Microservices

stress-free and without increased heart-attack risk

Uwe Friedrichsen (codecentric AG) – microxchg – Berlin, 12. February 2015

@ufried

Uwe Friedrichsen | uwe.friedrichsen@codecentric.de | http://slideshare.net/ufried | http://ufried.tumblr.com

tl;dr

Can I have stress-free µservices?

<Hysterical laughter>

Of course not !!!

:-(((… and now ???

You have the choice between the hard way and the not so hard way

Okay, let's try the not so hard way

Why aren’t μservices easy?

A single µservice is easy …

… but the complexity of the business functionality remains the same

☛ Complexity is shifted from single µservices to µservice collaboration

µServices are usually self-contained …

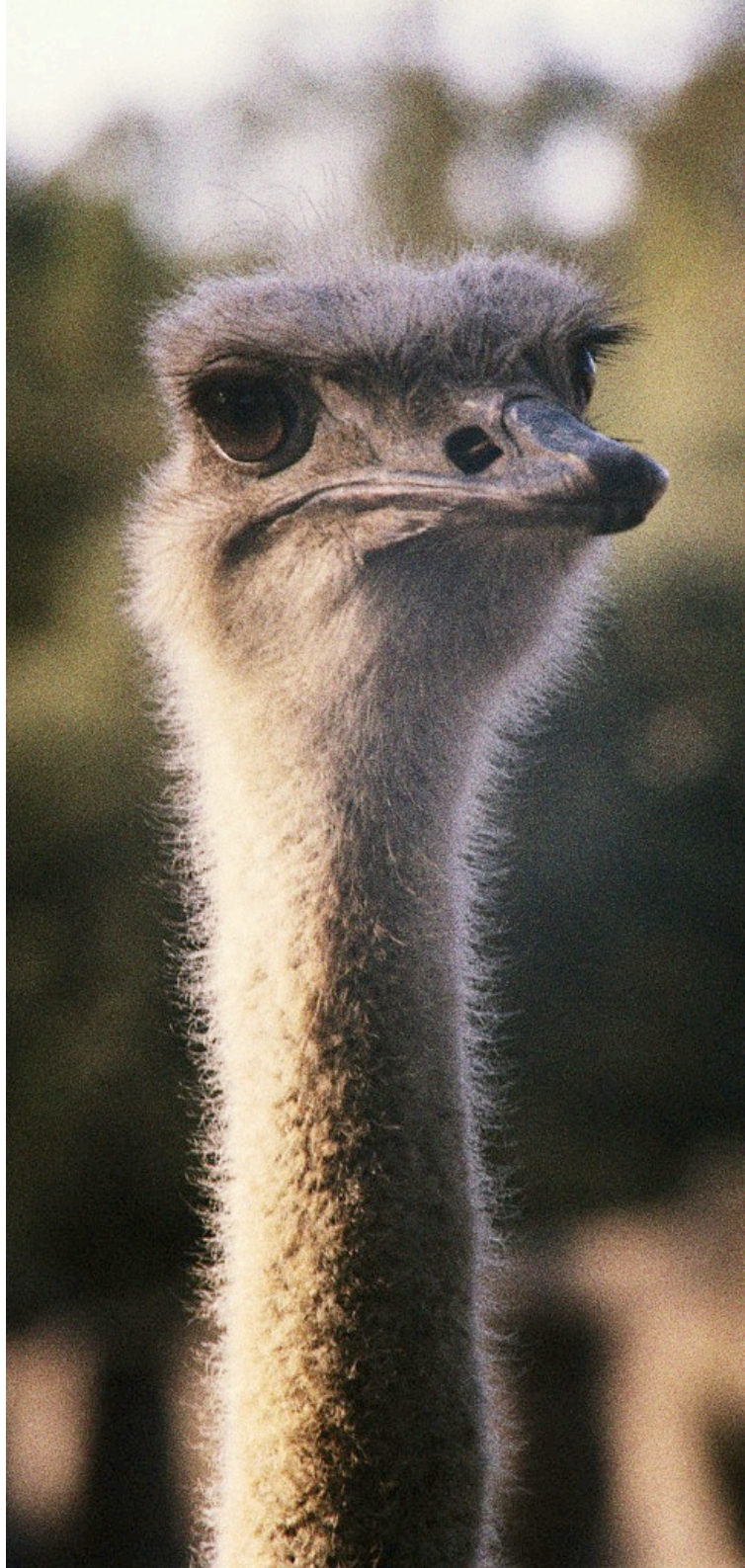
… i.e., µservices are independent runtime processes

☛ This results in a *highly interconnected, distributed system landscape*

Consequences

- Design is more challenging
- Implementation is more challenging
- Distributed systems are challenging
  - Lookup
  - Liveness
  - Partitioning
  - Latency
  - Consistency
  - …
- New challenges for “monolith developers”

→ µServices are not easy



But then … why are we doing μservices?

The pros of μservices

- Improved business responsiveness
- Improved business flexibility

- Team autonomy (Conway's law)

- Easy, isolated deployment
- Better scalability
- Replaceability (Lean Startup)

… if done right (no free lunch)

It’s an architectural tradeoff

µService

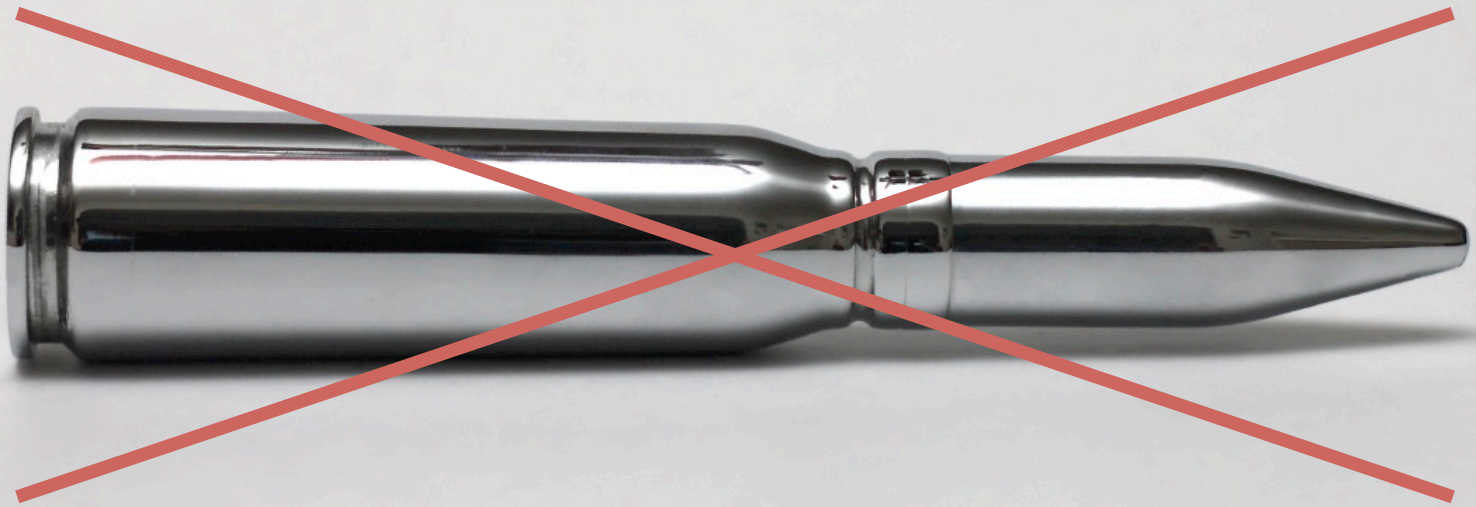
- Responsiveness
- Autonomy
- Many unknowns

Monolith

- Cost-Efficiency
- Standardization
- Well known

Then let’s talk about the not so hard way

Disclaimer



Topic areas

Design

Interfaces

User Interface

Frameworks

Datastores

Developer Runtime Environment

Deployment

Production

Resilience

"It seems as if teams are jumping on μservices because they're sexy, but the design thinking and decomposition strategy required to create a good μservices architecture are the same as those needed to create a well structured monolith.

If teams find it hard to create a well structured monolith, I don't rate their chances of creating a well structured μservices architecture."

- Simon Brown

http://www.codingthearchitecture.com/2014/07/06/distributed_big_balls_of_mud.html

"In theory, programming languages give us all we need to encapsulate state and environment - we just need to use them well.

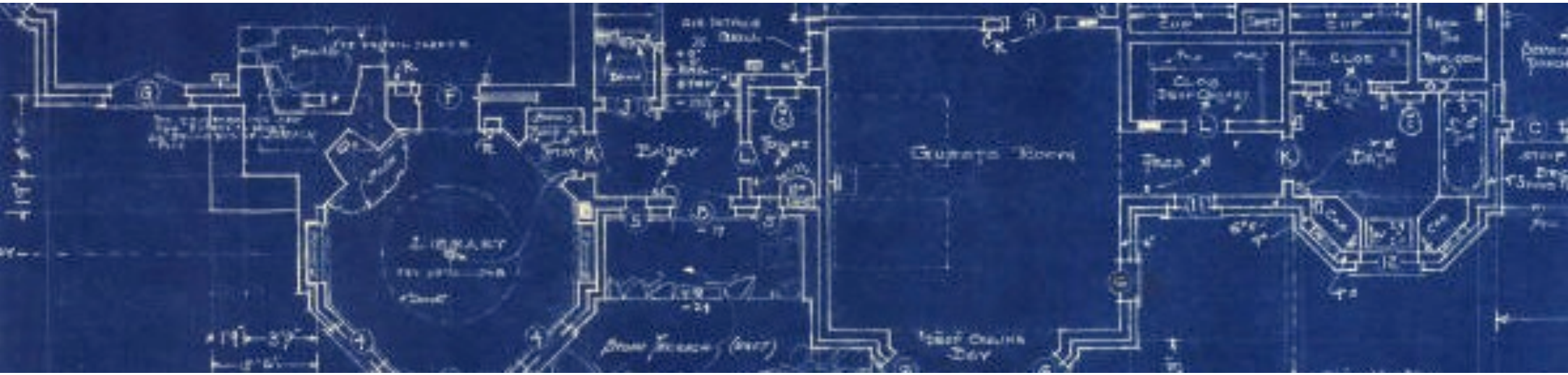
Maybe we just don't have the discipline? Maybe we had to explicitly advocate the practice of writing services running in completely different environments using different languages to trigger the sort of encapsulation that we want? If that's the case, we can either see it as a clever self-hack or something we were forced into by the fact that we programmers are adept at overcoming any sort of self-discipline we try to impose on ourselves.

Perhaps both are true."

- Michael Feathers

https://michaelfeathers.silvrback.com/microservices-and-the-failure-of-encapsulaton

Design



- Master modularization first
- Use bounded contexts as a modularization starting point
- Forget about layered architecture
- Rethink DRY – avoid deployment dependencies

"If every service needs to be updated at the same time it's not loosely coupled"

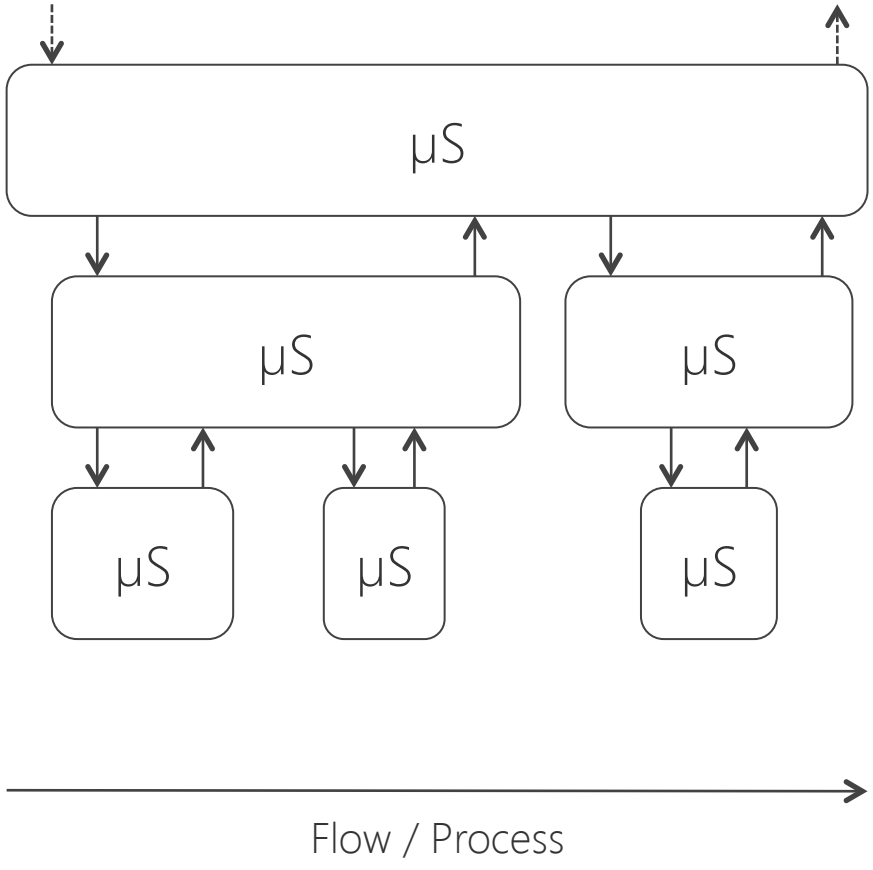
- Adrian Cockcroft

http://de.slideshare.net/adriancockcroft/dockercon-state-of-the-art-in-microservices

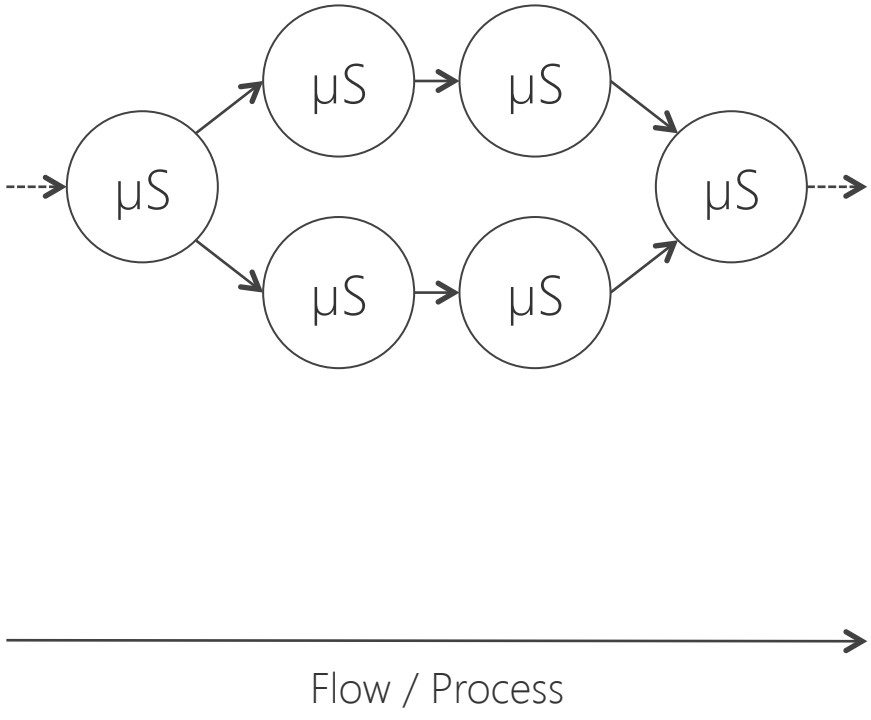
Interfaces

- Use versioned interfaces (don’t forget the interface data model)
- Remember Postel’s law
- Consider API gateways
- Synchronous vs. asynchronous

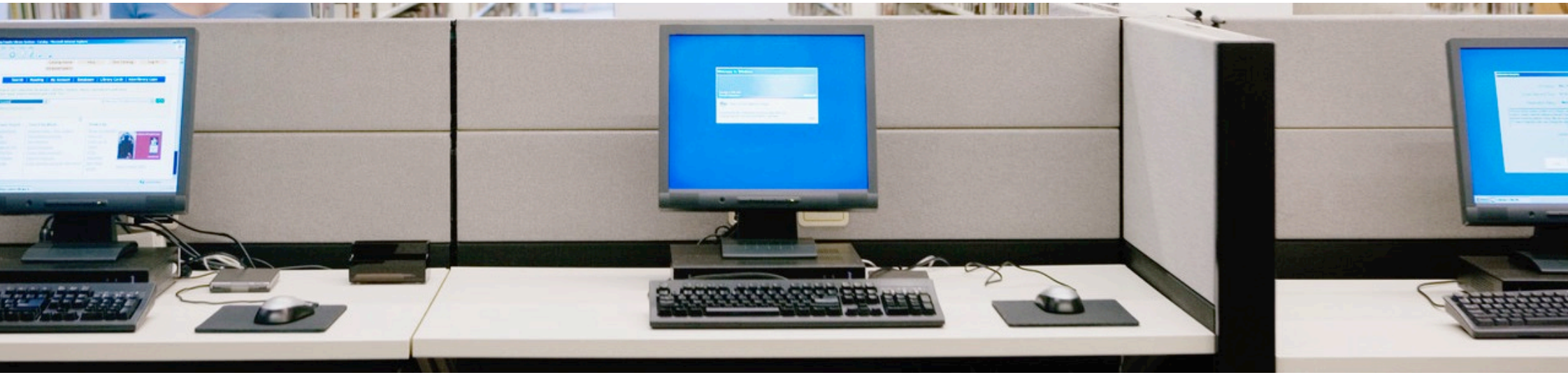
Request/Response : Horizontal slicing



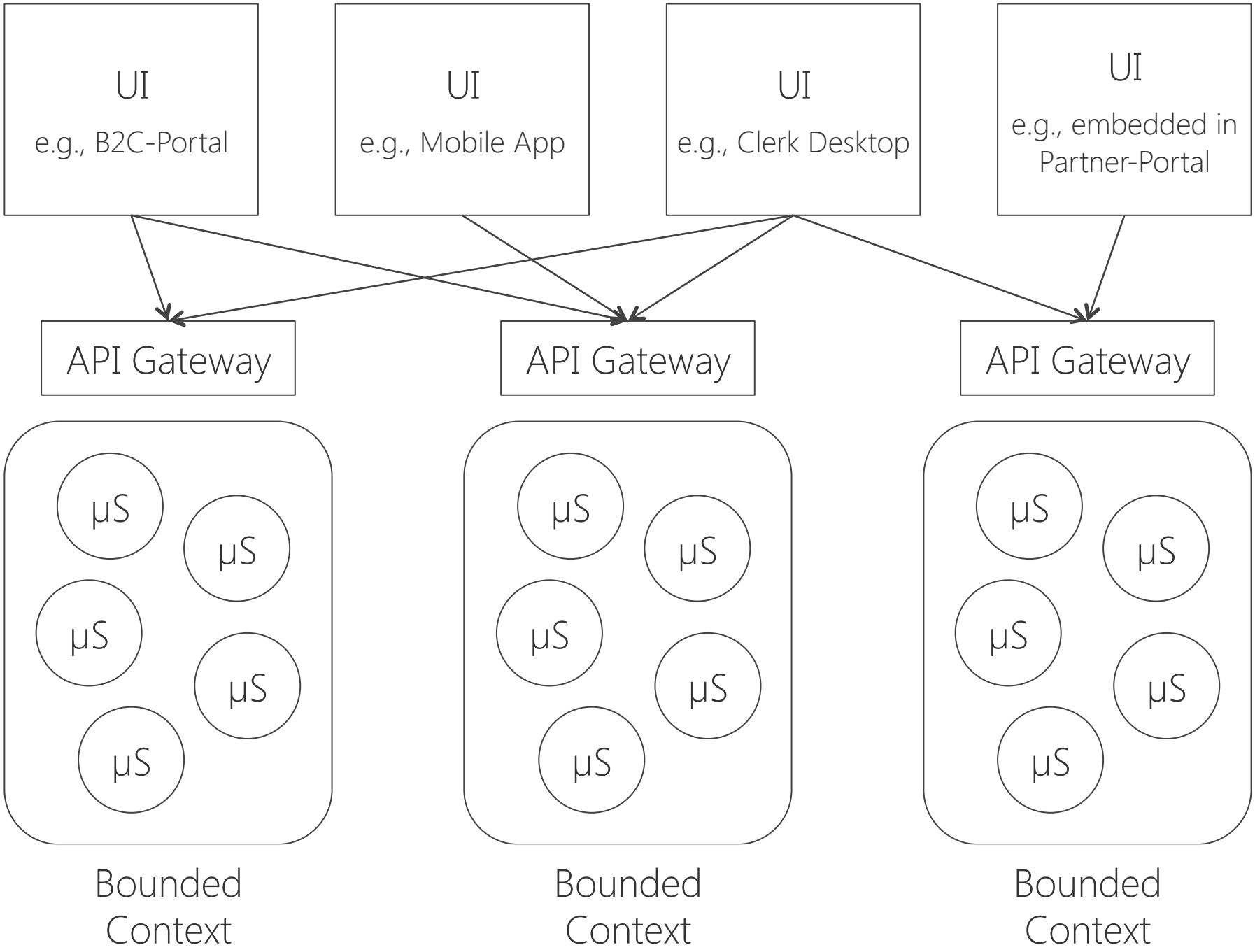
Event-driven : Vertical slicing



User Interface



- The single UI application is history
- Separate UI and services
- Decouple via client centric API gateway
- Including UI in service can make sense in special cases


UI
e.g., B2C-Portal
UI
e.g., Mobile App
UI
e.g., Clerk Desktop
UI
e.g., embedded in Partner-Portal
API Gateway
API Gateway
API Gateway
µS
µS
µS
µS
µS
µS
µS
µS
µS
µS
µS
µS
Bounded Context
Bounded Context
Bounded Context

Frameworks

- Not the most important issue of μservices
- Should support at least uniform interfaces, observability, resilience
- Nice if also support for uniform configuration and discoverability
- Spring Boot/Cloud, Dropwizard, Netflix Karyon, …

Datastores

- Avoid the “single, big database”
- Avoid distributed transactions
- Try to relax temporal constraints (and make actions idempotent)
- Treat your storage as being “ephemeral”

Development Runtime Environment

- Developers should be able to run the application locally
- Provide automatically deployable “development runtime environment”
- Containers are your friend
- Make sure things build and deploy fast locally

Deployment

- Must be “one click”
- Unify deployment artifact format
- Use either IaC tool deployment …
- … or distributed infrastructure & scheduler

Production

- You need to solve at least the following issues
  - Configuration, Orchestration, Discovery, Routing, Observability, Resilience
- No standard solution (yet) in sight
- Container management infrastructures may be of help

Configuration

- Netflix Archaius

Orchestration

- Apache Aurora on Apache Mesos
- Marathon
- Kubernetes
- Fleet

Discovery

- Netflix Eureka
- Apache ZooKeeper
- Kubernetes
- Etcd

Routing

- Netflix Zuul & Ribbon
- Twitter Finagle

Monitoring

- Hystrix
- Twitter Zipkin (Distributed Tracing)

Measuring

- Dropwizard Metrics

Logging

- ELK
- Graylog2
- Splunk

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Leslie Lamport

Failures in complex, distributed, interconnected systems are not an exceptional case

- *They are the normal case*
- *They are not predictable*
- *They are not avoidable*

µService systems *are*
complex, distributed, interconnected systems

Failures in µservice systems are not an exceptional case

- *They are the normal case*
- *They are not predictable*
- *They are not avoidable*

How can I maximize availability
in μservice systems?

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

MTTF: Mean Time To Failure

MTTR: Mean Time To Recovery

Traditional stability approach

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

Maximize MTTF

Resilience approach

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

Minimize MTTR

Do not try to avoid failures. Embrace them.

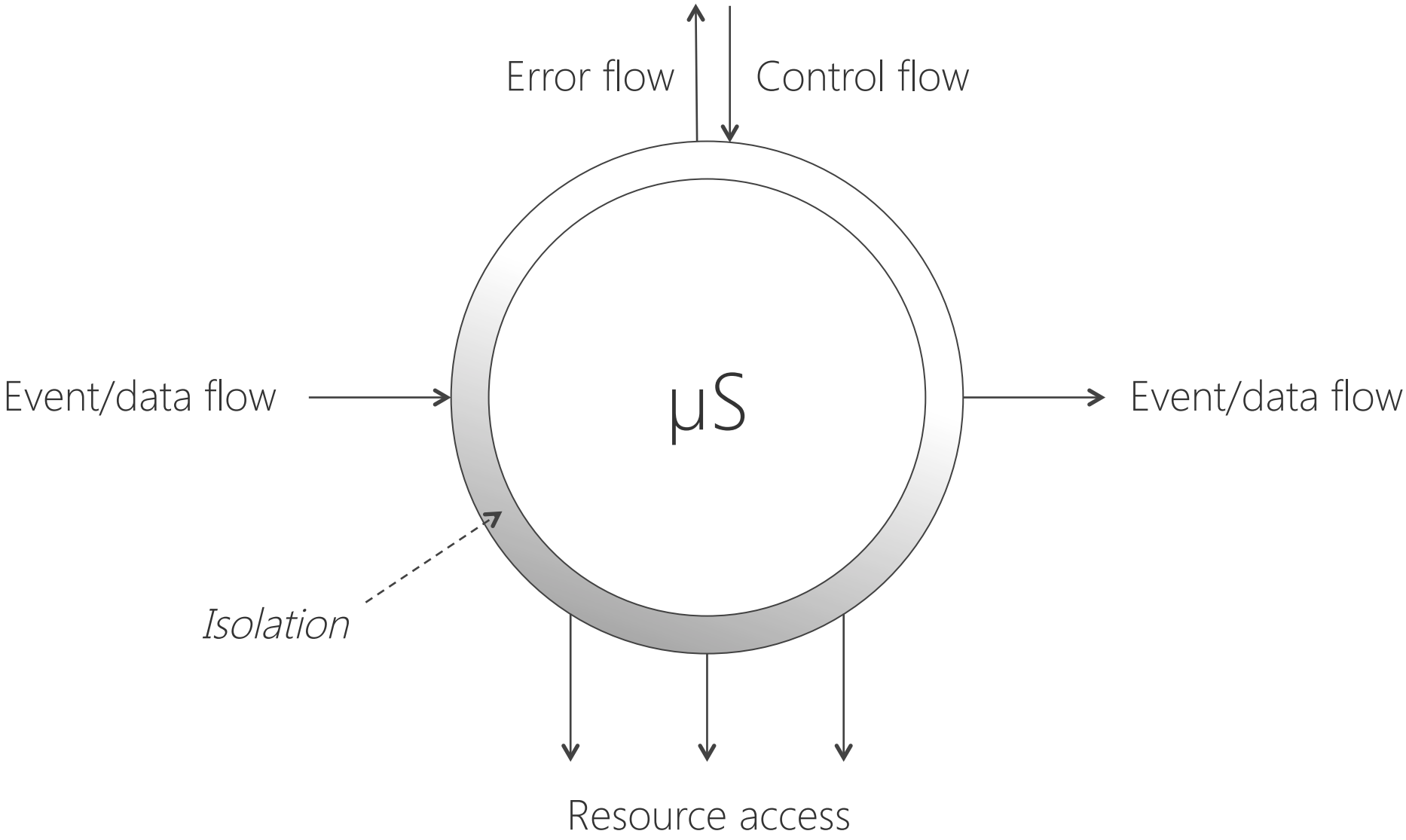
resilience (IT)

the ability of a system to handle unexpected situations

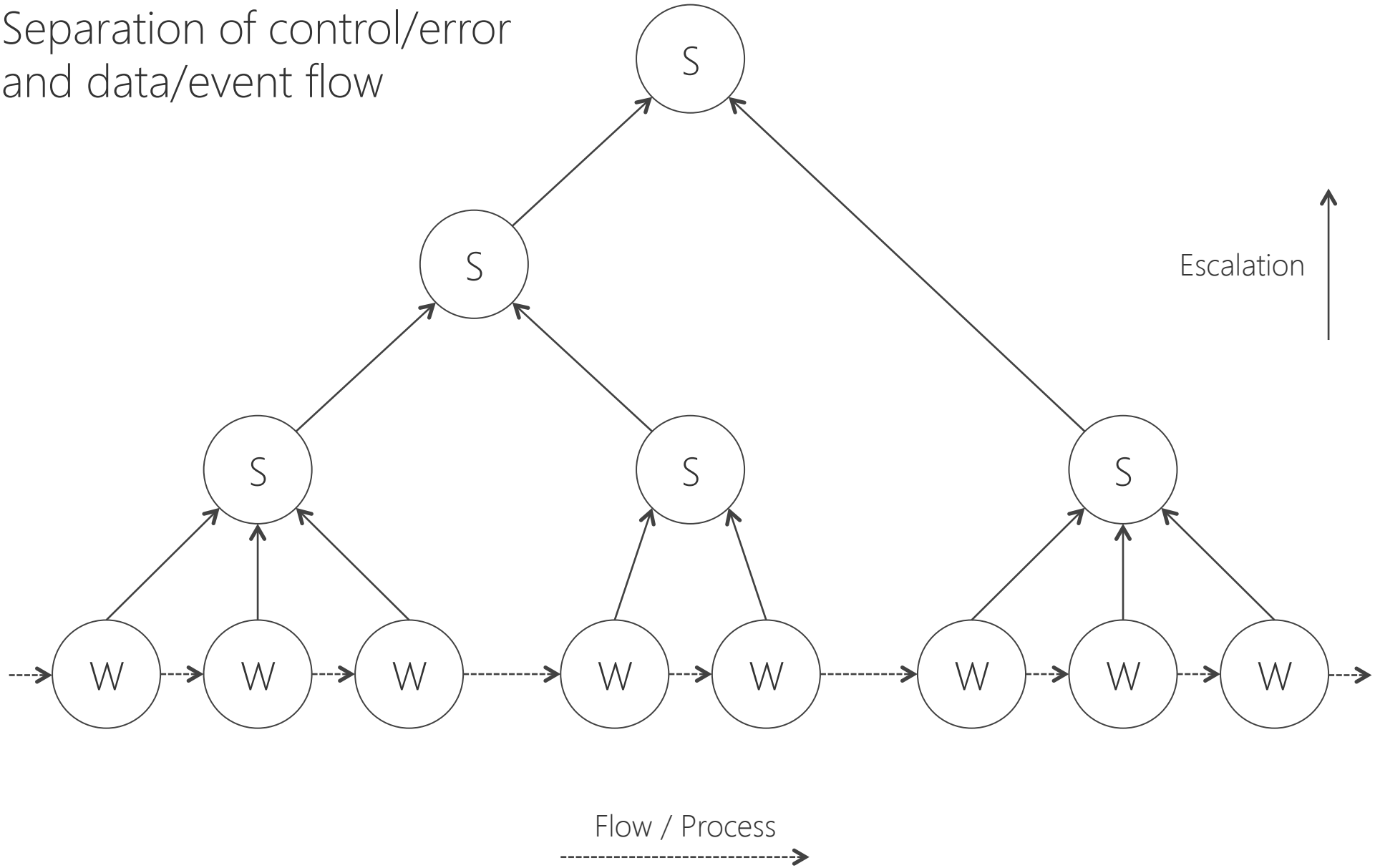
- without the user noticing it (best case)
- with a graceful degradation of service (worst case)

Resilience

- Resilient software design is mandatory
- Start with isolation and latency control
- Add automated error recovery and mitigation
- Separate control and data flow


Error flow
Control flow
Event/data flow
μS
Event/data flow
Isolation
Resource access

Separation of control/error and data/event flow



Home

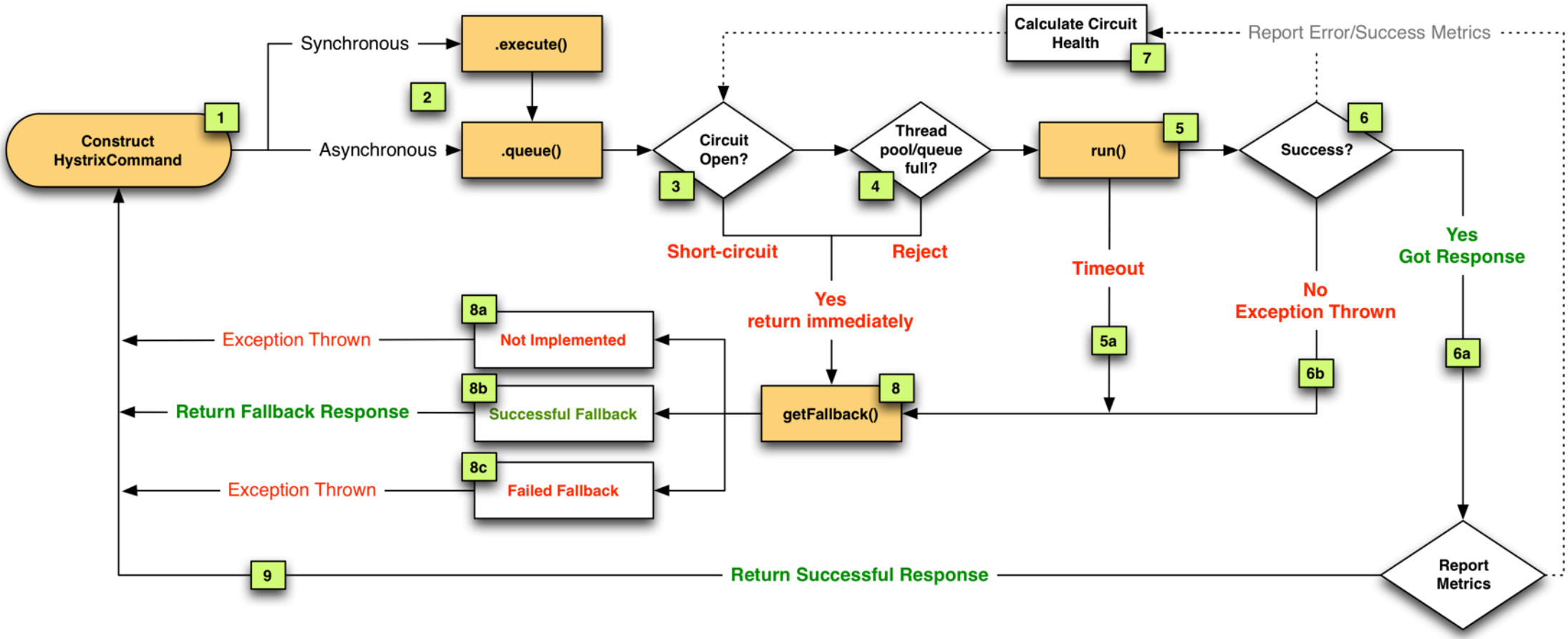
Page History Clone URL

What is Hystrix?

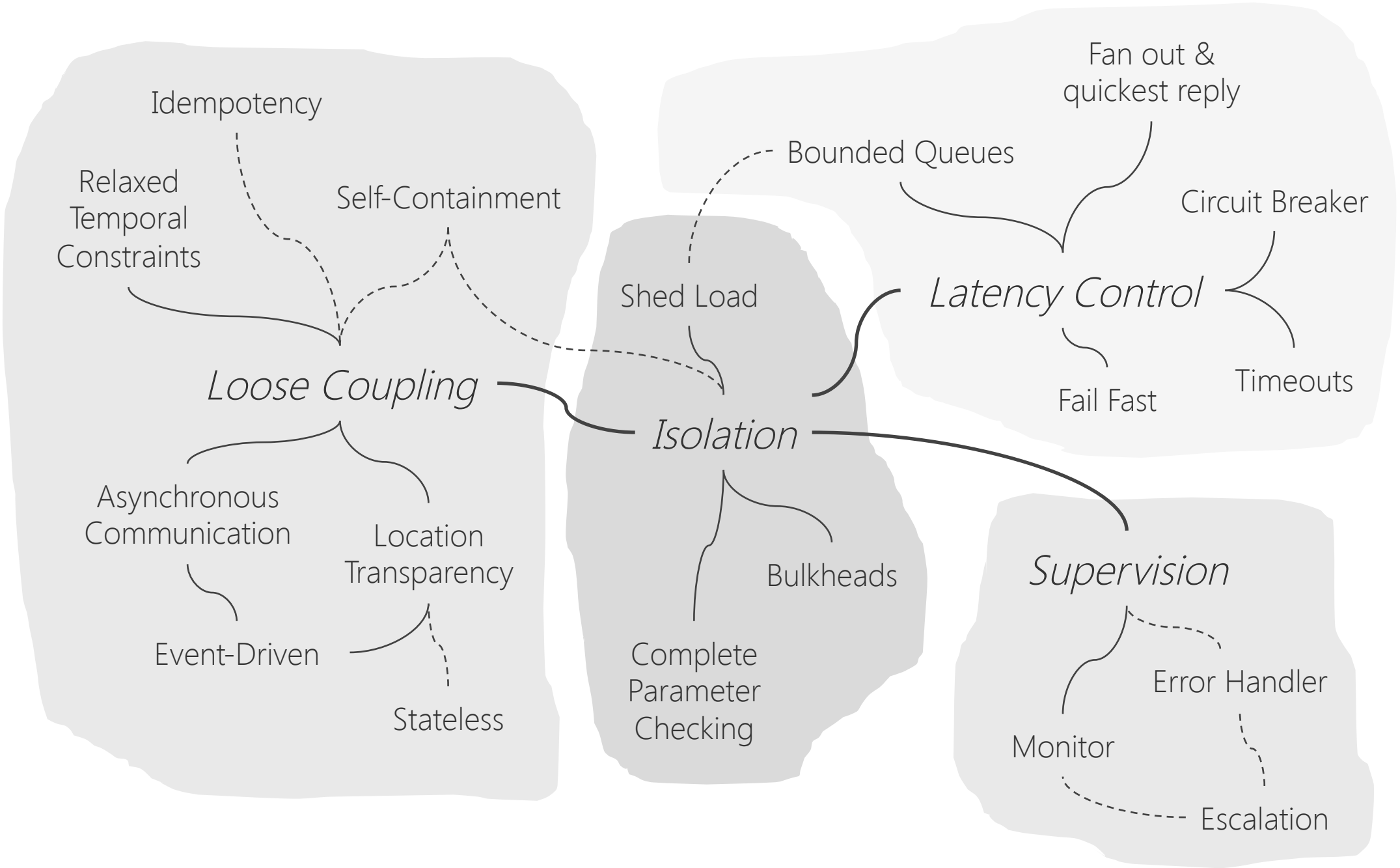
In a distributed environment, failure of any given service is inevitable. Hystrix is a library designed to control the interactions between these distributed services providing greater latency and fault tolerance. Hystrix does this by isolating points of access between the services, stopping cascading failures across them, and providing fallback options, all of which improve the system's overall resiliency.

Hystrix evolved out of resilience engineering work that the Netflix API team began in 2011. Over the course of 2012, Hystrix continued to evolve and mature, eventually leading to adoption

```
// Hystrix “Hello world”

public class HelloCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = ”Hello”; // Not important here
    private final String name;

    // Request parameters are passed in as constructor parameters
    public HelloCommand(String name) {
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));
        this.name = name;
    }

    @Override
    protected String run() throws Exception {
        // Usually here would be the resource call that needs to be guarded
        return "Hello, " + name;
    }
}


// Usage of a Hystrix command - synchronous variant
@Test
public void shouldGreetWorld() {
    String result = new HelloCommand("World").execute();
    assertEquals("Hello, World", result);
}
```



Source: https://github.com/Netflix/Hystrix/wiki/How-it-Works


Idempotency
Relaxed Temporal Constraints
Self-Containment
Loose Coupling
Asynchronous Communication
Event-Driven
Location Transparency
Stateless
Bounded Queues
Shed Load
Isolation
Complete Parameter Checking
Bulkheads
Fan out & quickest reply
Circuit Breaker
Latency Control
Fail Fast
Timeouts
Supervision
Error Handler
Monitor
Escalation

Wrap-up

- µServices are no free lunch
- Use if responsiveness is crucial
- Reduce stress by especially taking care of
  - Good functional design
  - Production readiness (incl. resilience)
- New challenges for developers (& ops)

@ufried

Uwe Friedrichsen | uwe.friedrichsen@codecentric.de | http://slideshare.net/ufried | http://ufried.tumblr.com