expect everything to be like this

But somehow it feels more like that ...

... or even that

What the **** went wrong?

The road to resilience is a twisted one

“7 quests you must complete!”

Quest #1

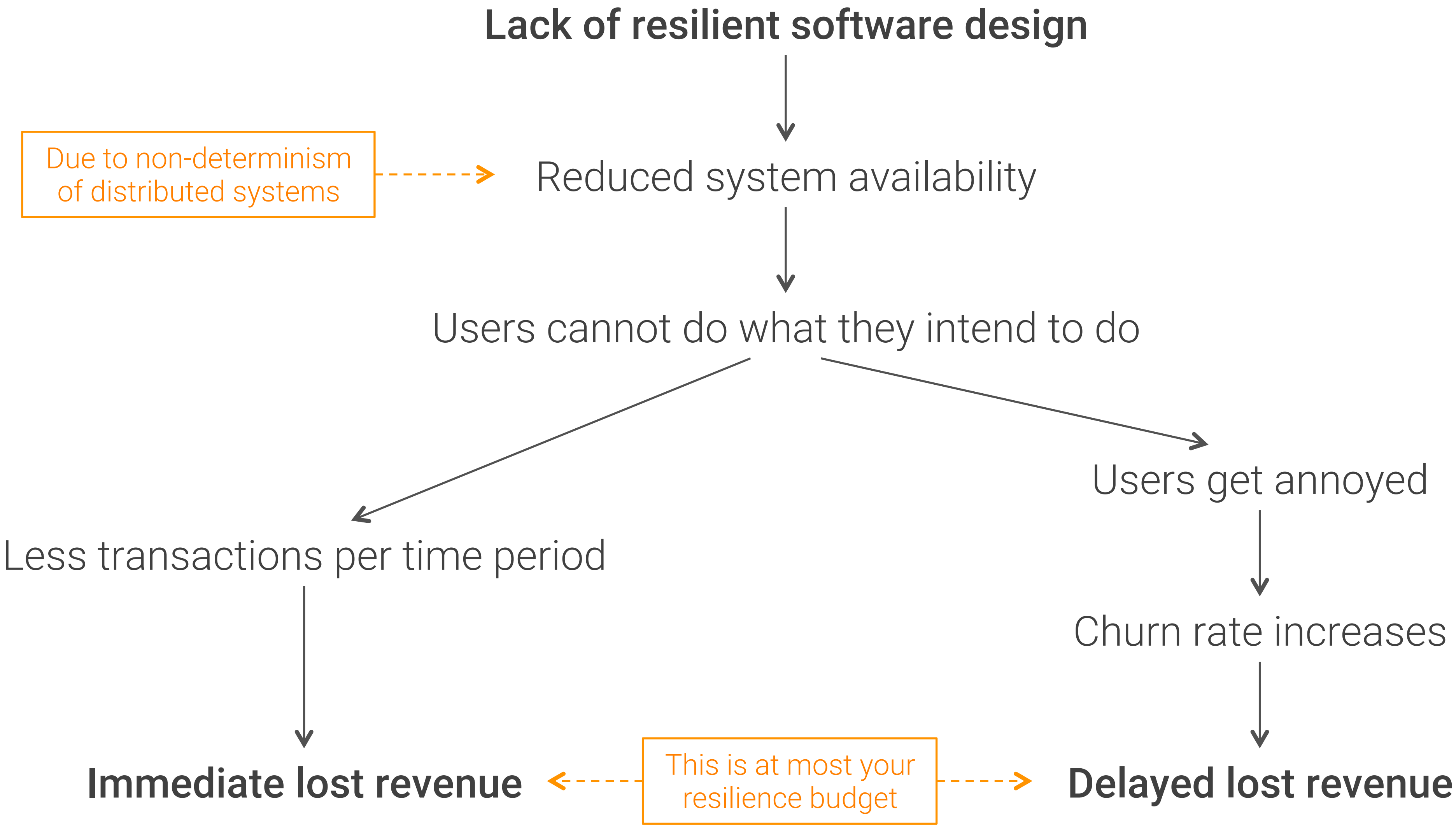
Understand the business case

“How much money will we earn with it?”

“Does it improve our velocity?”

Resilience is not about making money

Resilience is about **not losing** money



Lack of resilient software design

Due to non-determinism of distributed systems

Reduced system availability

Users cannot do what they intend to do

Less transactions per time period

Users get annoyed

Churn rate increases

Immediate lost revenue

This is at most your resilience budget

Delayed lost revenue

Quest #2

Embrace distributed systems

Everything fails, all the time.

-- Werner Vogels

What we learned in our IT education

If X then Y

Inside process thinking

Reasoning about deterministic behavior

Designing a complicated system

We are good at this (due to how our brains work)

What we need for distributed systems

If X then *maybe* Y

This changes everything!

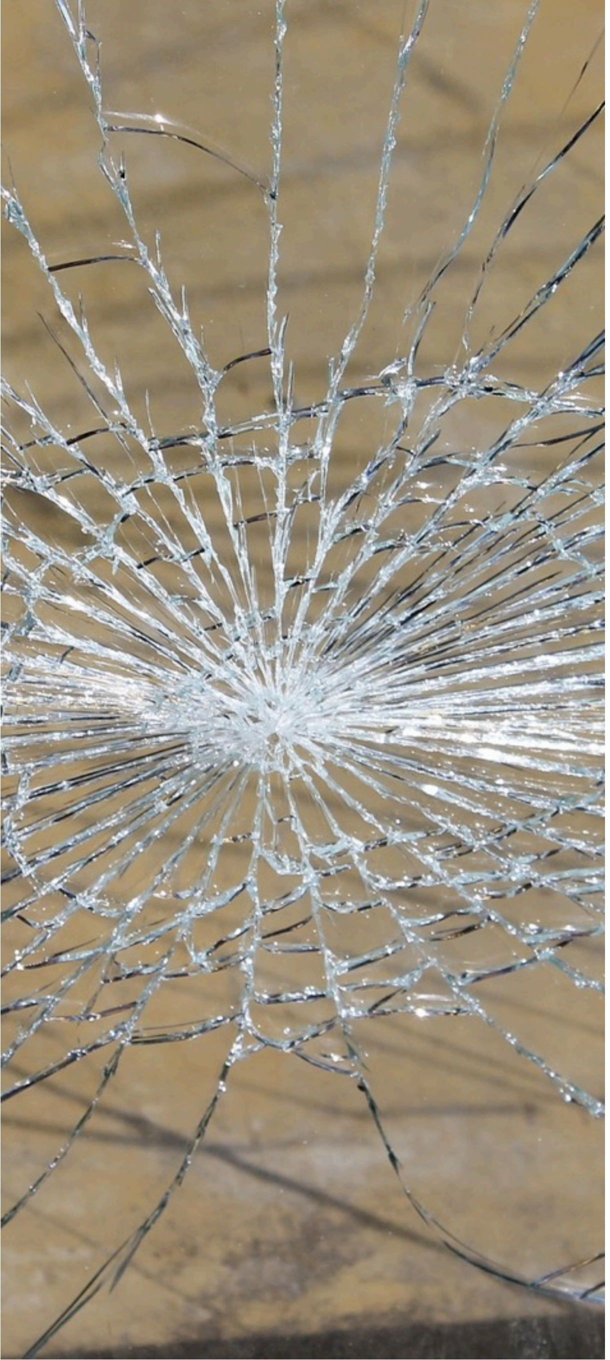
Across process thinking

Reasoning about non-deterministic behavior

Designing a complex system

We are poor at that (due to how our brains work)

Yet, we usually use deterministic thinking
to reason about distributed systems



Failures in distributed systems ...

- Crash failure
- Omission failure
- Timing failure
- Response failure
- Byzantine failure

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

Time & Ordering

Leslie Lamport

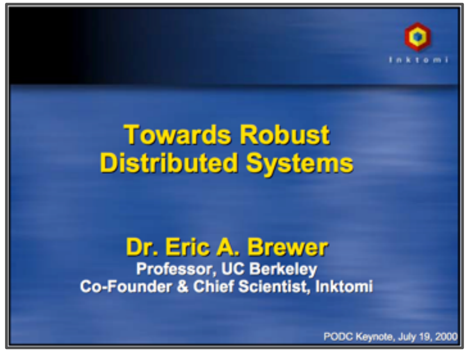
"Time, clocks, and the ordering of events in distributed systems"

The Part-Time Parliament
Leslie Lamport

Consensus

Leslie Lamport

"The part-time parliament" (Paxos)



Towards Robust Distributed Systems

Dr. Eric A. Brewer
Professor, UC Berkeley
Co-Founder & Chief Scientist, Inktomi

PODC Keynote, July 19, 2000

CAP

Eric A. Brewer

"Towards robust distributed systems"

The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

Faulty processes

Leslie Lamport,
Robert Shostak,
Marshall Pease

"The Byzantine generals problem"

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER
Yale University, New Haven, Connecticut
NANCY A. LYNCH
Massachusetts Institute of Technology, Cambridge, Massachusetts
AND
MICHAEL S. PATERSON
University of Warwick, Coventry, England

Consensus

Michael J. Fischer,
Nancy A. Lynch,
Michael S. Paterson

"Impossibility of distributed consensus with one faulty process" (FLP)

A Hundred Impossibility Proofs for Distributed Computing

Nancy A. Lynch *
Lab for Computer Science
MIT, Cambridge, MA 02139
lynch@tds.lcs.mit.edu

Impossibility

Nancy A. Lynch

"A hundred impossibility proofs for distributed computing"

... turn seemingly simple issues into very hard ones

Embrace distributed systems

- **Distributed systems introduce non-determinism regarding**
  - **Execution completeness**
  - **Message ordering**
  - **Communication timing**

- You will be affected by this at the application level
  - Don’t expect your infrastructure to hide all effects from you
  - Better have a plan to detect and recover from inconsistencies

But do I really need to care?

(The system, I am working on, is not a distributed system)

(Almost) every system is a distributed system

-- Chas Emerick

http://www.infoq.com/presentations/problems-distributed-systems

… and it's getting "worse"

- Cloud-based systems
- Microservices
- Zero Downtime
- Mobile & IoT
- Social Web

Quest #3

Avoid the “100% available” trap

The “100% available” trap, version #1

You: “How should the application respond if a technical failure occurs?”

Business owner: “This must not happen! It is your responsibility to make sure that this will not happen.”

The “100% available” trap, version #2

You: “How do you handle the situation if the service you call does not respond (or does not respond timely)?”

Developer 1: “We did not implement any extra measures. The other service is so important and thus needs to be so highly available that it is not worth any extra effort.”

Developer 2: “Actually, if that service should be down, we would not be able to do anything useful anyway. Thus, it just needs to be up.”

The question is not, if a failure will happen

The question is, **when** a failure will happen

A short note about availability

Assume a service availability of 99,5% (incl. planned downtimes)

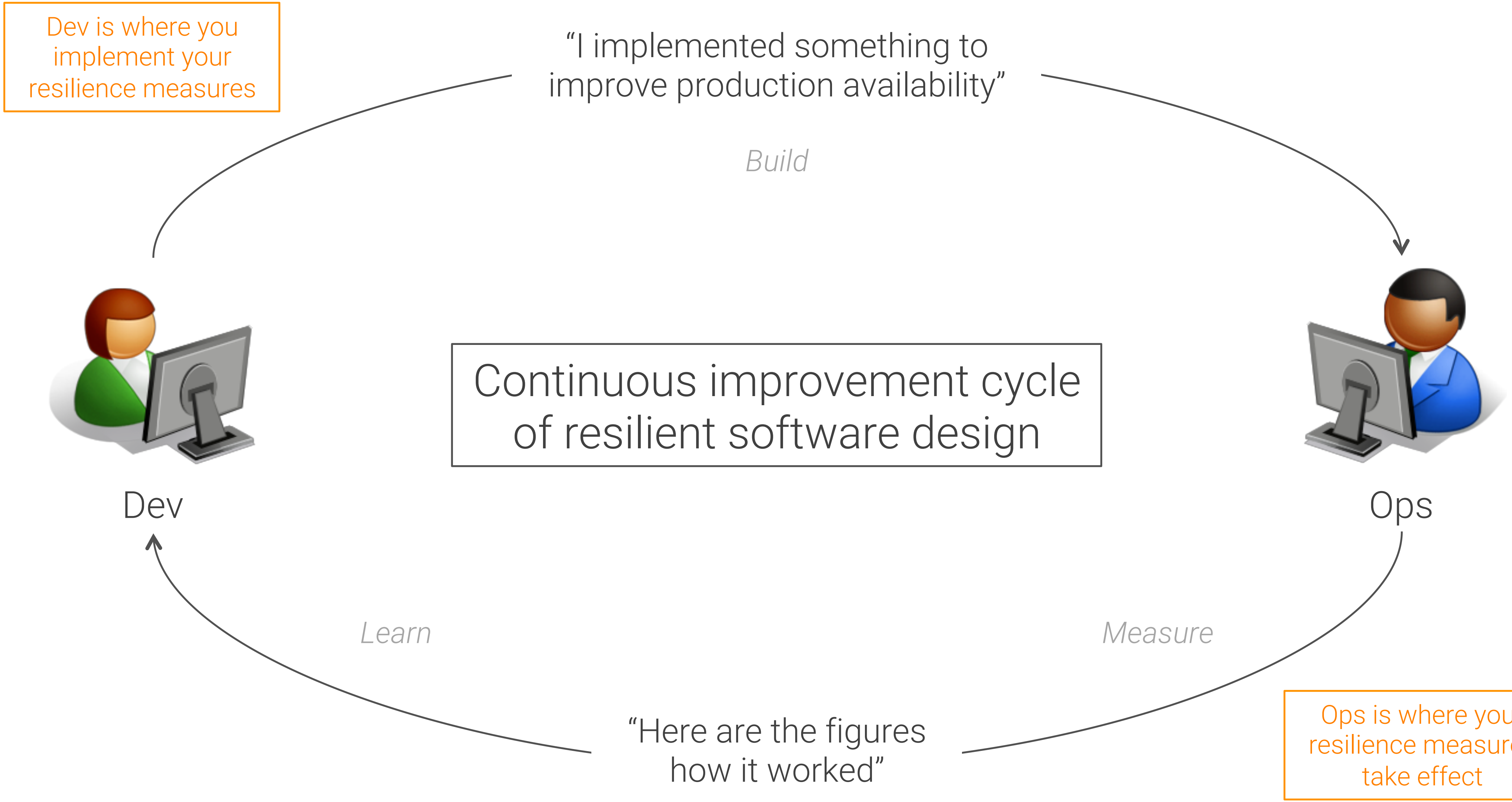
- 10 services involved in a request → 95,1% probability of success
- 50 services involved in a request → 77,8% probability of success

Quest #4

Establish the ops-dev feedback loop

The big wall between Dev and Ops

In a distributed environment, you cannot solve availability issues on an infrastructure level only



Continuous improvement cycle of resilient software design

“I implemented something to improve production availability”

Build

Dev

Ops

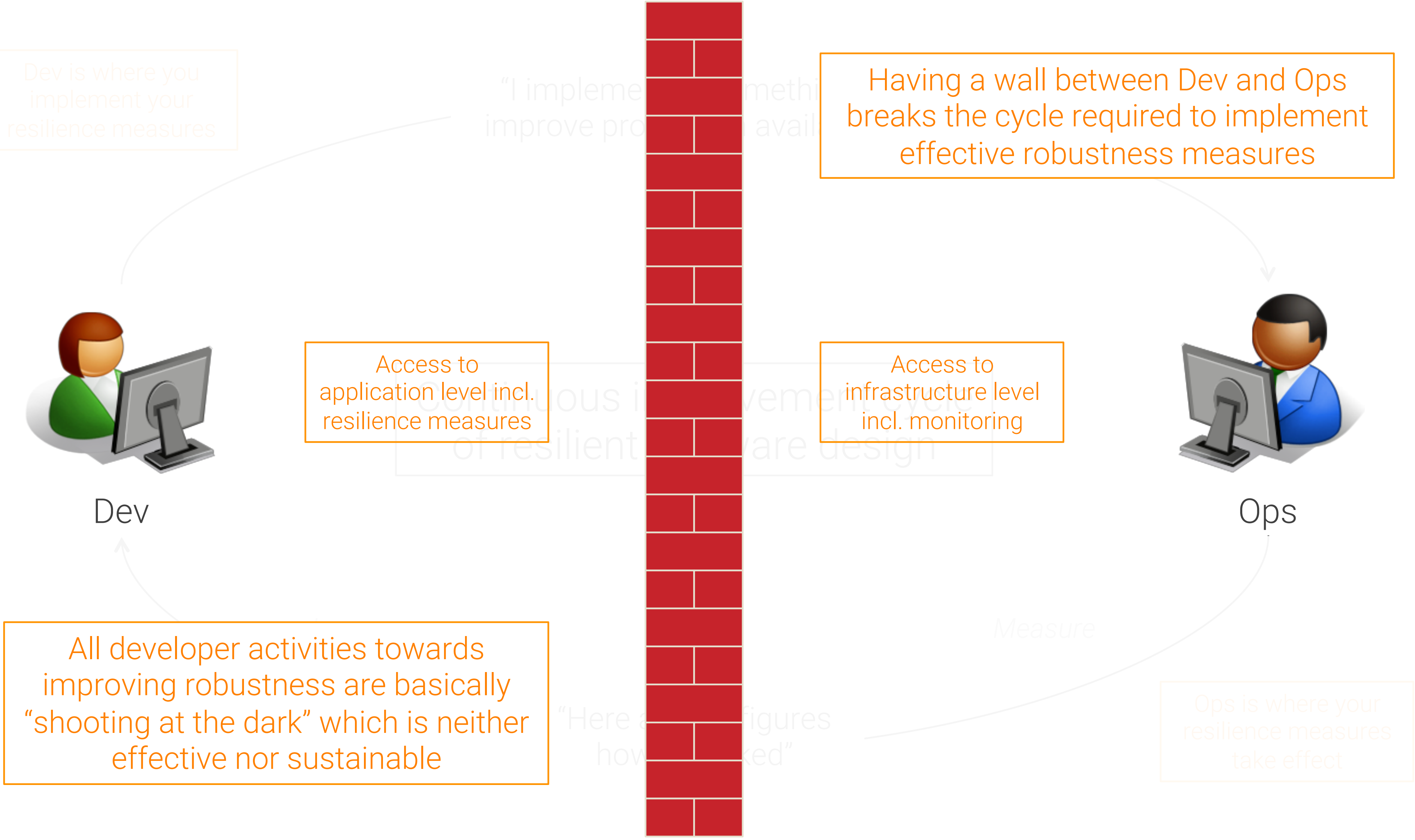
Measure

“Here are the figures how it worked”

Learn

Dev is where you implement your resilience measures

Ops is where your resilience measures take effect



Dev is where you implement your resilience measures

“I impleme… methi… improve pro… avail…

Having a wall between Dev and Ops breaks the cycle required to implement effective robustness measures

Access to application level incl. resilience measures

Continuous i… vement cycle of resilient … ware design

Access to infrastructure level incl. monitoring

Measure

All developer activities towards improving robustness are basically “shooting at the dark” which is neither effective nor sustainable

“Here a… figures how… ked”

Ops is where your resilience measures take effect

For effective resilient software design

you need a working ops-dev feedback loop

Establishing the feedback loop

- Adopt DevOps
- Adopt Site Reliability Engineering (SRE)
- Or do it your own way if you know a better way ...
    - ... but make sure you establish the required feedback loops!

Quest #5

Master functional design

Without proper functional design

nothing else matters

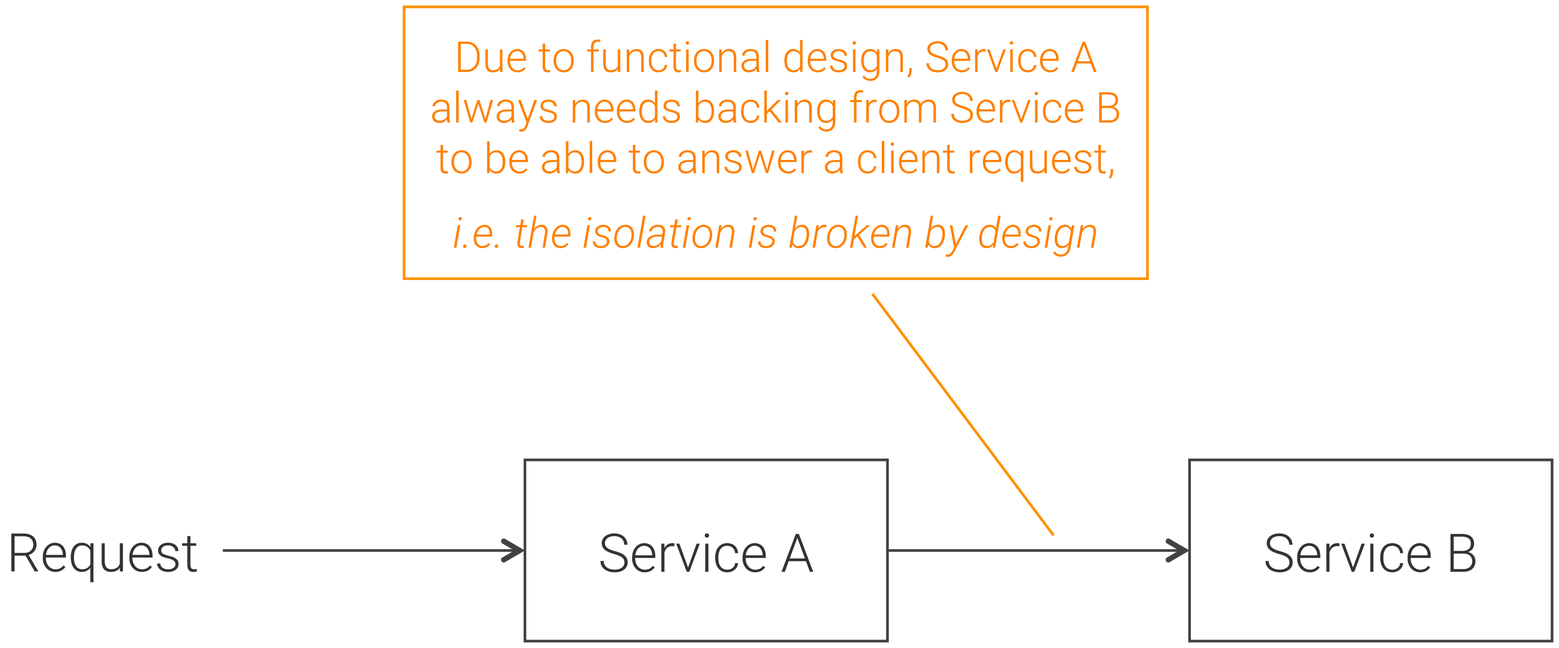
Isolation

- System must not fail as a whole
- Split system in parts and isolate parts against each other
- Avoid cascading failures
- Foundation of resilient software design

Bulkhead

- Bulkheads implement the “parts” that need to be isolated
- Core isolation pattern (a.k.a. “failure units” or “units of mitigation”)
- Diverse implementation choices available, e.g., (micro)services, actors, SCS, ...
- Shaping good bulkheads is a pure functional design issue (and extremely hard)

Hmm, sound easy. Why should that be hard?



Due to functional design, Service A always needs backing from Service B to be able to answer a client request,

i.e. the isolation is broken by design

Request

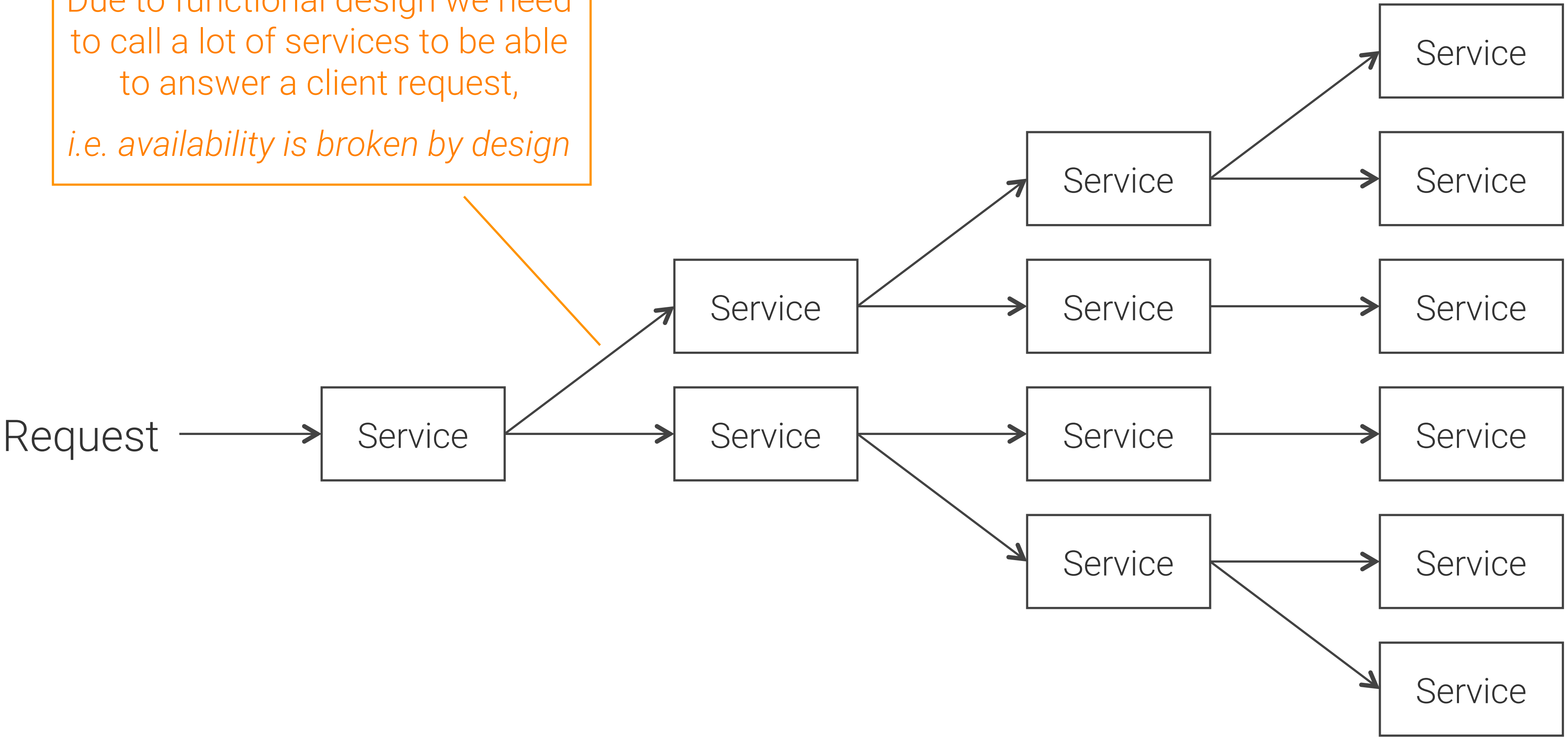
Service A

Service B

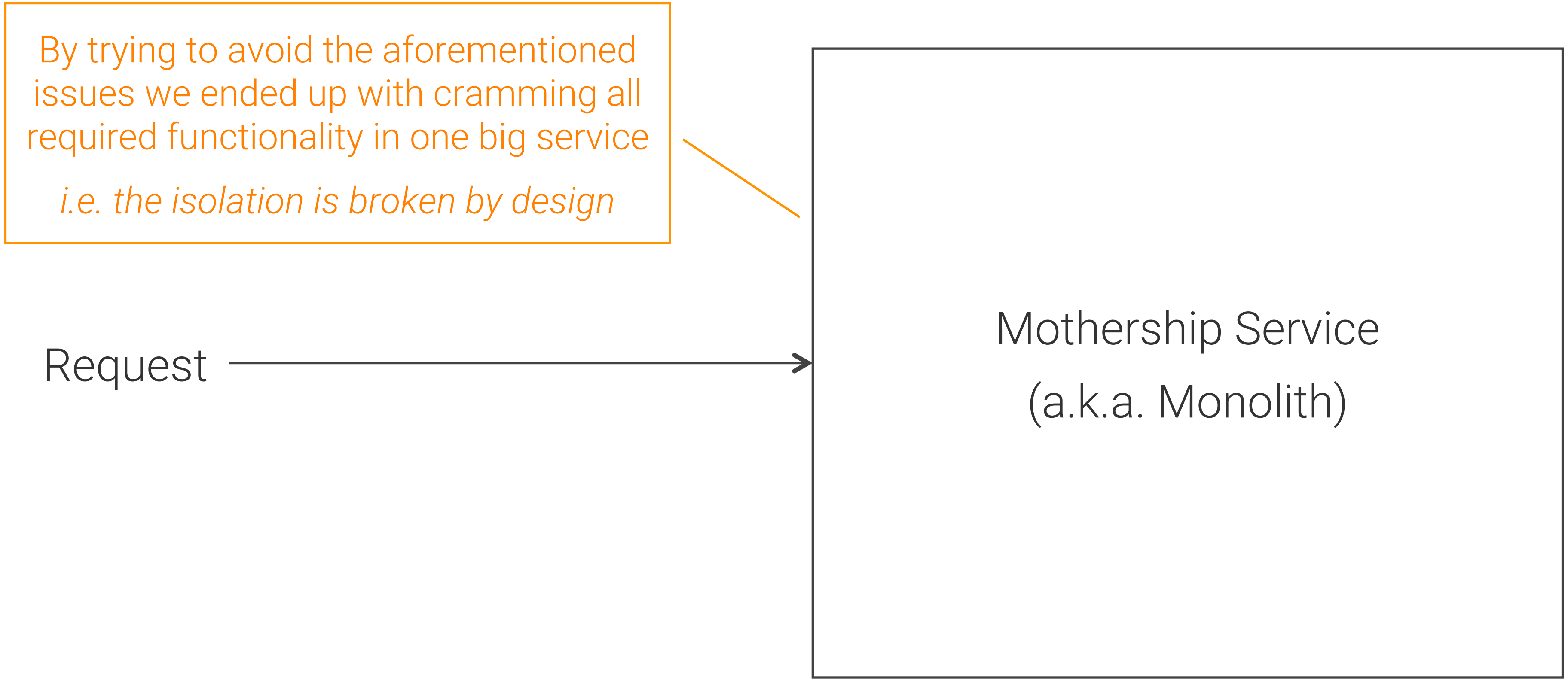
How do we avoid this …

Due to functional design we need to call a lot of services to be able to answer a client request,

i.e. availability is broken by design



... and this ...



By trying to avoid the aforementioned issues we ended up with cramming all required functionality in one big service

i.e. the isolation is broken by design

Request

Mothership Service

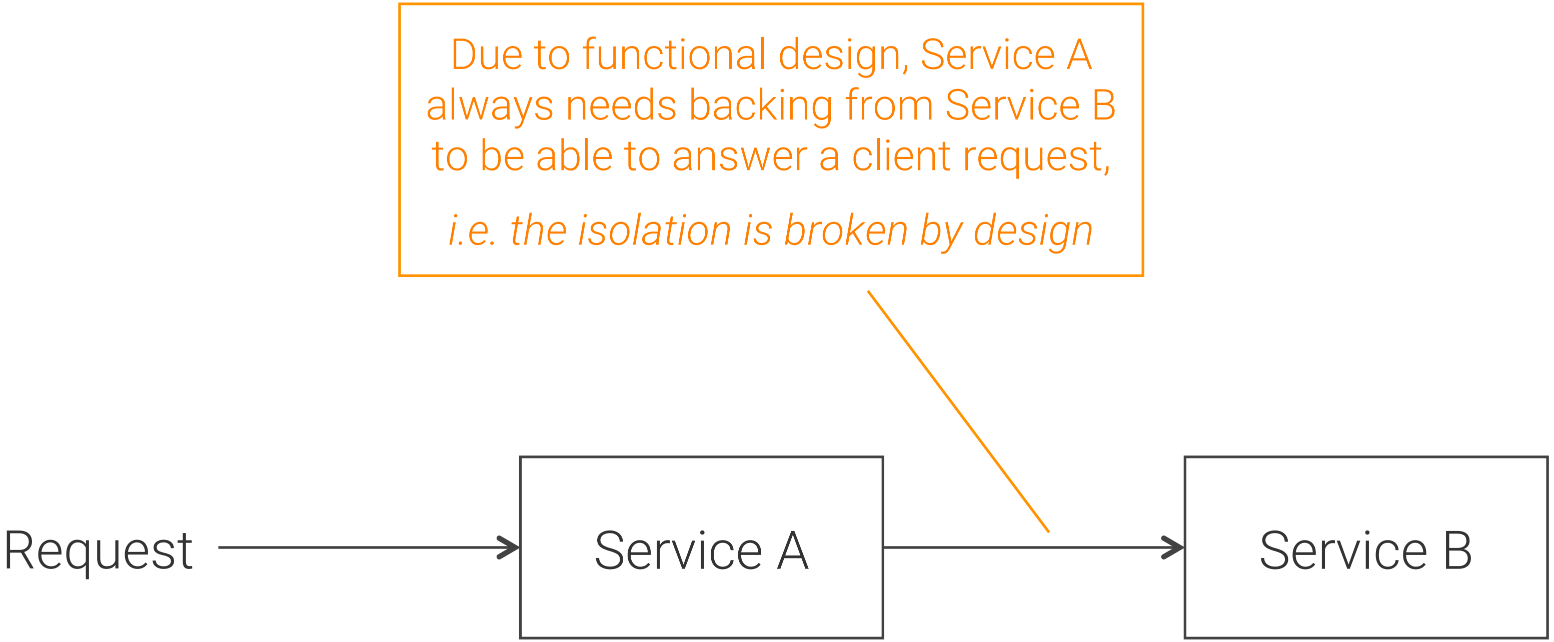
(a.k.a. Monolith)

... without ending up with this?

Let us apply our well-known best practices

- Divide & conquer a.k.a. functional decomposition
- DRY (Don’t Repeat Yourself)
- Design for reusability
- Layered architecture
- …

Unfortunately …



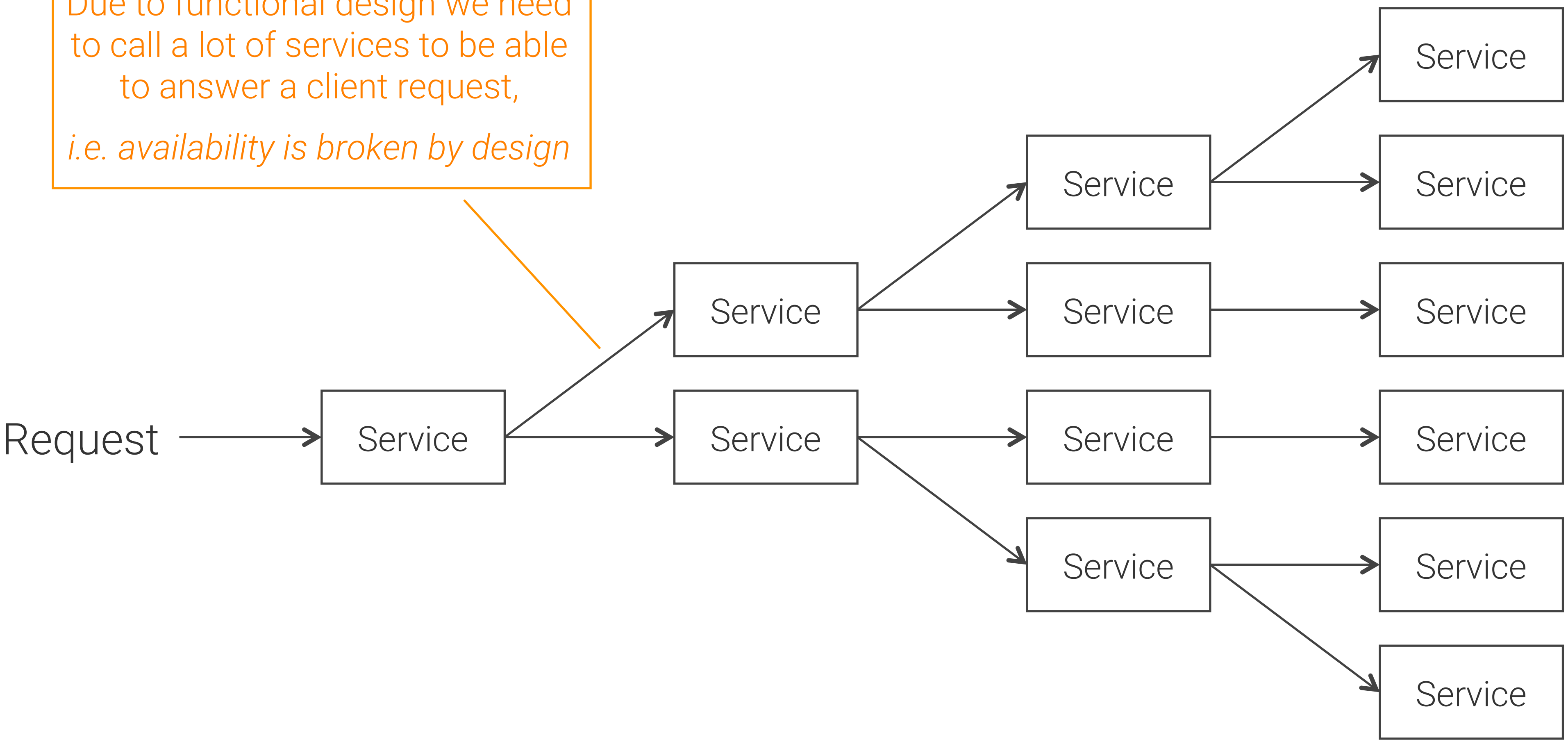
Due to functional design, Service A always needs backing from Service B to be able to answer a client request,

i.e. the isolation is broken by design

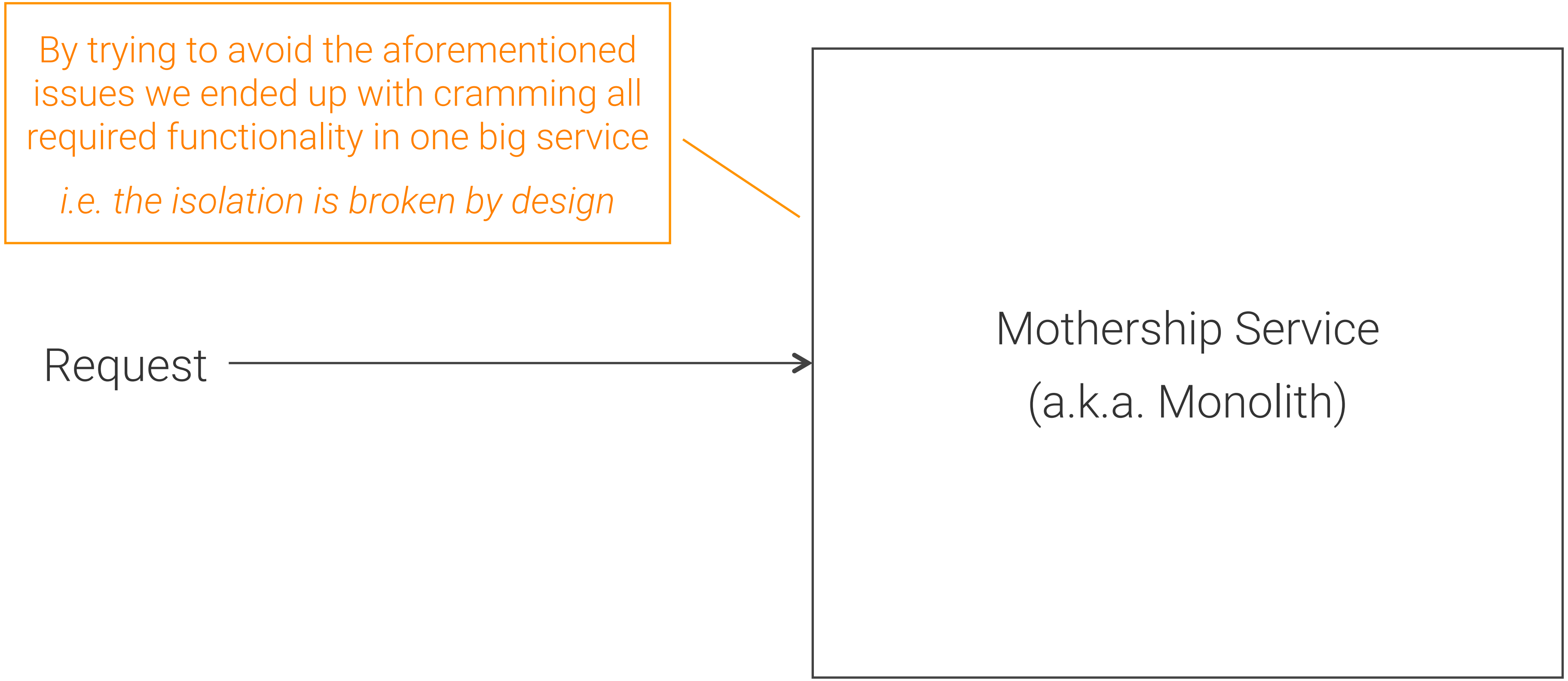
… this usually leads to this …

Due to functional design we need to call a lot of services to be able to answer a client request,

i.e. availability is broken by design



... and this ...



By trying to avoid the aforementioned issues we ended up with cramming all required functionality in one big service

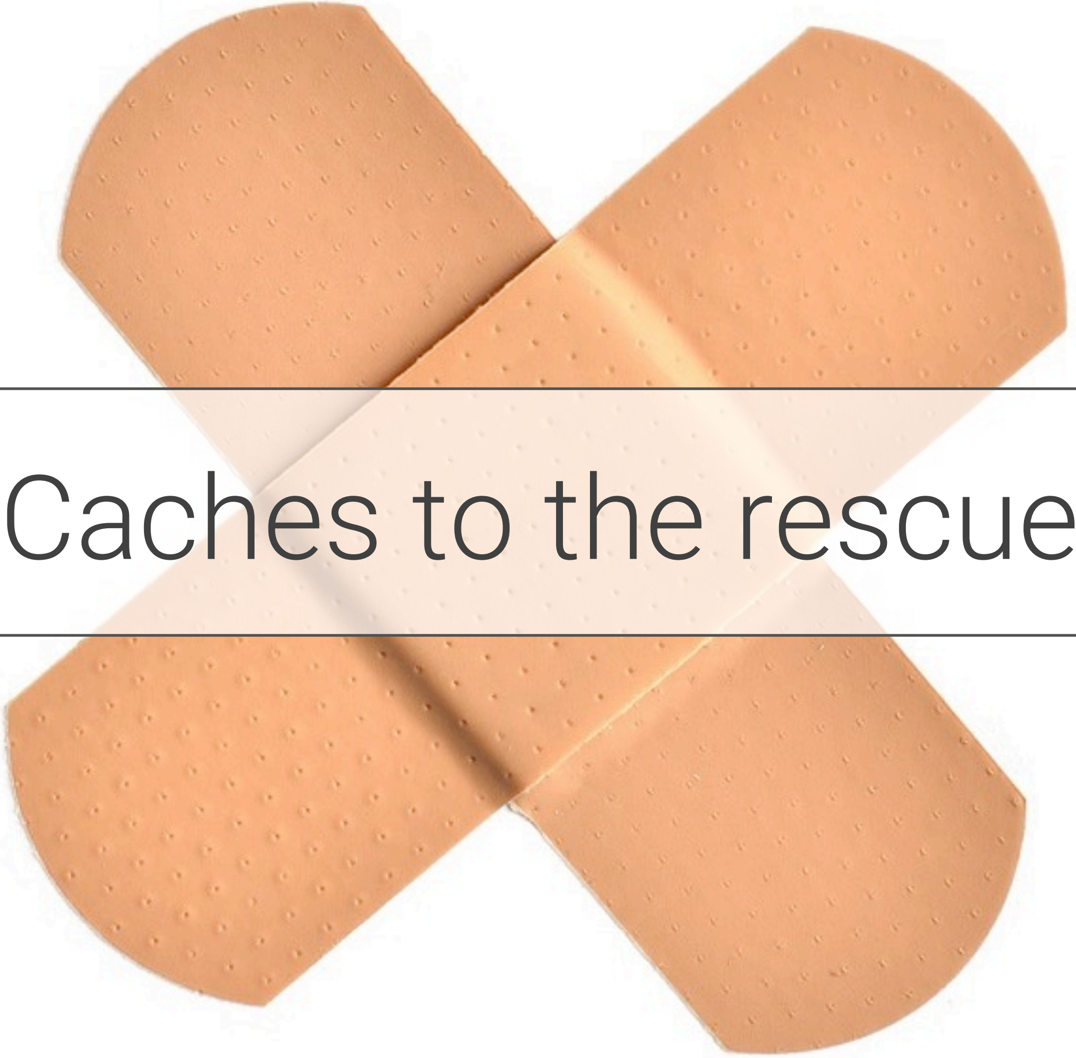
i.e. the isolation is broken by design

Request

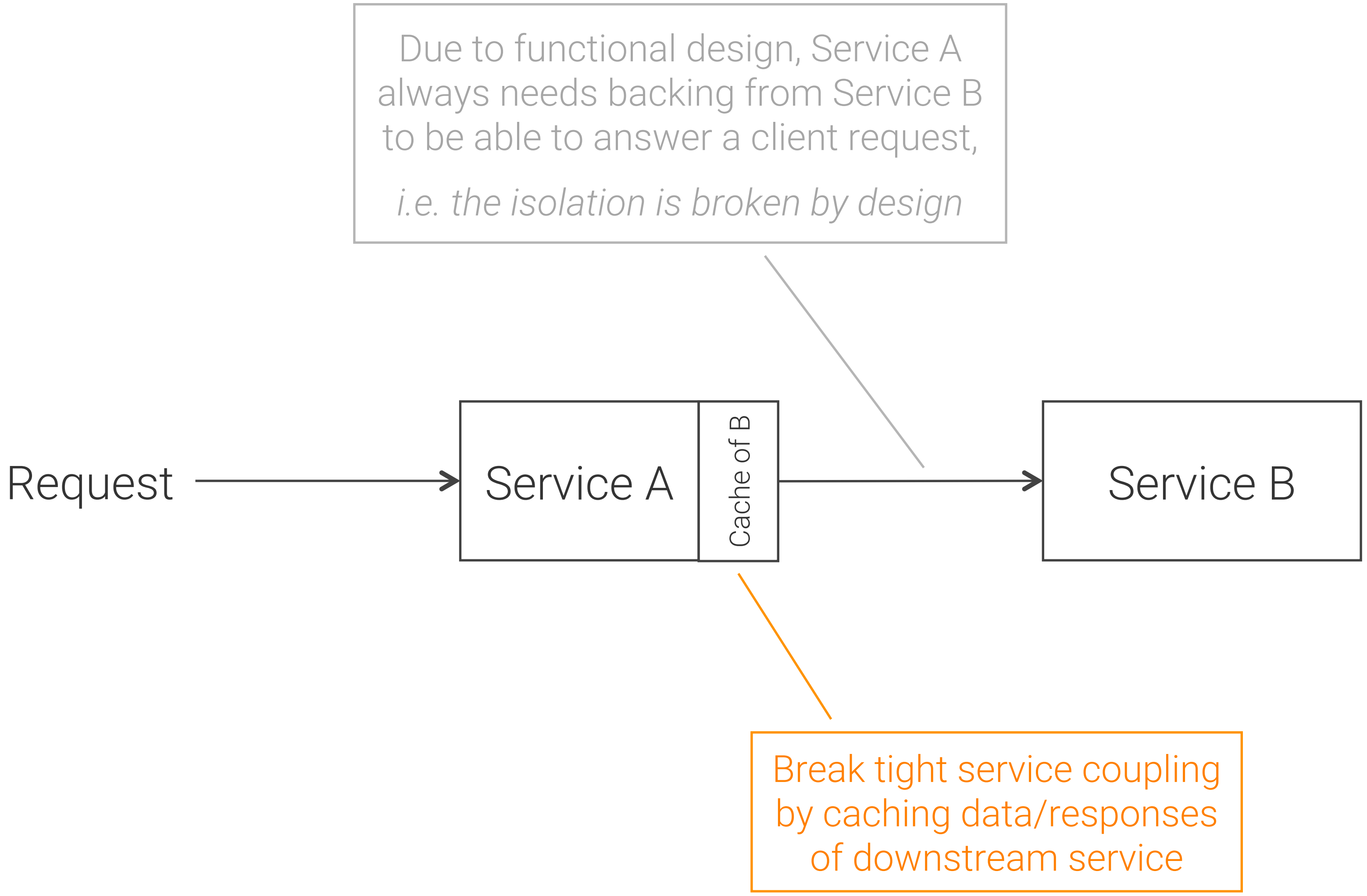
Mothership Service
(a.k.a. Monolith)

... and in the end also often to this.

Welcome to distributed hell!



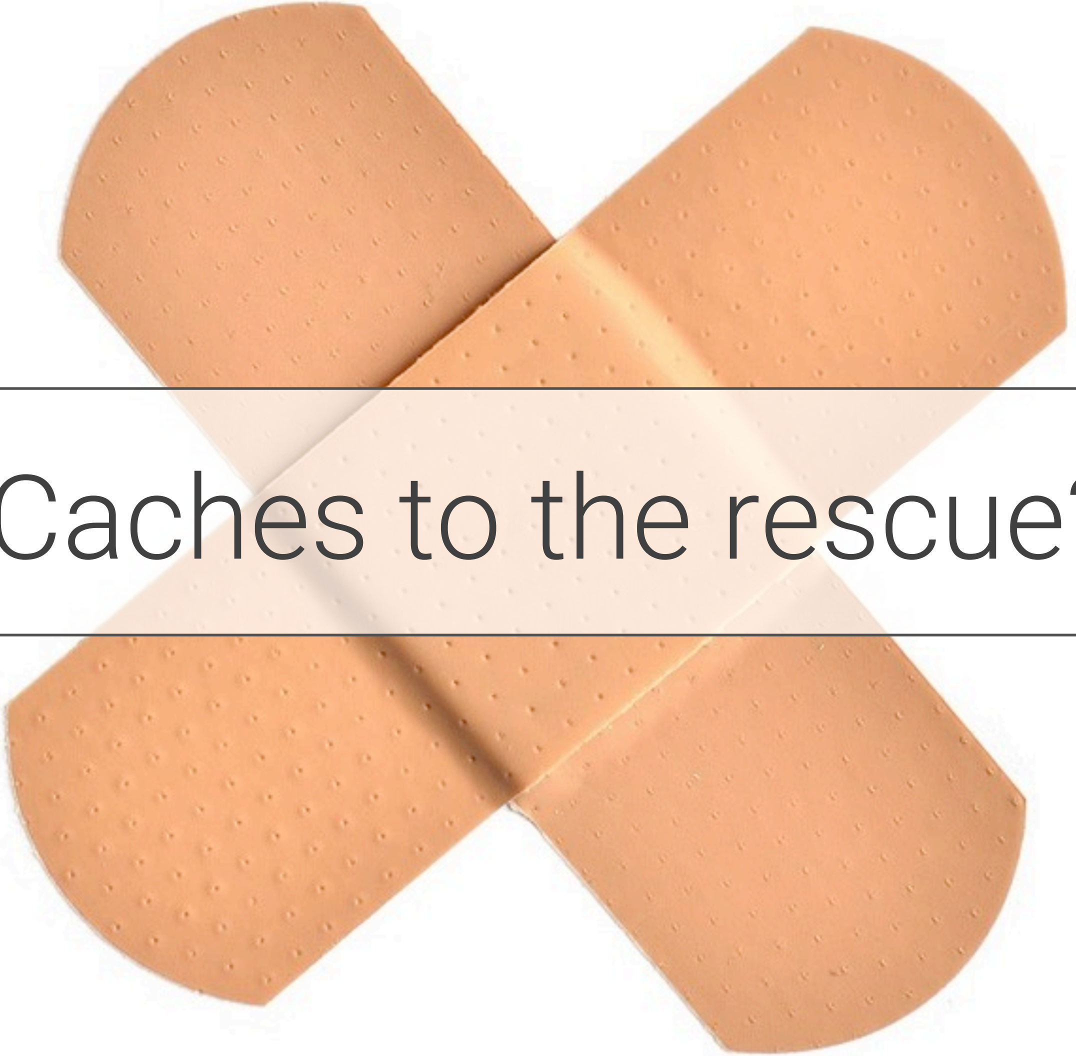
Caches to the rescue!



Due to functional design, Service A always needs backing from Service B to be able to answer a client request,

i.e. the isolation is broken by design

Break tight service coupling by caching data/responses of downstream service



Caches to the rescue?

Do you really think
that copying stale data all over your system
is a suitable measure
to fix an inherently broken design? *

* Side note: Caches are a very important and powerful measure in many places. But they are not suitable as a cheap fix for a broken functional design

We have to re-learn design for distributed system

No silver bullet

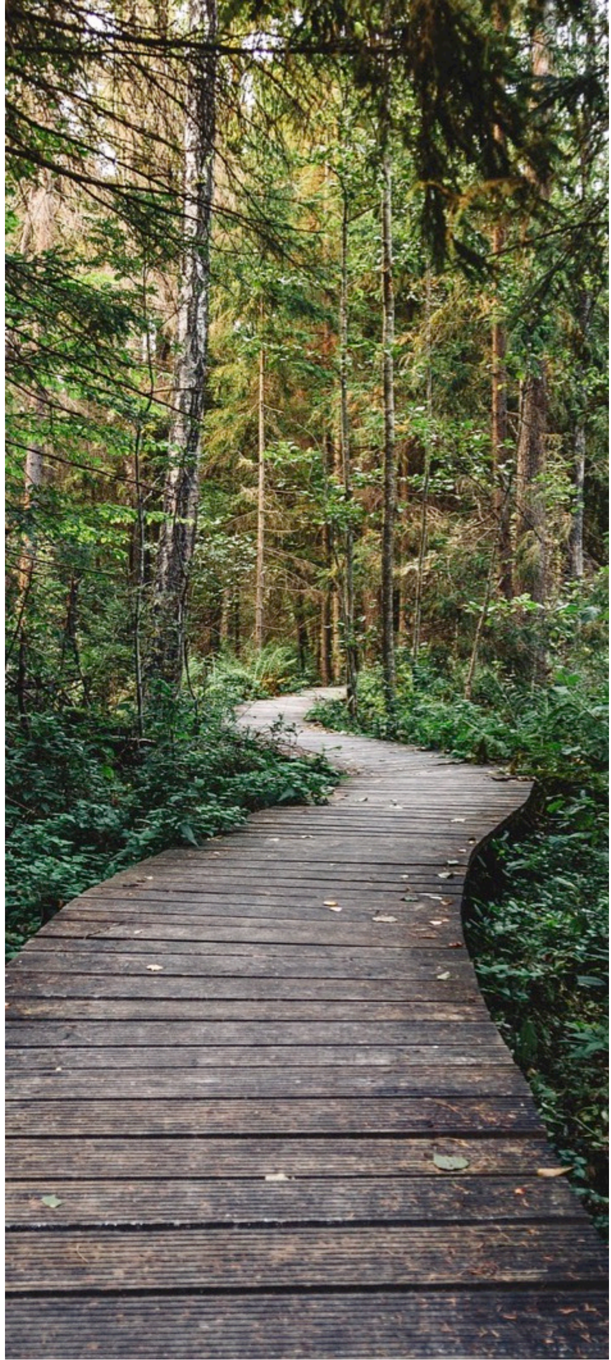
Yet, a few guiding thoughts ...

Foundations of design

- “High cohesion, low coupling” & “separation of concerns”
  - “Crucial across process boundaries
  - Still poorly understood issue
- Start with
  - Understanding organizational boundaries
  - Understanding use cases and flows
  - Identifying functional domains (→ DDD)
  - Finding areas that change independently
  - Do *not* start with a data model!

Short activation paths

- Long activation paths affect availability
- Increase likelihood of failures
- Minimize remote calls per request
- Need to balance opposing forces
    - Avoid monolith → clear separation of concerns
    - Minimize requests → cluster functionality & data
    - Caches can sometimes help, but stale data as trade-off

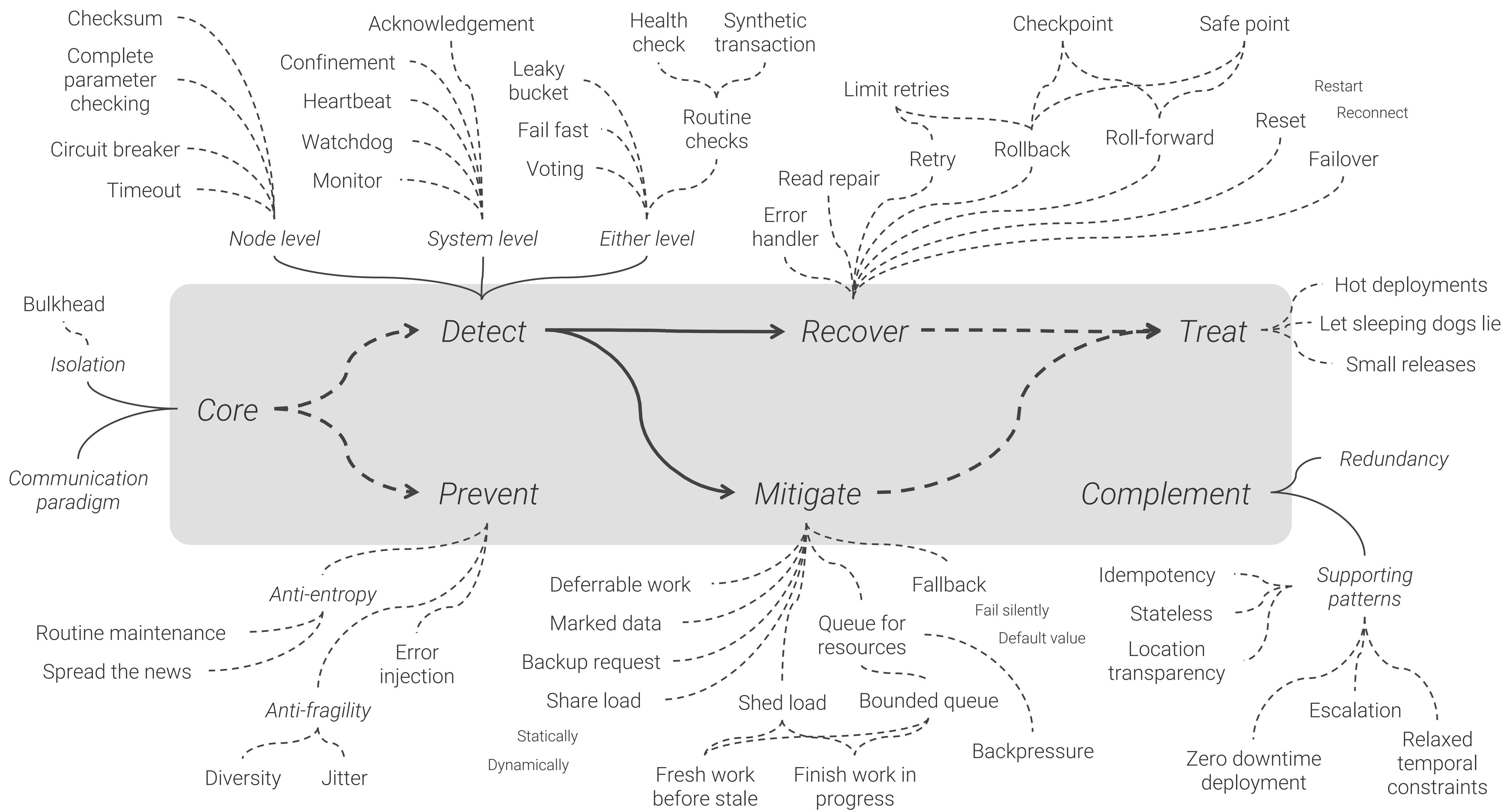


Be (extremely) wary of reusability

- Reusability increases coupling
- Reusability usually leads to bad service design
- Reusability compromises availability
- Reusability rarely pays
- Do not strive for reusable services
- Strive for replaceable services instead
- Try to tackle reusability issues with libraries

Quest #6

Know your toolbox


Checksum
Complete parameter checking
Circuit breaker
Timeout
Node level
Acknowledgement
Confinement
Heartbeat
Watchdog
Monitor
System level
Leaky bucket
Fail fast
Voting
Either level
Health check
Synthetic transaction
Routine checks
Limit retries
Checkpoint
Safe point
Restart
Reconnect
Reset
Rollback
Roll-forward
Retry
Failover
Read repair
Error handler
Bulkhead
Isolation
Communication paradigm
Core
Detect
Recover
Treat
Prevent
Mitigate
Complement
Hot deployments
Let sleeping dogs lie
Small releases
Redundancy
Supporting patterns
Idempotency
Stateless
Location transparency
Escalation
Zero downtime deployment
Relaxed temporal constraints
Anti-entropy
Routine maintenance
Spread the news
Anti-fragility
Diversity
Jitter
Error injection
Deferrable work
Marked data
Backup request
Share load
Statically
Dynamically
Shed load
Fresh work before stale
Finish work in progress
Queue for resources
Bounded queue
Backpressure
Fallback
Fail silently
Default value

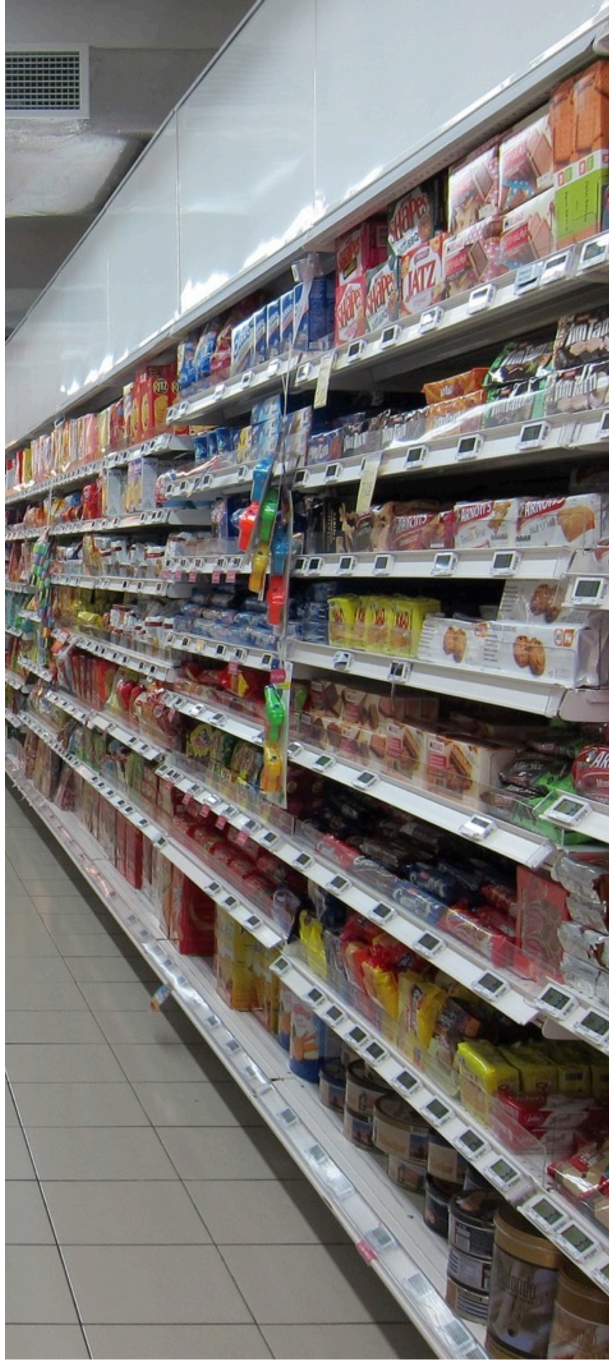
Using resilience patterns

- Patterns are options, not obligations
- Don’t pick too many patterns

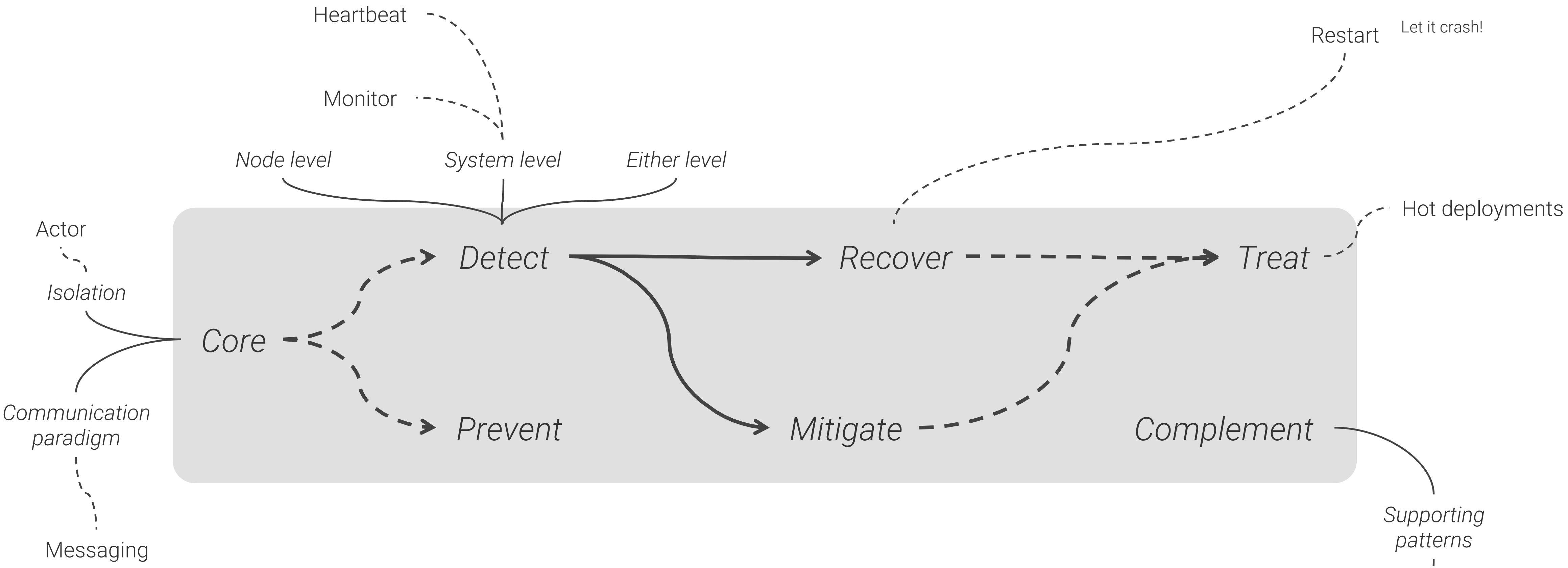
- Each pattern increases complexity
- Complexity is the enemy of robustness

- Each pattern costs money in dev & ops
- You only have a limited resilience budget

- Look for complementary patterns



How other people did it

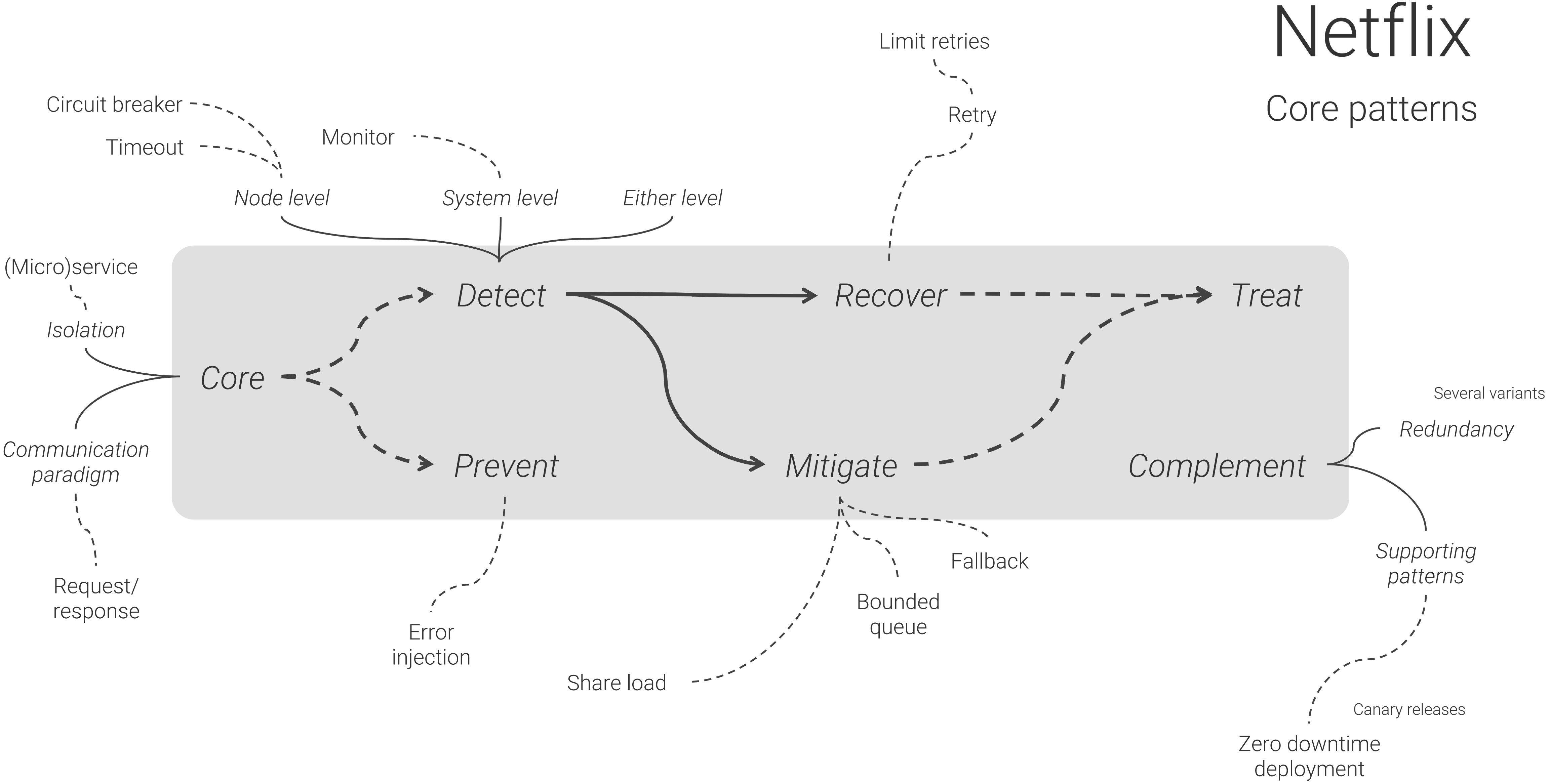


Erlang (Akka)

Core patterns

Netflix

Core patterns



Quest #7

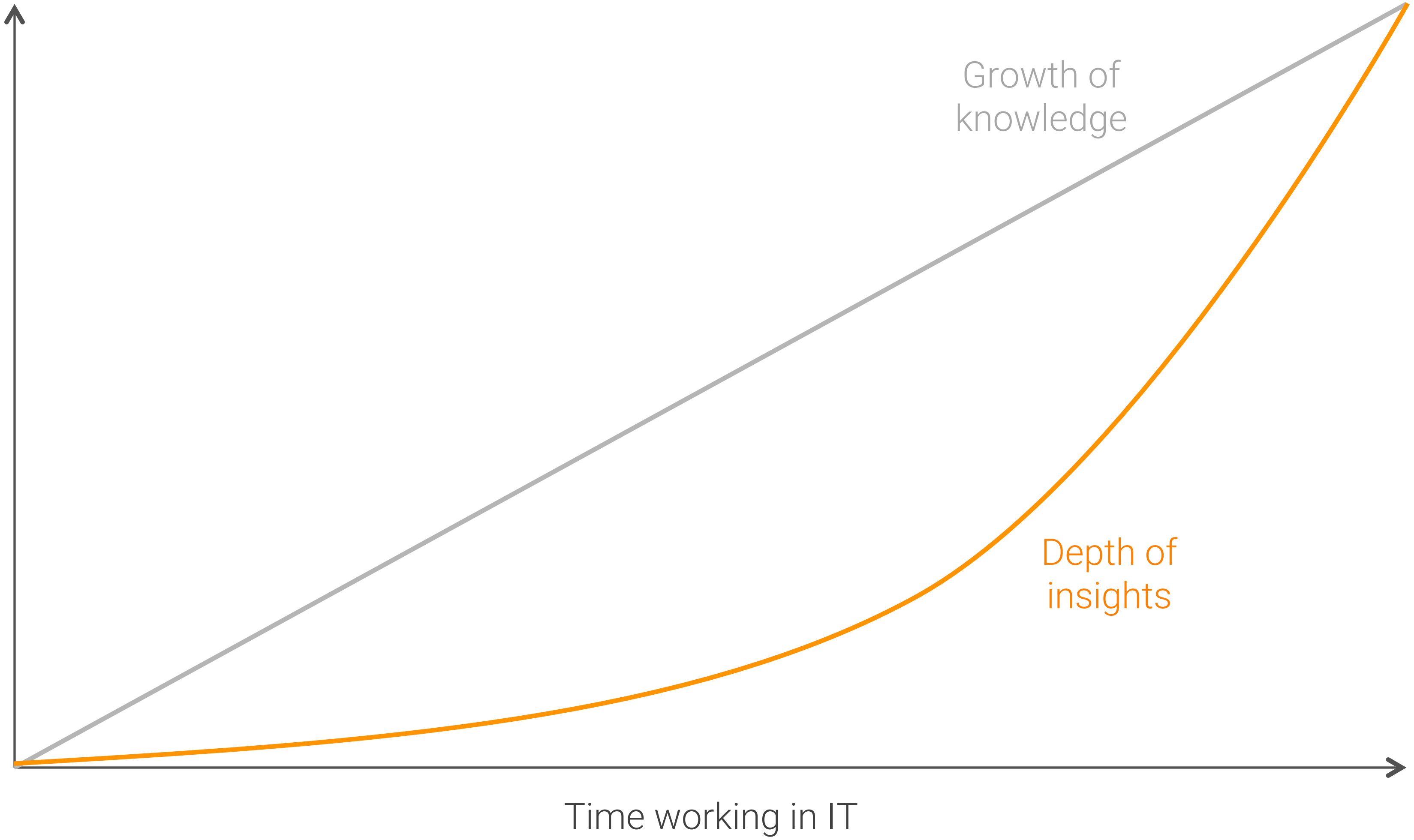
Preserve the collective memory

We face a new generation of developers

every 5 years

We loose our collective memory every 5 years *

* Mean time until a topic discussion in the community starts over form scratch


Growth of knowledge
Depth of insights
Time working in IT

What do we do to compensate this effect?

We look for the new & shiny stuff ...

… as anything not new must be useless crap!

We need to rediscover our insights

every 5 years

In IT, we suffer from

continuous collective amnesia

and we are even proud of it!

How can we become better?

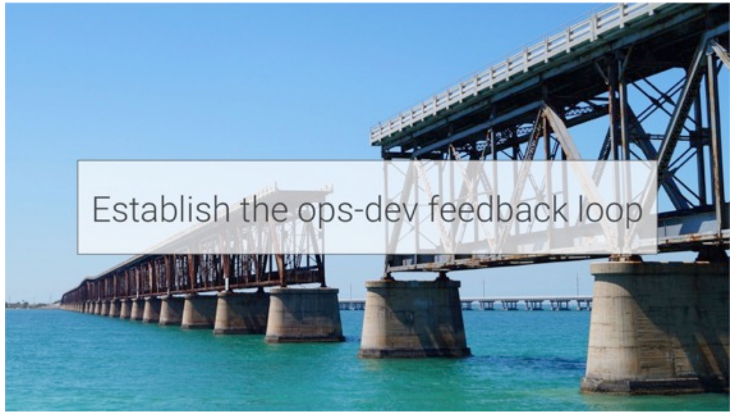
Wrap-up

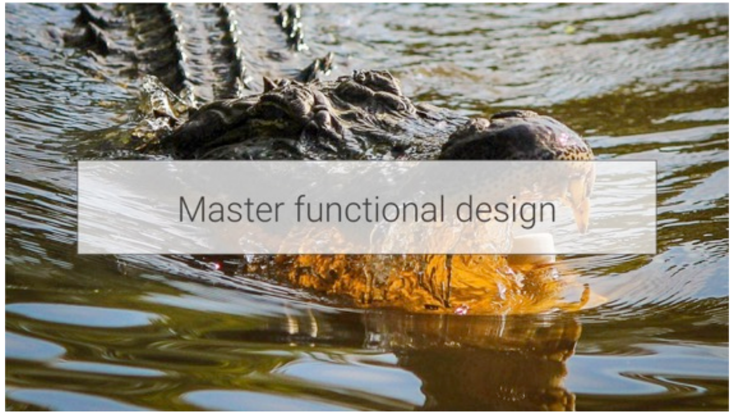
The 7 quests at a glance

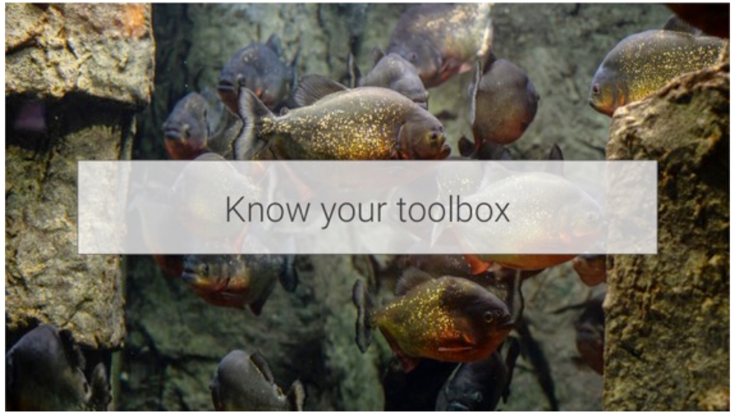
Understand the business case

Embrace distributed systems

Avoid the “100% available” trap


Establish the ops-dev feedback loop

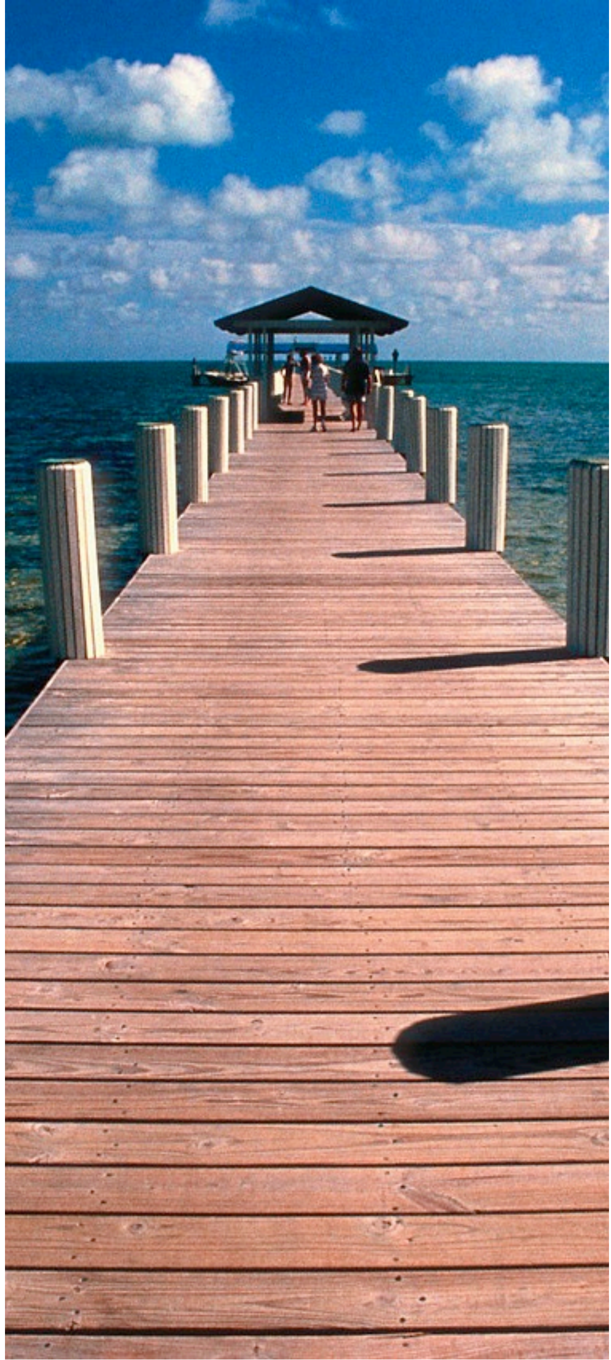

Master functional design


Know your toolbox

Preserve the collective memory

Wrap-up

- The road to resilient software design is a twisted one!
- Most challenges are only indirectly related to RSD
- Most challenges are not coding related
- Mastering functional design is extremely hard ...
  - ... while learning the patterns is relatively easy
- How do we preserve our collective memory?



Uwe Friedrichsen

IT traveller.

Dot Connector.

Cartographer of uncharted territory.

Keeper of timeless wisdom.

CTO and Fellow at codecentric.

https://www.slideshare.net/ufried

https://medium.com/@ufried

@ufried