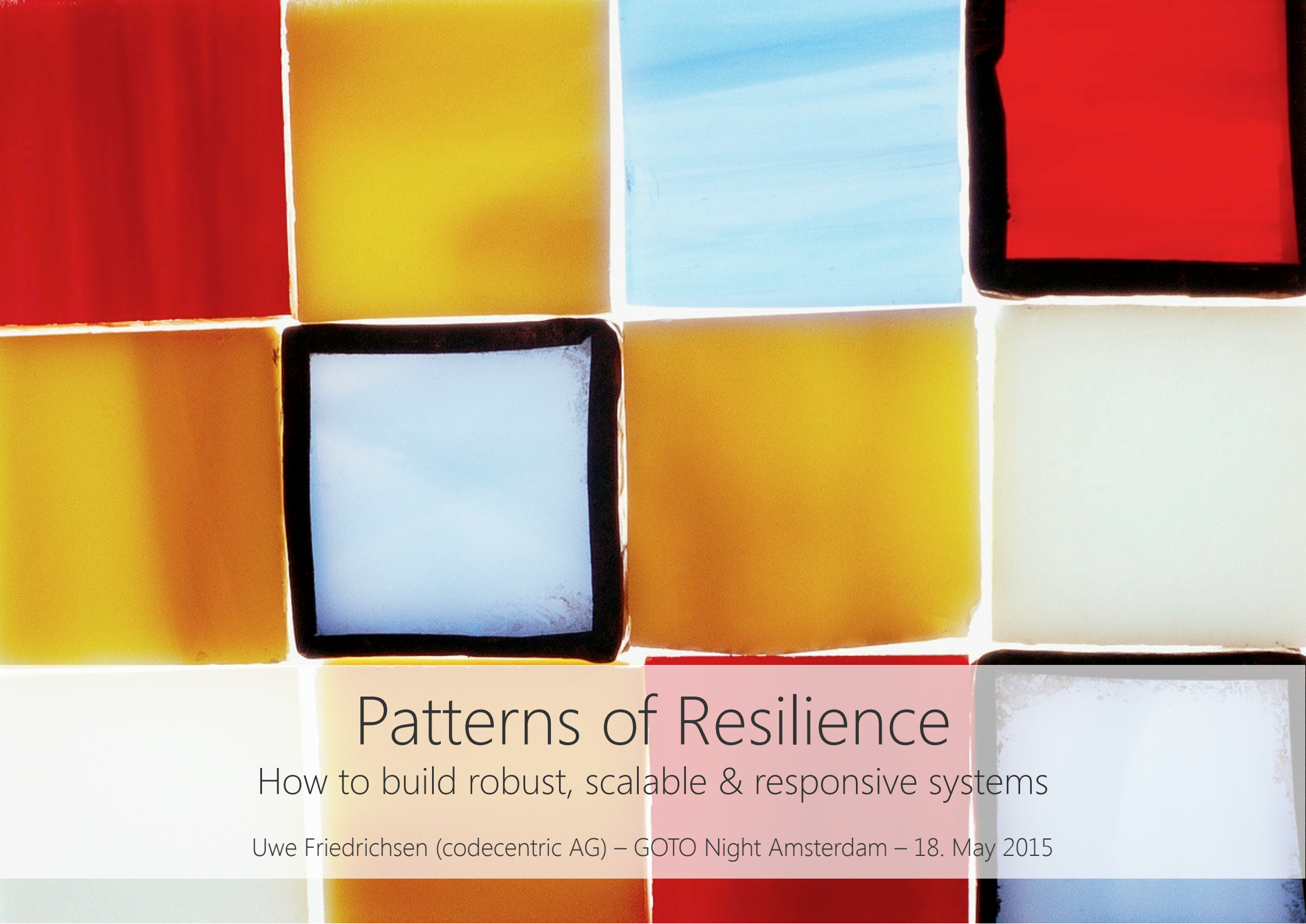


Patterns of Resilience

How to build robust, scalable & responsive systems

Uwe Friedrichsen (codecentric AG) – GOTO Night Amsterdam – 18. May 2015

@ufried

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


Resilience? Never heard of it …

re•sil•ience (rɪˈzɪl yəns) also **re•sil′ien•cy**, n.

1. the power or ability to return to the original form, position, etc., after being bent, compressed, or stretched; elasticity.
2. ability to recover readily from illness, depression, adversity, or the like; buoyancy.

http://www.thefreedictionary.com/resilience

What's all the fuss about?


It’s all about production!

Business

Production

Availability

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

MTTF: Mean Time To Failure

MTTR: Mean Time To Recovery

How can I maximize availability?

Traditional stability approach

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

Maximize MTTF

Underlying assumption

reliability

degree to which a system, product or component performs specified functions under specified conditions for a specified period of time

ISO/IEC 25010:2011(en)

https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en

What's the problem?

(Almost) every system is a distributed system

Chas Emerick

The Eight Fallacies of Distributed Computing

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

Peter Deutsch

https://blogs.oracle.com/jag/resource/Fallacies.html

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

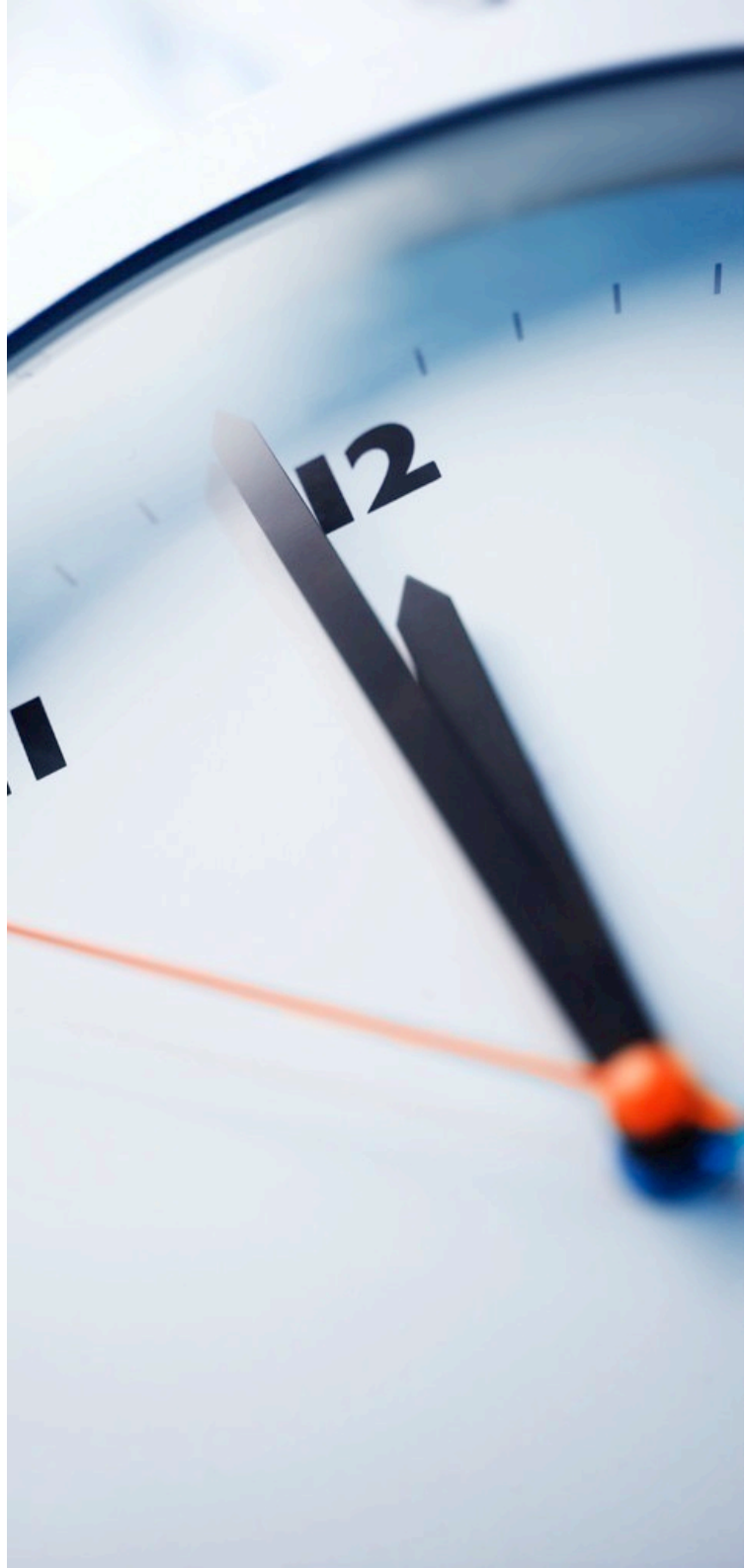
Failures in todays complex, distributed and interconnected systems are not the exception.

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

… and it's getting "worse"

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

→ *Ever-increasing complexity and connectivity*



Do not try to avoid failures. Embrace them.

Resilience approach

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

Minimize MTTR

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)

Designing for resilience

A small pattern language

Isolation

Isolation

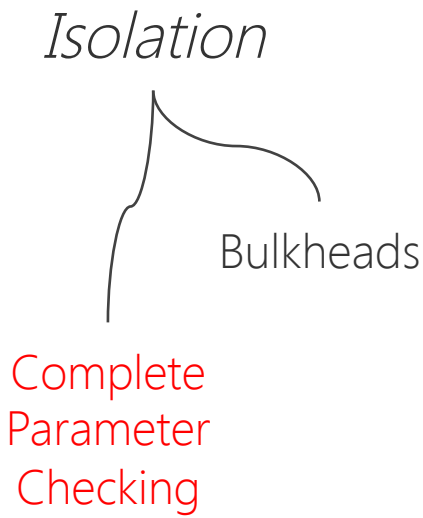
- System must not fail as a whole
- Split system in parts and isolate parts against each other
- Avoid cascading failures
- Requires set of measures to implement

Isolation

Bulkheads

Bulkheads

- Core isolation pattern
- a.k.a. “failure units” or “units of mitigation”
- Used as units of redundancy
- Pure design issue

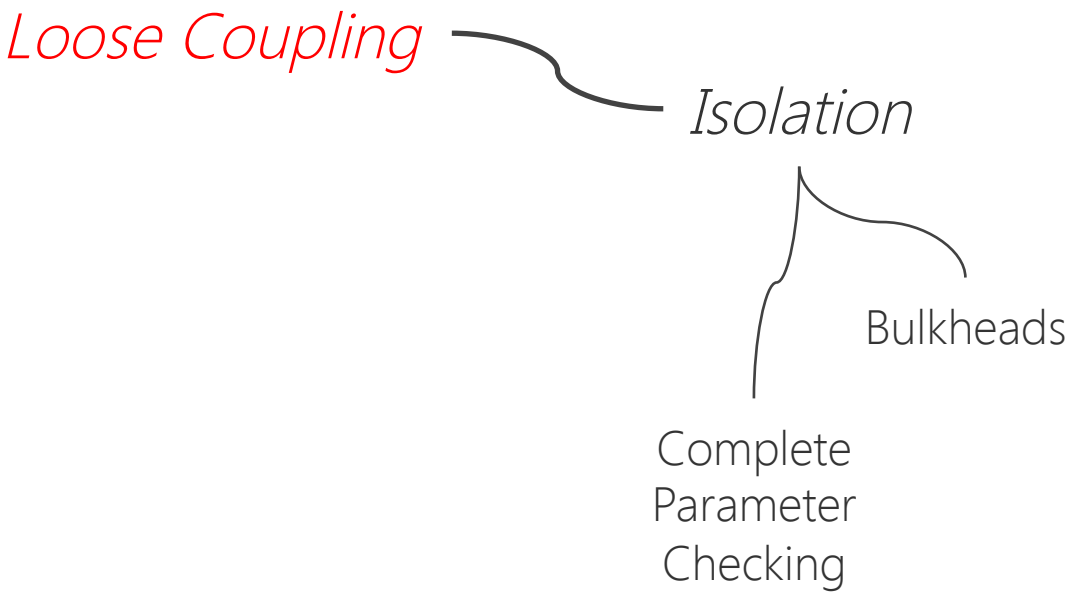

Isolation
Bulkheads
Complete
Parameter
Checking

Complete Parameter Checking

- As obvious as it sounds, yet often neglected
- Protection from broken/malicious calls (and return values)
- Pay attention to Postel's law
- Consider specific data types

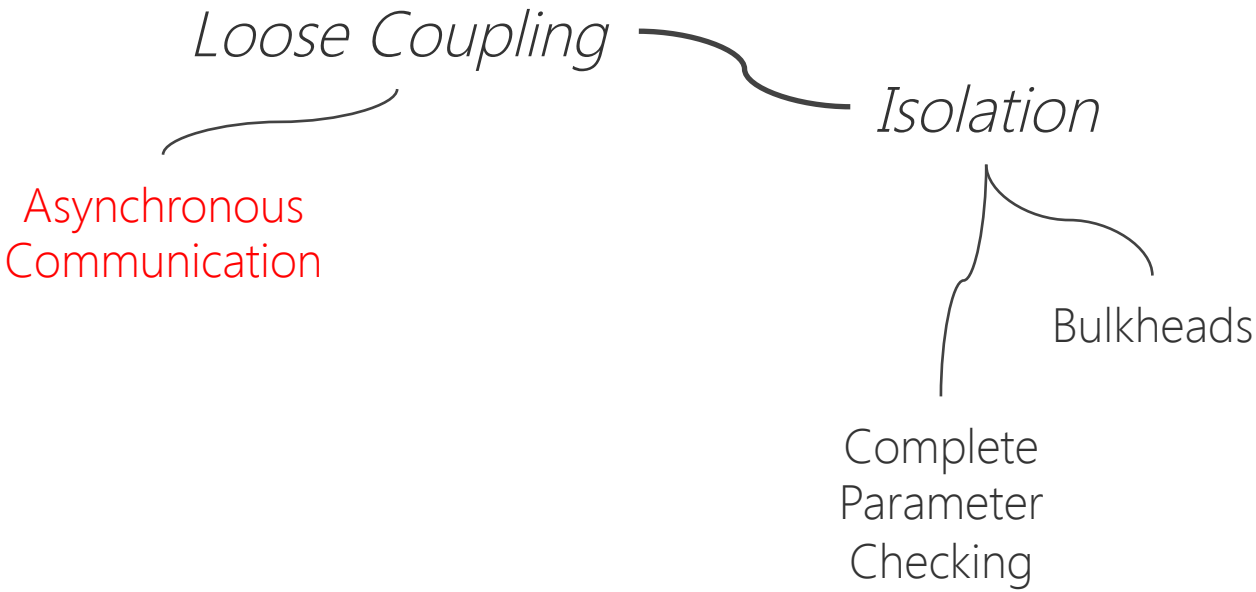
Complete Parameter Checking

```
// How to design request parameters

// Worst variant - requires tons of checks
String buySomething(Map<String, String> params);


// Still a bad variant - still a lot of checks required
String buySomething(String customerId, String productId, int count);


// Much better - only null checks required
PurchaseStatus buySomething(Customer buyer, Article product, Quantity count);
```


Loose Coupling
Isolation
Bulkheads
Complete
Parameter
Checking

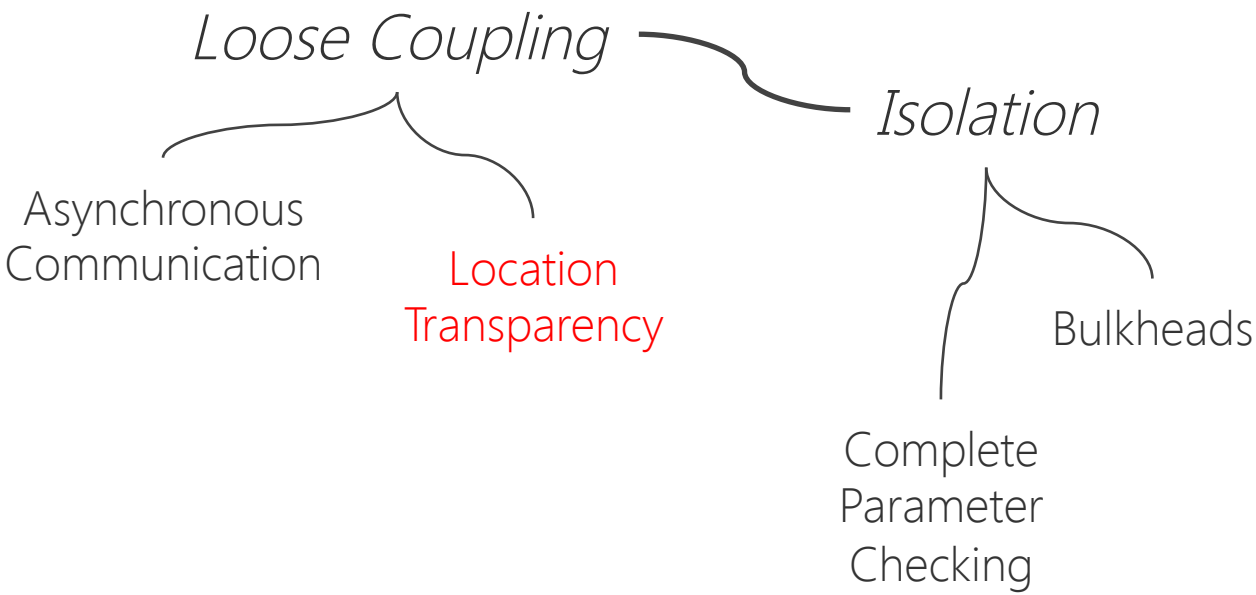
Loose Coupling

- Complements isolation
- Reduce coupling between failure units
- Avoid cascading failures
- Different approaches and patterns available


Loose Coupling
Asynchronous Communication
Isolation
Bulkheads
Complete Parameter Checking

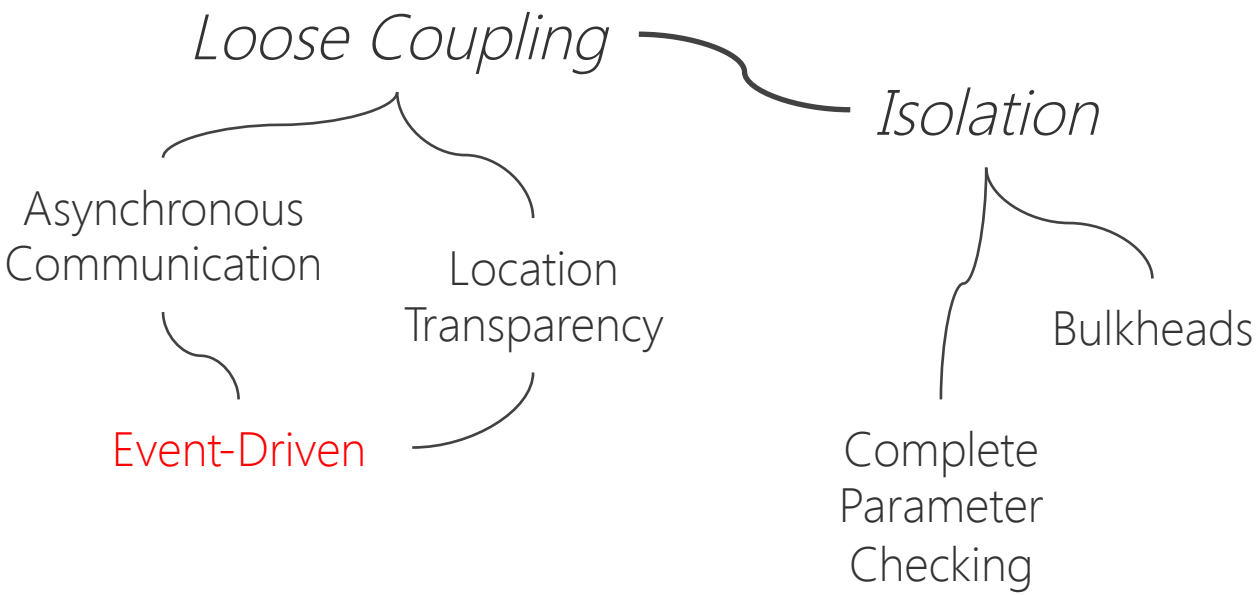
Asynchronous Communication

- Decouples sender from receiver
- Sender does not need to wait for receiver's response
- Useful to prevent cascading failures due to failing/latent resources
- Breaks up the call stack paradigm


Loose Coupling
Isolation
Asynchronous Communication
Location Transparency
Bulkheads
Complete Parameter Checking

Location Transparency

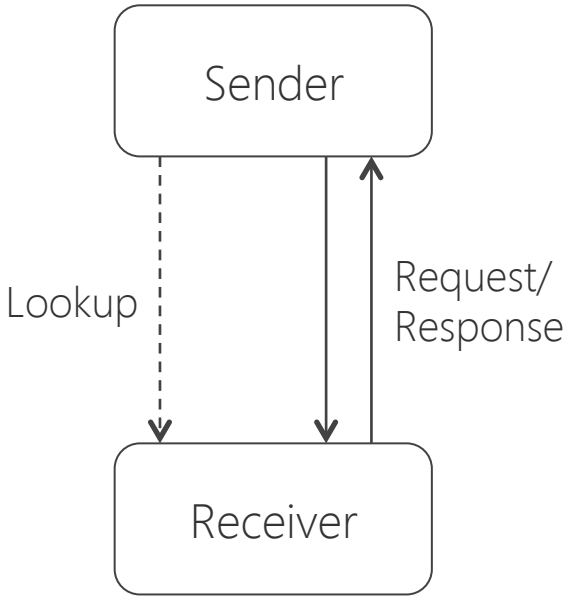
- Decouples sender from receiver
- Sender does not need to know receiver's concrete location
- Useful to implement redundancy and failover transparently
- Usually implemented using load balancers or middleware


Loose Coupling
Isolation
Asynchronous Communication
Location Transparency
Event-Driven
Bulkheads
Complete Parameter Checking

Event-Driven

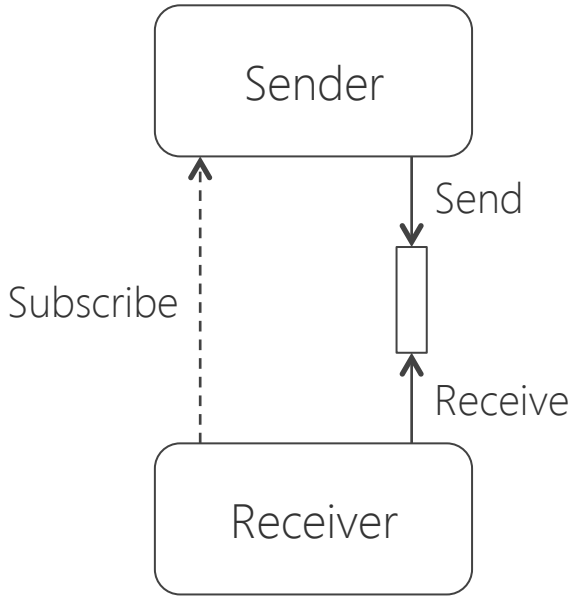
- Popular asynchronous communication style
- Without broker location dependency is reversed
- With broker location transparency is easily achieved
- Very different from request-response paradigm

Request/response
(Sender depends on receiver)



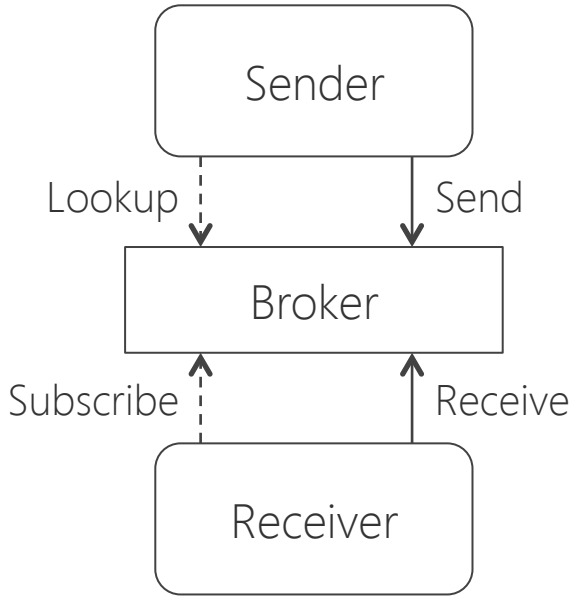

```
// from sender
receiver = lookup()

// from sender
result =
  receiver.call()
```

Event-driven without broker
(Receiver depends on sender)

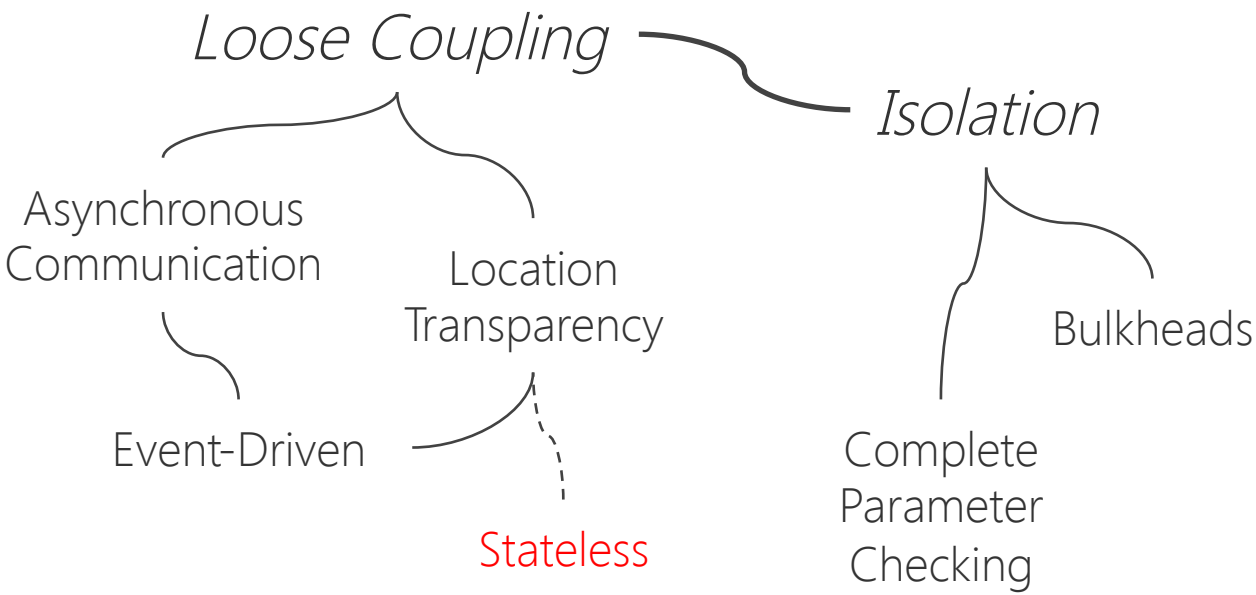



```
// from sender
queue.send(msg)

// from receiver
queue =
  sender.subscribe()
msg = queue.receive()
```

Event-driven with broker
(Sender and receiver decoupled)

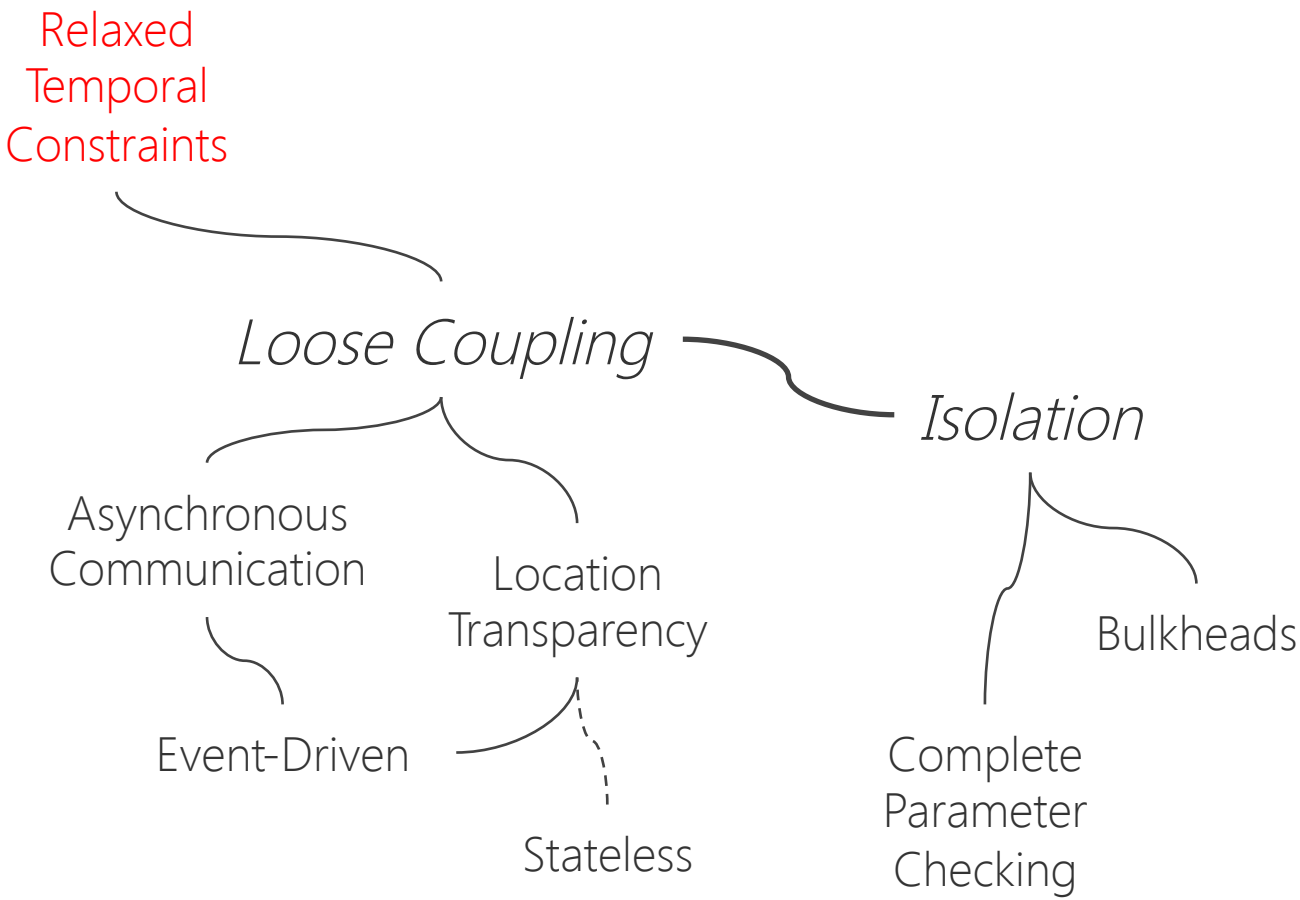



```
// from sender
broker = lookup()
broker.send(msg)

// from receiver
queue =
  broker.subscribe()
msg = queue.receive()
```


Loose Coupling
Isolation
Asynchronous Communication
Location Transparency
Bulkheads
Event-Driven
Stateless
Complete Parameter Checking

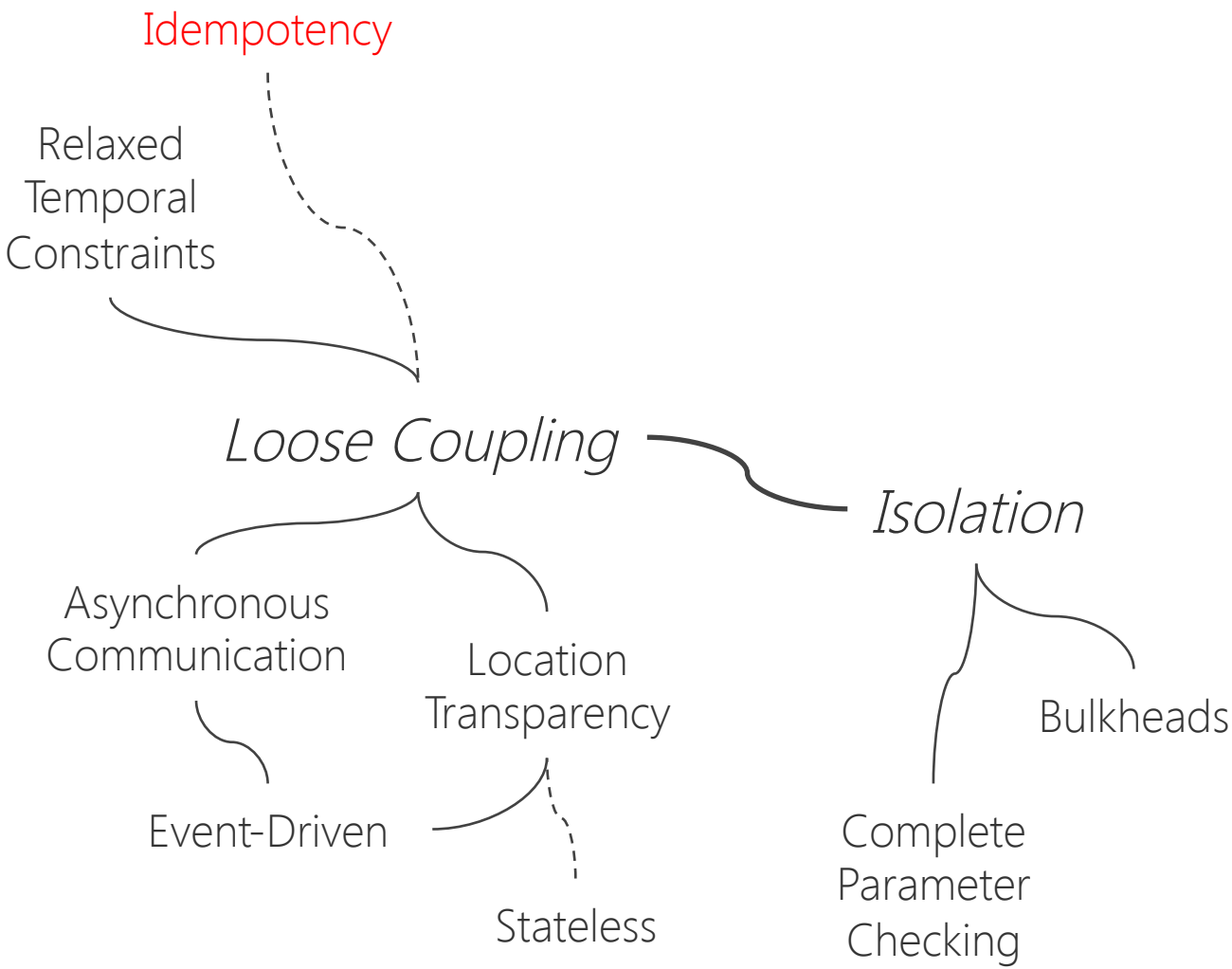
Stateless

- Supports location transparency (amongst other patterns)
- Service relocation is hard with state
- Service failover is hard with state
- Very fundamental resilience and scalability pattern


Relaxed
Temporal
Constraints
Loose Coupling
Isolation
Asynchronous
Communication
Location
Transparency
Bulkheads
Event-Driven
Stateless
Complete
Parameter
Checking

Relaxed Temporal Constraints

- Strict consistency requires tight coupling of the involved nodes
- Any single failure immediately compromises availability
- Use a more relaxed consistency model to reduce coupling
- The real world is not ACID, it is BASE!


Idempotency
Relaxed
Temporal
Constraints
Loose Coupling
Isolation
Asynchronous
Communication
Location
Transparency
Bulkheads
Event-Driven
Stateless
Complete
Parameter
Checking

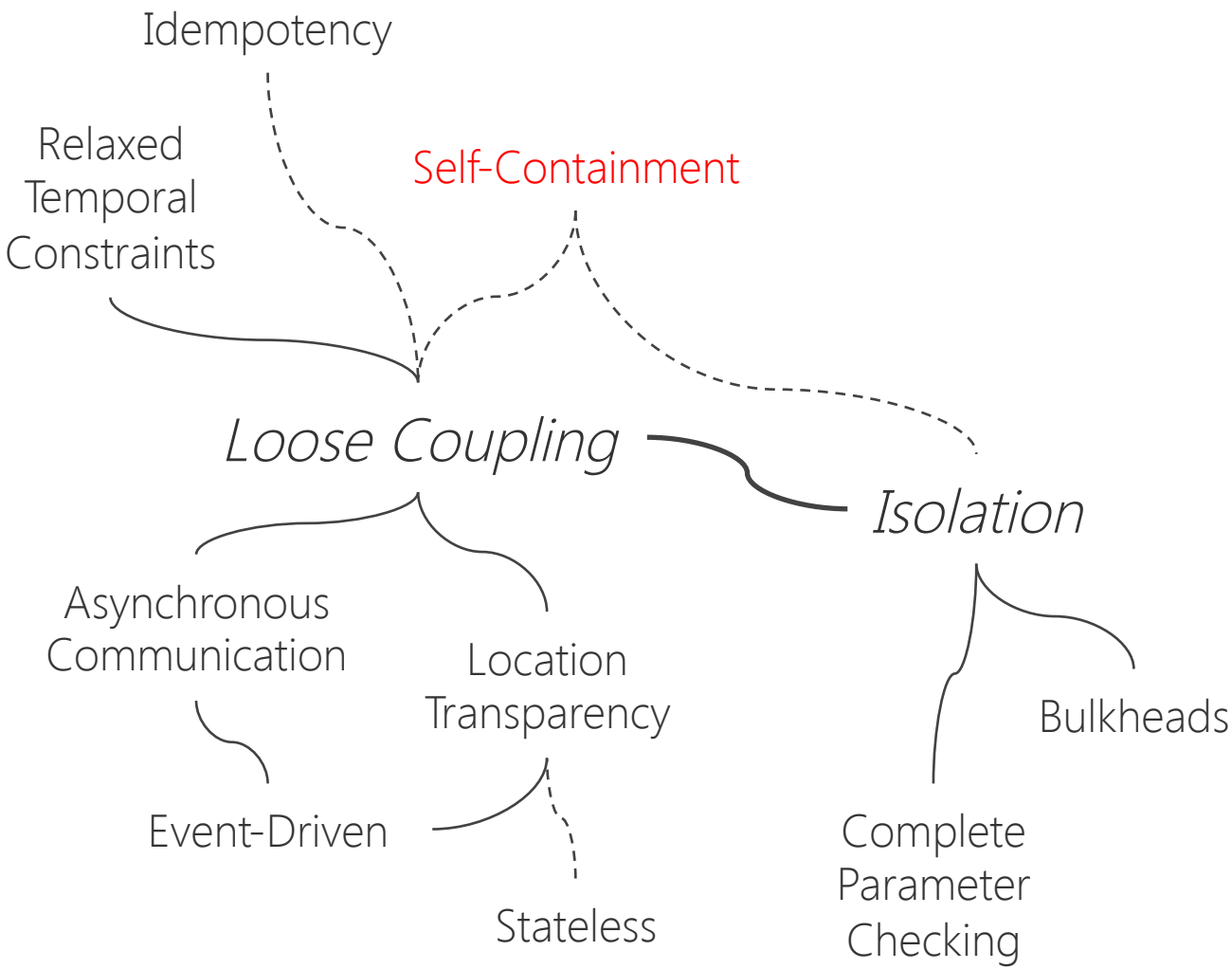
Idempotency

- Non-idempotency is complicated to handle in distributed systems
- (Usually) increases coupling between participating parties
- Use idempotent actions to reduce coupling between nodes
- Very fundamental resilience and scalability pattern

Unique request token (schematic)

```
// Client/Sender part

// Create request with unique request token (e.g., via UUID)
token = createUniqueToken()
request = createRequest(token, payload)

// Send request until successful
while (!successful)
  send(request, timeout) // Do not forget failure handling
```

```
// Server/Receiver part

// Receive request
request = receive()

// Process request only if token is unknown
if (!lookup(request.token)) // needs to implemented in a CAS way to be safe
  process(request)
  store(token) // Store token for lookup (can be garbage collected eventually)
```


Idempotency
Relaxed
Temporal
Constraints
Self-Containment
Loose Coupling
Isolation
Asynchronous
Communication
Location
Transparency
Bulkheads
Event-Driven
Stateless
Complete
Parameter
Checking

Self-Containment

- Services are self-contained deployment units
- No dependencies to other runtime infrastructure components
- Reduces coupling at deployment time
- Improves isolation and flexibility

Use a framework ...

Spring Boot

fabric8

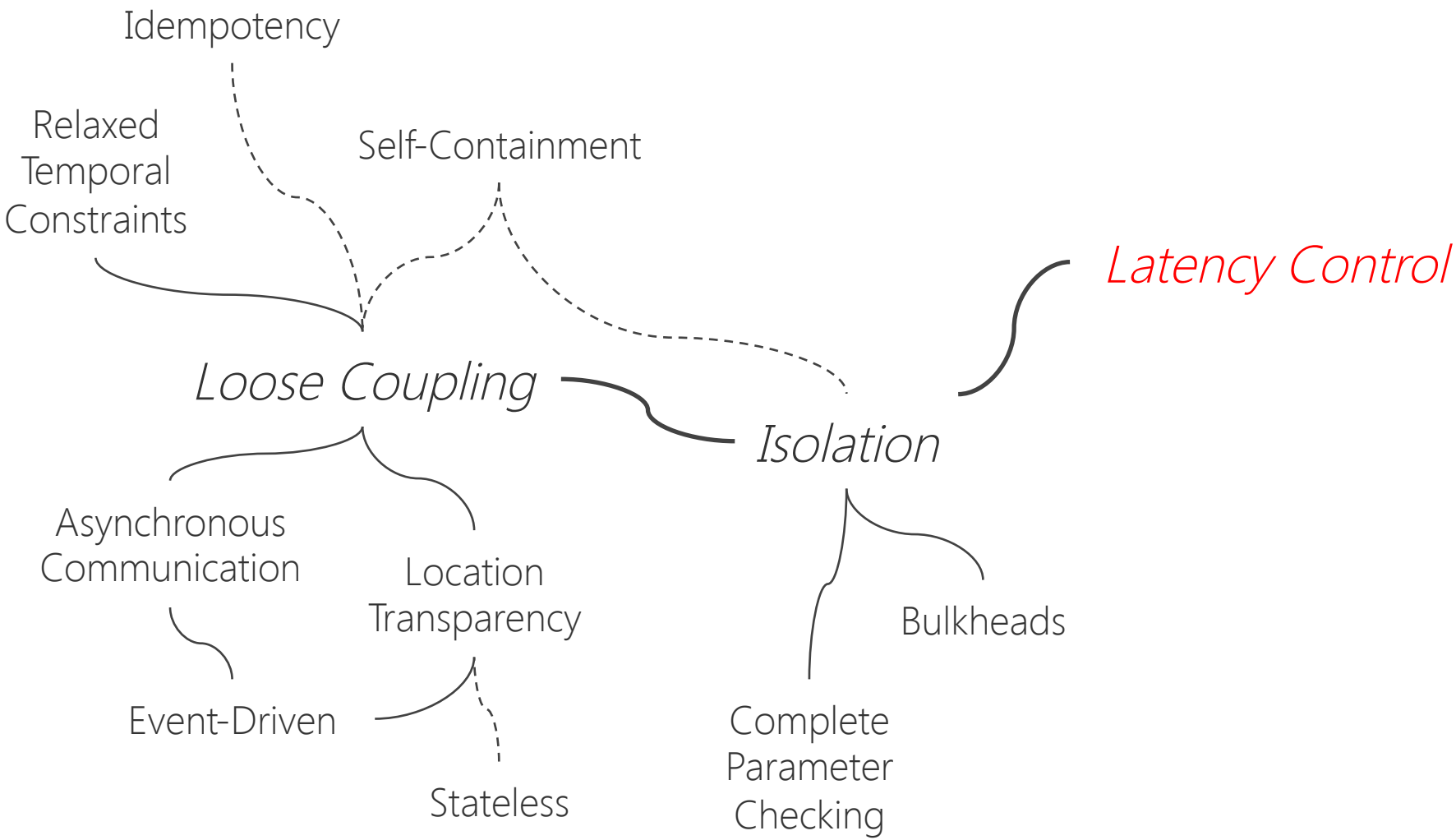
Dropwizard

...

Jackson

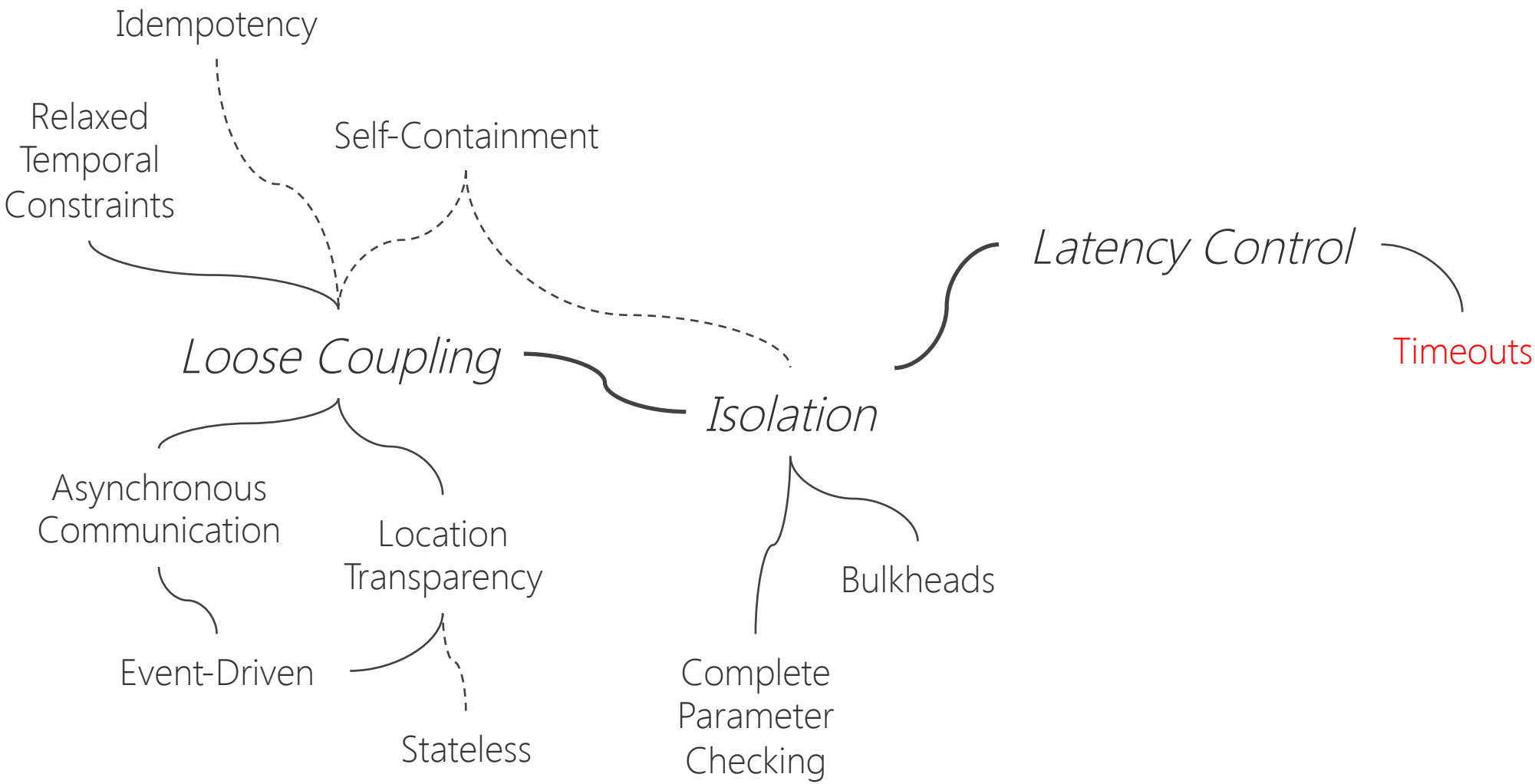
Metrics

... or do it yourself


Idempotency
Relaxed Temporal Constraints
Self-Containment
Latency Control
Loose Coupling
Isolation
Asynchronous Communication
Location Transparency
Bulkheads
Event-Driven
Stateless
Complete Parameter Checking

Latency control

- Complements isolation
- Detection and handling of non-timely responses
- Avoid cascading temporal failures
- Different approaches and patterns available

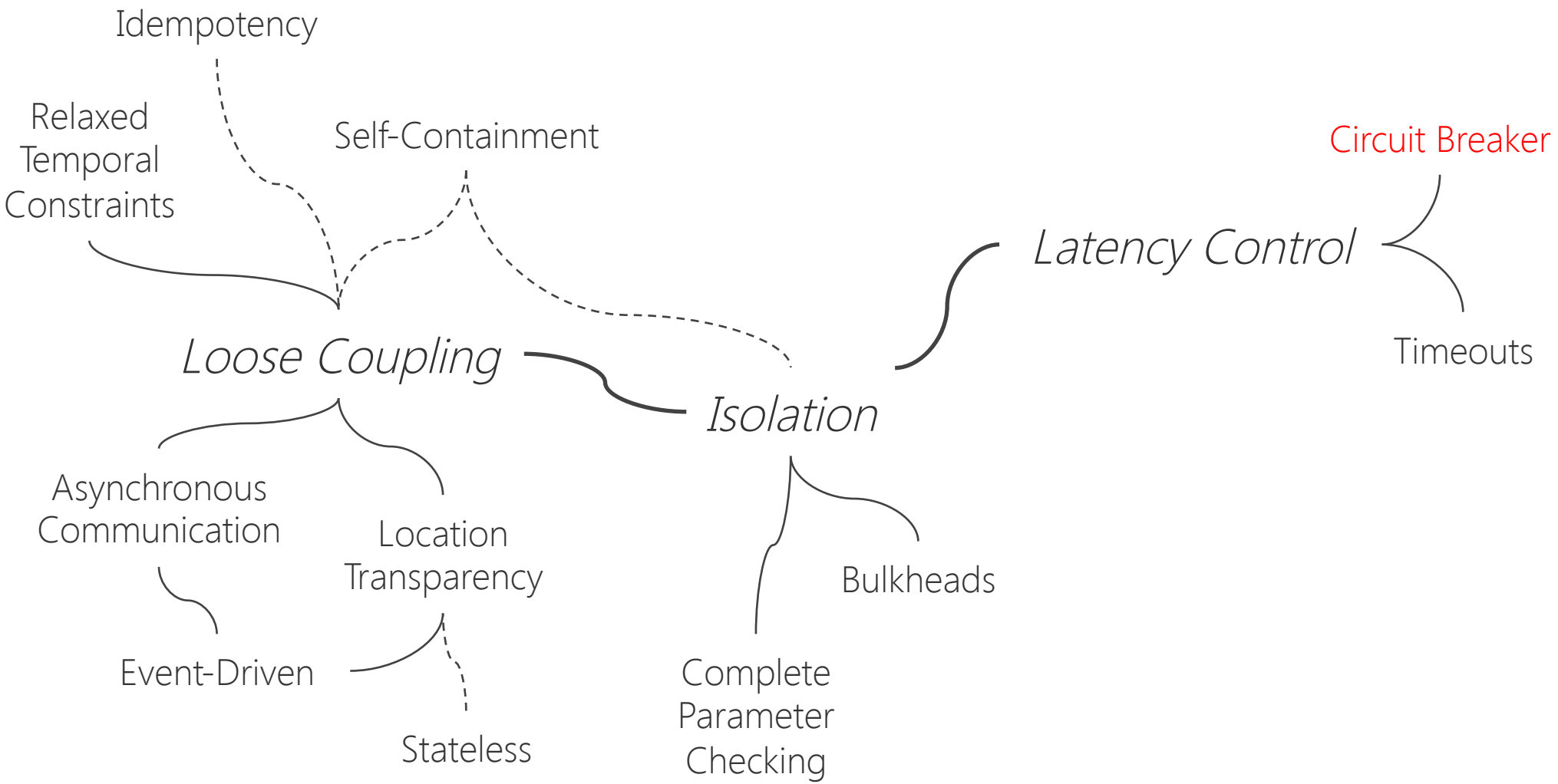

Idempotency
Relaxed Temporal Constraints
Self-Containment
Latency Control
Timeouts
Loose Coupling
Isolation
Asynchronous Communication
Location Transparency
Bulkheads
Event-Driven
Stateless
Complete Parameter Checking

Timeouts

- Preserve responsiveness independent of downstream latency
- Measure response time of downstream calls
- Stop waiting after a pre-determined timeout
- Take alternate action if timeout was reached

Timeouts with standard library means

```
// Wrap blocking action in a Callable
Callable<MyActionResult> myAction = <My Blocking Action>

// Use a simple ExecutorService to run the action in its own thread
ExecutorService executor = Executors.newSingleThreadExecutor();
Future<MyActionResult> future = executor.submit(myAction);
MyActionResult result = null;

// Use Future.get() method to limit time to wait for completion
try {
  result = future.get(TIMEOUT, TIMEUNIT);
  // Action completed in a timely manner – process results
} catch (TimeoutException e) {
    // Handle timeout (e.g., schedule retry, escalate, alternate action, …)
} catch (...) {
    // Handle other exceptions that can be thrown by Future.get()
} finally {
    // Make sure the callable is stopped even in case of a timeout
    future.cancel(true);
}
```


Idempotency
Relaxed Temporal Constraints
Self-Containment
Circuit Breaker
Latency Control
Timeouts
Loose Coupling
Isolation
Asynchronous Communication
Location Transparency
Bulkheads
Event-Driven
Stateless
Complete Parameter Checking

Circuit Breaker

- Probably most often cited resilience pattern
- Extension of the timeout pattern
- Takes downstream unit offline if calls fail multiple times
- Specific variant of the fail fast pattern

Home

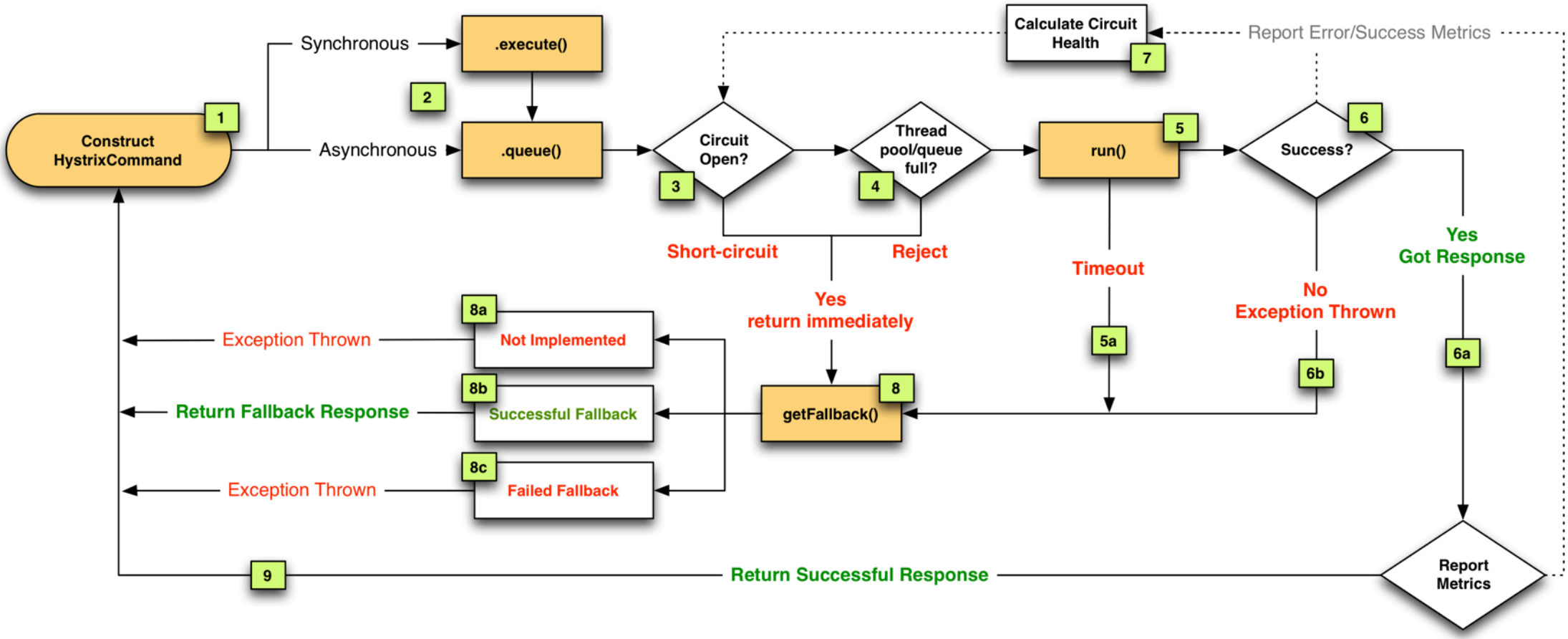
Page History Clone URL

What is Hystrix?

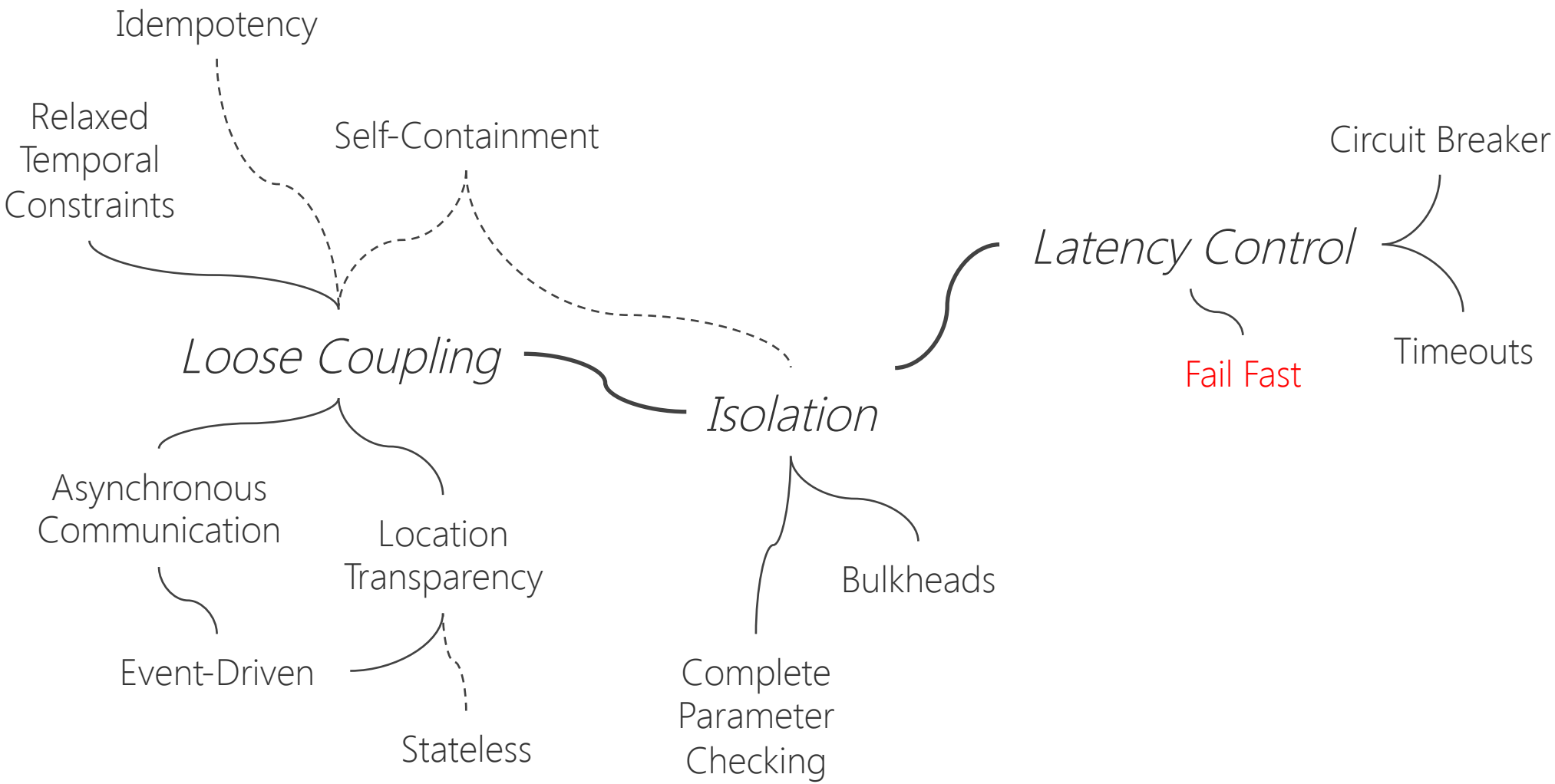
In a distributed environment, failure of any given service is inevitable. Hystrix is a library designed to control the interactions between these distributed services providing greater latency and fault tolerance. Hystrix does this by isolating points of access between the services, stopping cascading 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);
}
```

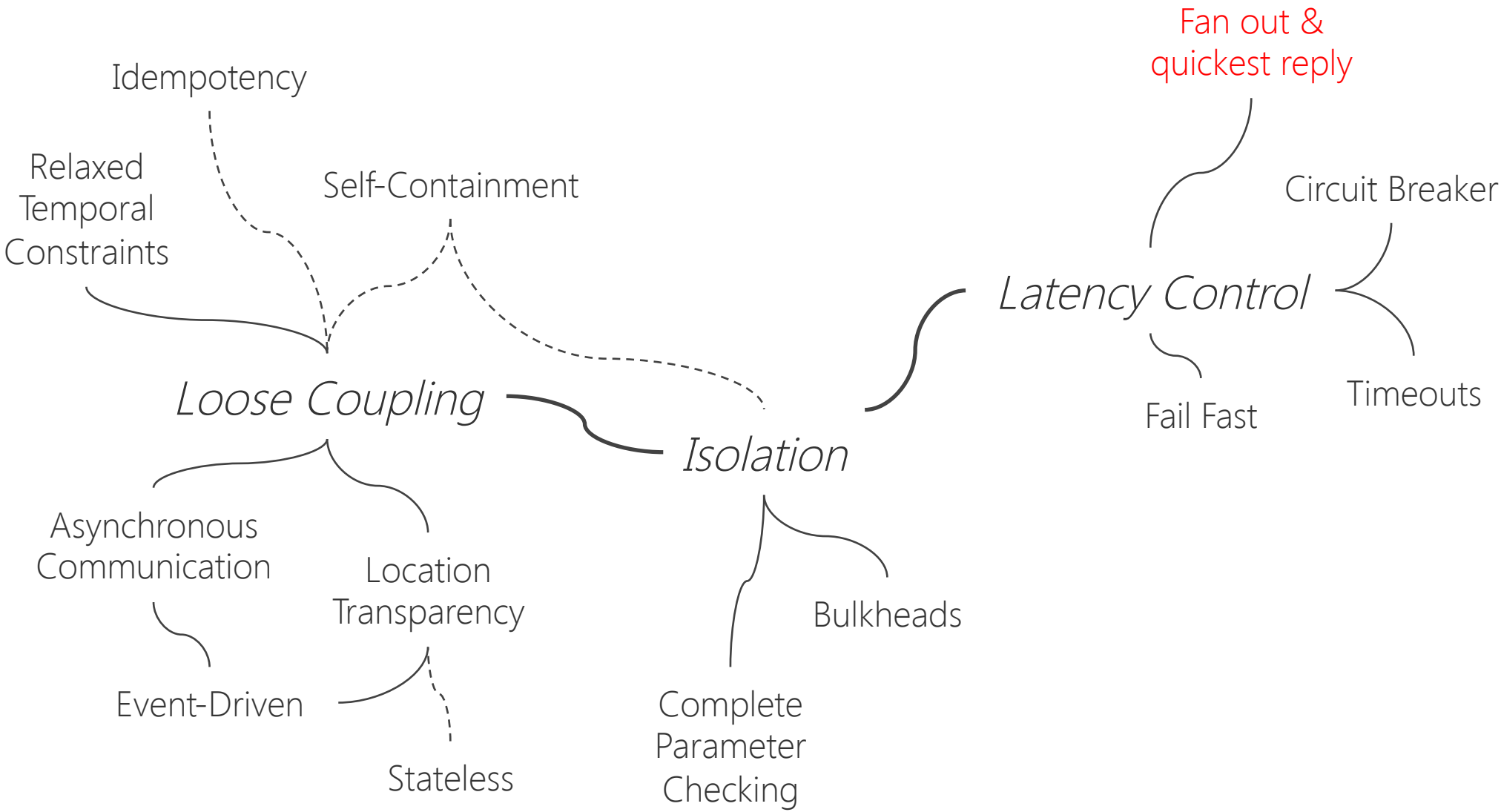


Source: https://github.com/Netflix/Hystrix/wiki/How-it-Works


Idempotency
Relaxed Temporal Constraints
Self-Containment
Circuit Breaker
Latency Control
Fail Fast
Timeouts
Loose Coupling
Isolation
Asynchronous Communication
Location Transparency
Bulkheads
Event-Driven
Stateless
Complete Parameter Checking

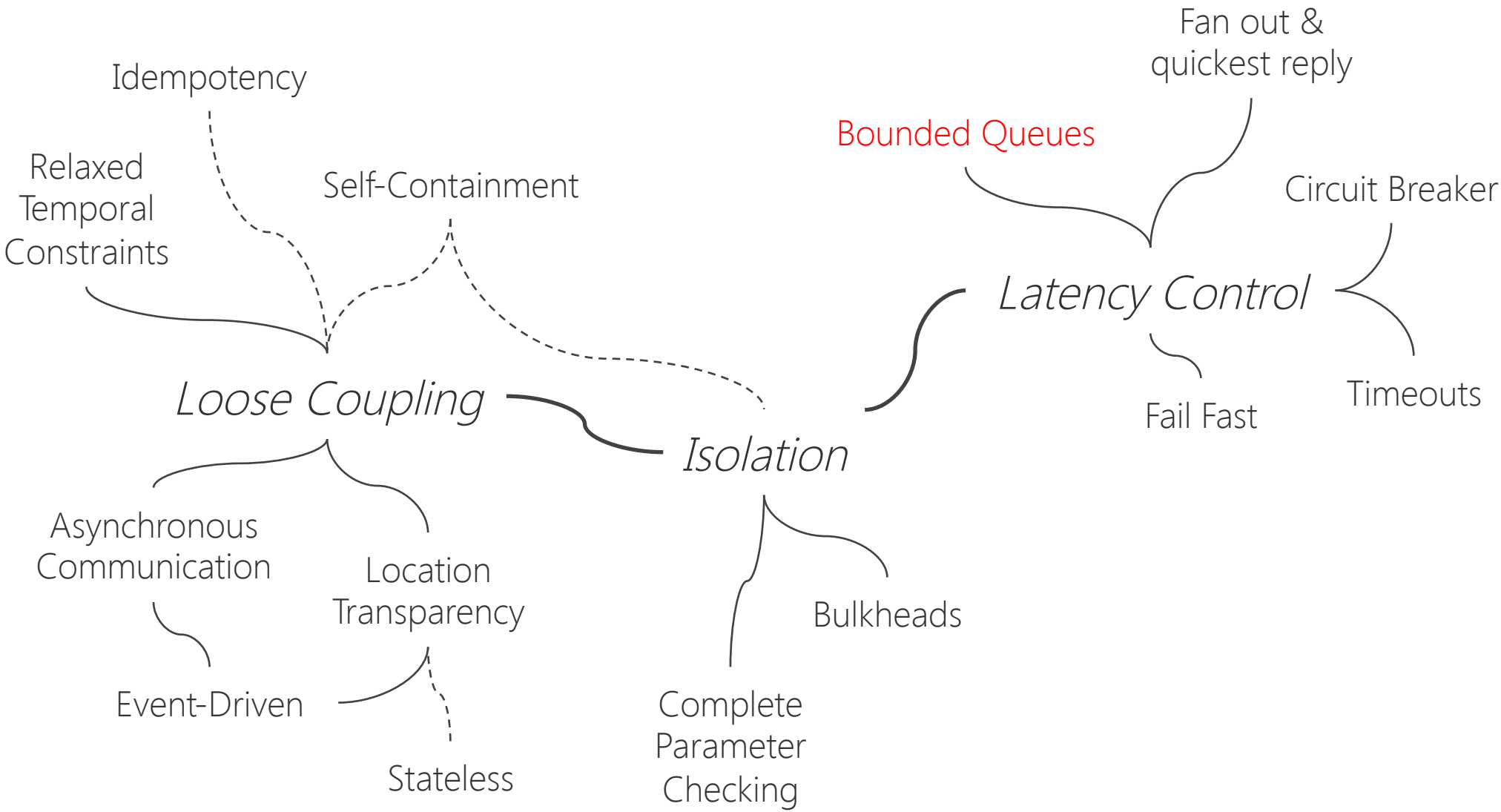
Fail Fast

- “If you know you’re going to fail, you better fail fast”
- Avoid foreseeable failures
- Usually implemented by adding checks in front of costly actions
- Enhances probability of not failing


Idempotency
Relaxed Temporal Constraints
Self-Containment
Loose Coupling
Asynchronous Communication
Event-Driven
Location Transparency
Stateless
Isolation
Complete Parameter Checking
Bulkheads
Latency Control
Fan out & quickest reply
Fail Fast
Circuit Breaker
Timeouts

Fan out & quickest reply

- Send request to multiple workers
- Use quickest reply and discard all other responses
- Reduces probability of latent responses
- Tradeoff is “waste” of resources

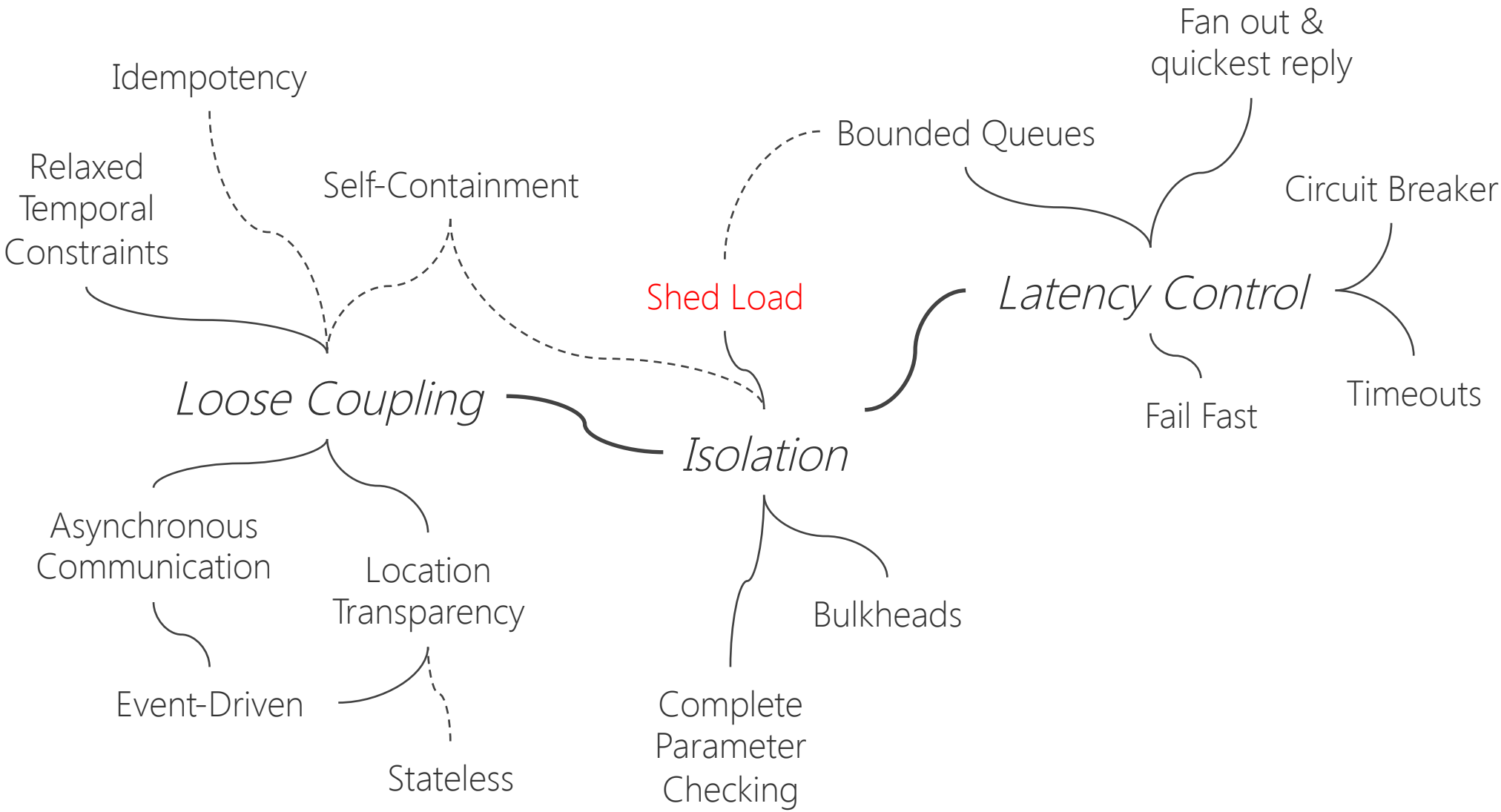

Idempotency
Relaxed Temporal Constraints
Self-Containment
Loose Coupling
Asynchronous Communication
Event-Driven
Location Transparency
Stateless
Isolation
Complete Parameter Checking
Bulkheads
Bounded Queues
Fan out & quickest reply
Latency Control
Circuit Breaker
Fail Fast
Timeouts

Bounded Queues

- Limit request queue sizes in front of highly utilized resources
- Avoids latency due to overloaded resources
- Introduces pushback on the callers
- Another variant of the fail fast pattern

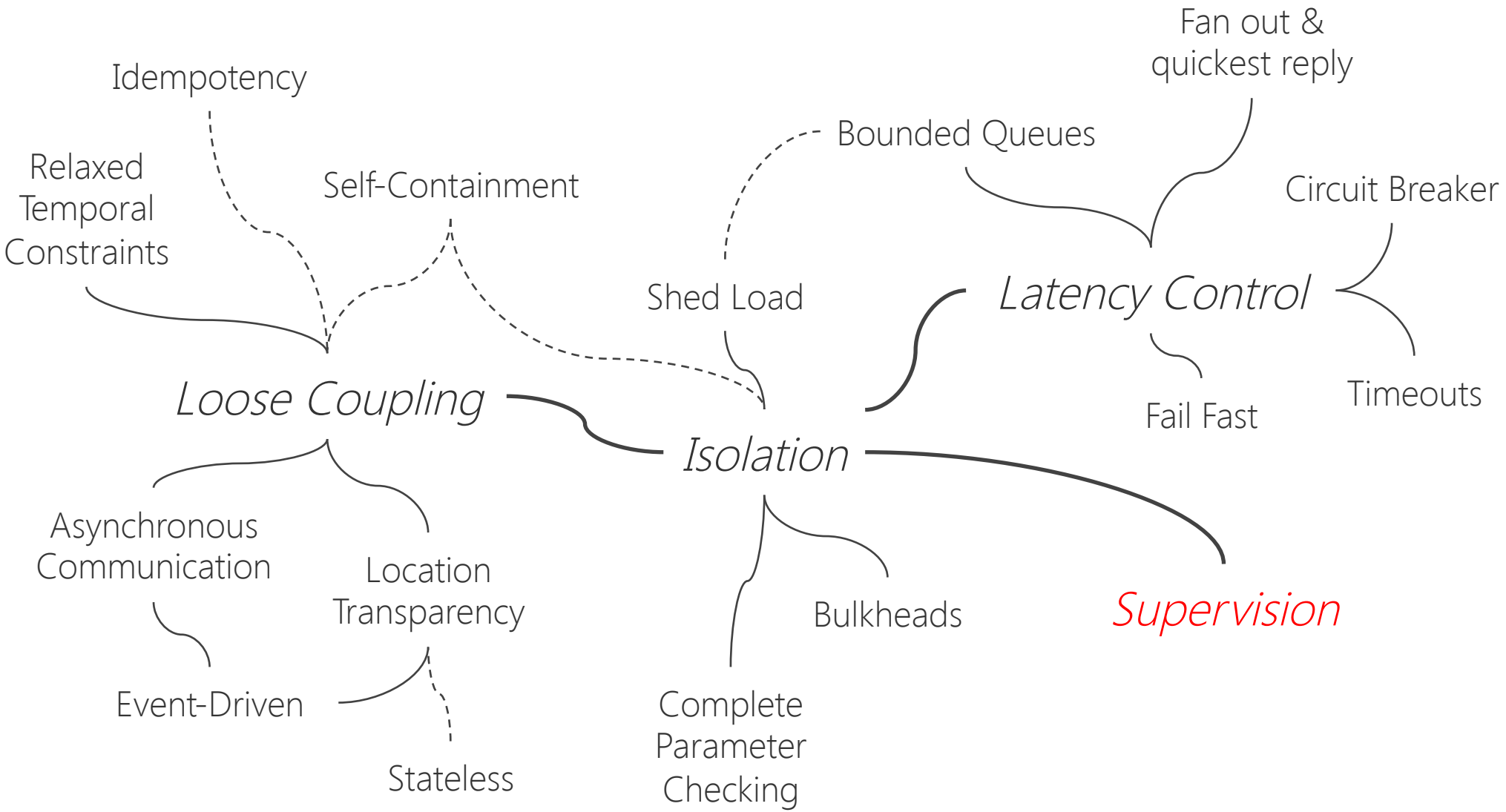
Bounded Queue Example

```
// Executor service runs with up to 6 worker threads simultaneously
// When thread pool is exhausted, up to 4 tasks will be queued -
// additional tasks are rejected triggering the PushbackHandler
final int POOL_SIZE = 6;
final int QUEUE_SIZE = 4;

// Set up a thread pool executor with a bounded queue and a PushbackHandler
ExecutorService executor =
  new ThreadPoolExecutor(POOL_SIZE, POOL_SIZE, // Core pool size, max pool size
                         0, TimeUnit.SECONDS,  // Timeout for unused threads
                         new ArrayBlockingQueue(QUEUE_SIZE),
                         new PushbackHandler);


// PushbackHandler - implements the desired pushback behavior
public class PushbackHandler implements RejectedExecutionHandler {
  @Override
  public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
    // Implement your pushback behavior here
  }
}
```


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

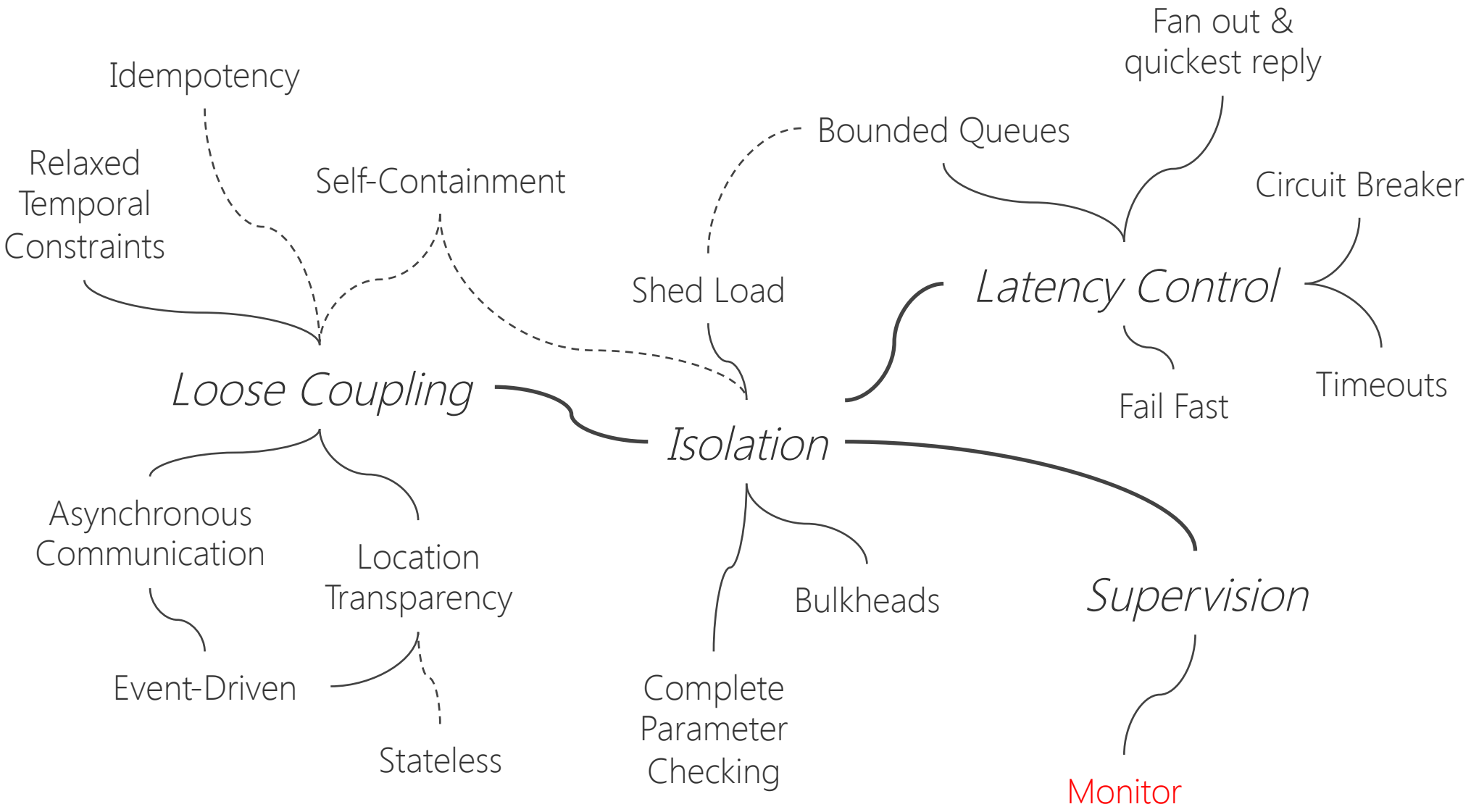
Shed Load

- Upstream isolation pattern
- Avoid becoming overloaded due to too many requests
- Install a gatekeeper in front of the resource
- Shed requests based on resource load


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

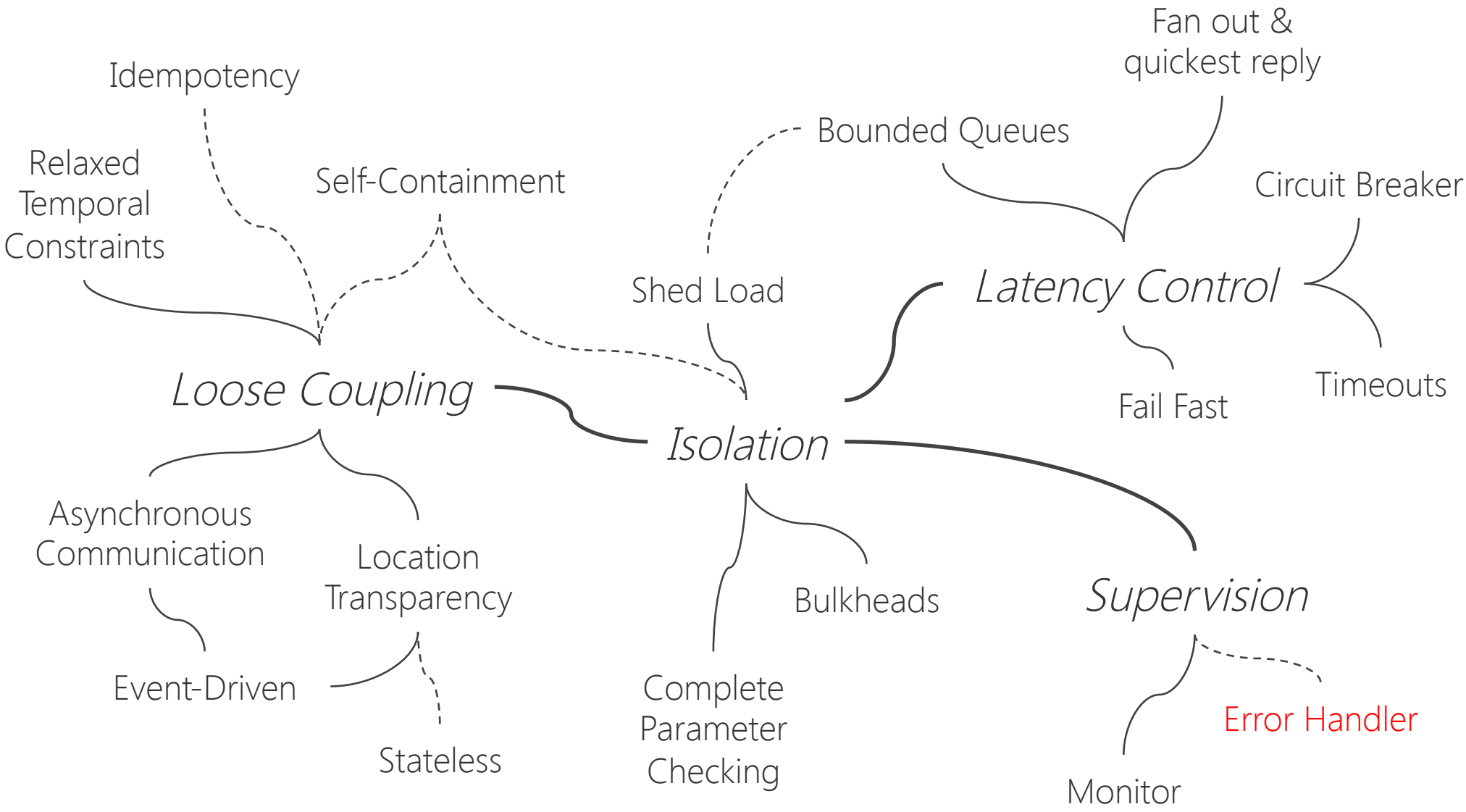
Supervision

- Provides failure handling beyond the means of a single failure unit
- Detect unit failures
- Provide means for error escalation
- Different approaches and patterns available


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

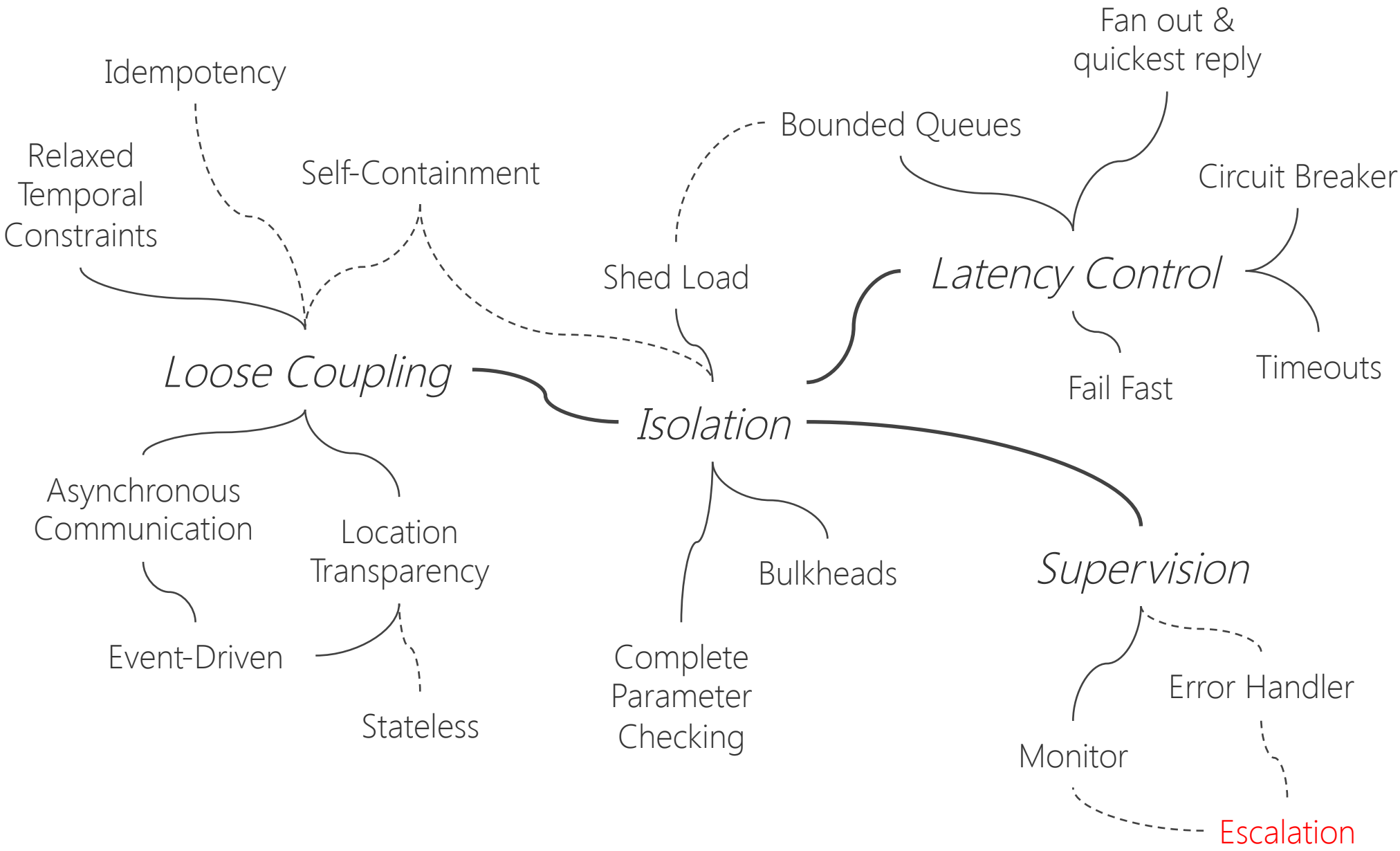
Monitor

- Observe unit behavior and interactions from the outside
- Automatically respond to detected failures
- Part of the system – complex failure handling strategies possible
- Outside the system – more robust against system level failures


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

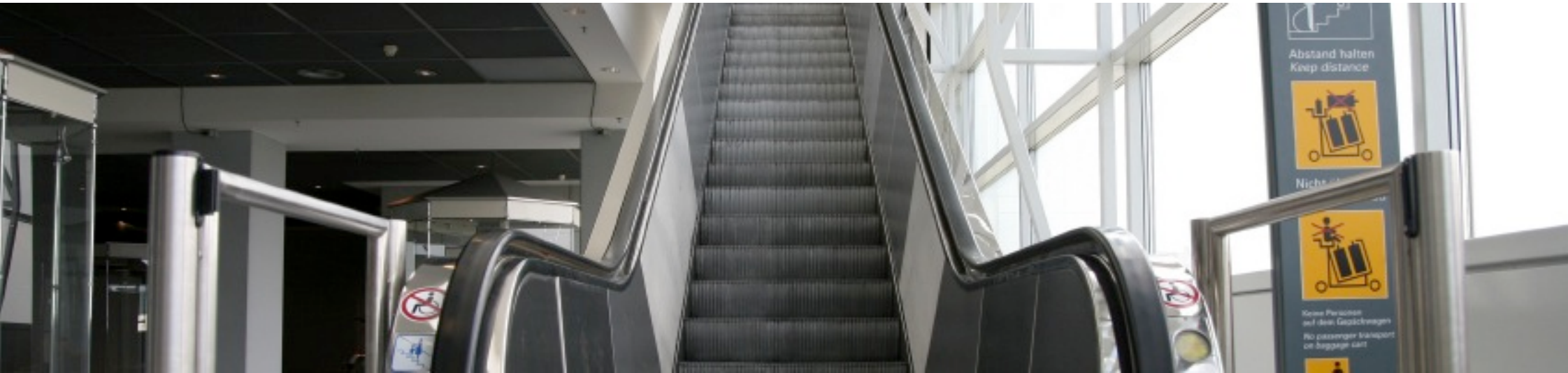
Error Handler

- Units often don’t have enough time or information to handle errors
- Separate business logic and error handling
- Business logic just focuses on getting the task done (quickly)
- Error handler has sufficient time and information to handle errors

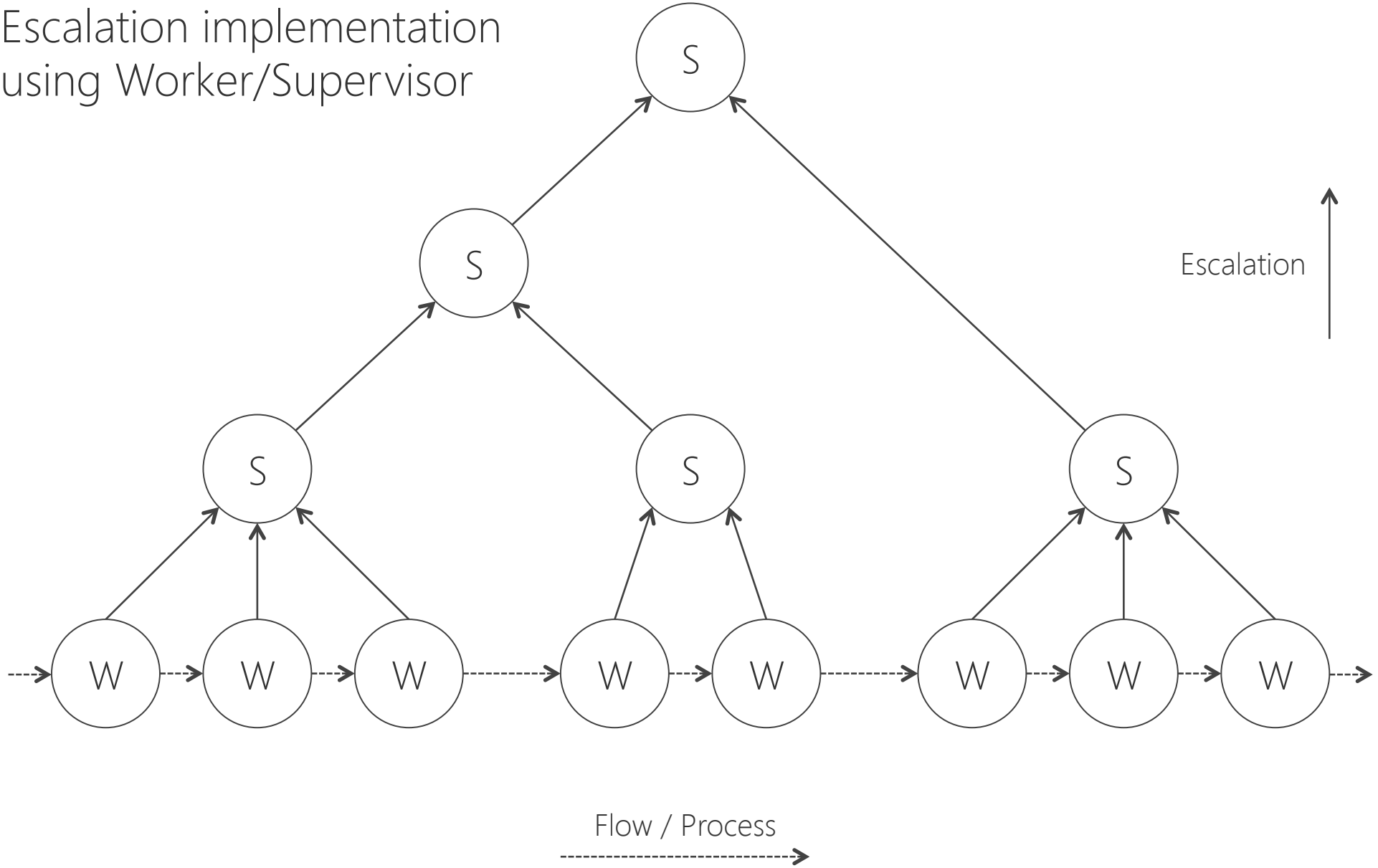

Idempotency
Relaxed Temporal Constraints
Self-Containment
Bounded Queues
Fan out & quickest reply
Circuit Breaker
Shed Load
Latency Control
Fail Fast
Timeouts
Loose Coupling
Isolation
Asynchronous Communication
Location Transparency
Event-Driven
Stateless
Complete Parameter Checking
Bulkheads
Supervision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

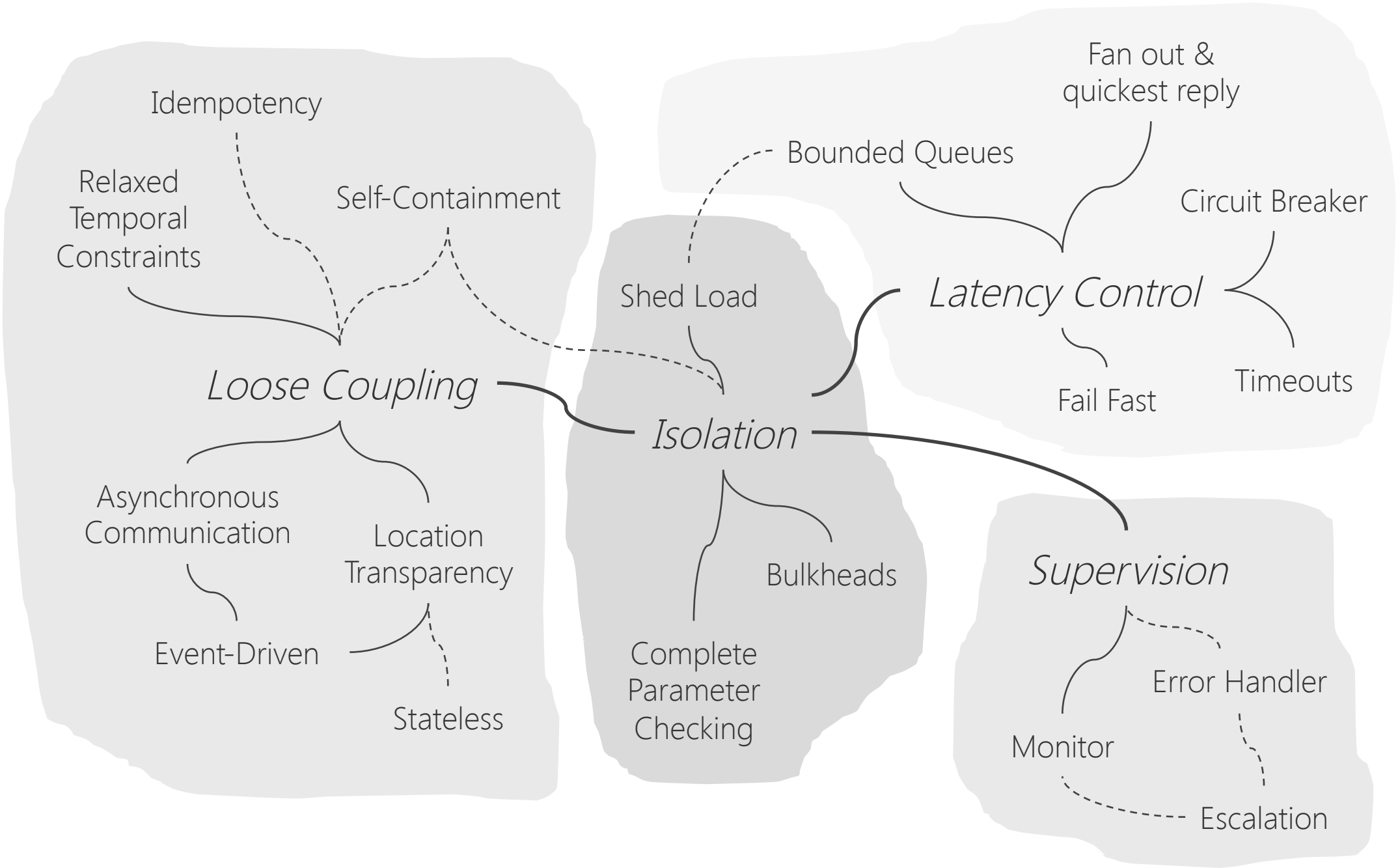
Escalation

- Units often don’t have enough time or information to handle errors
- Escalation peer with more time and information needed
- Often multi-level hierarchies
- Pure design issue



Escalation implementation using Worker/Supervisor




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

… and there is more

- Recovery & mitigation patterns
- More supervision patterns
- Architectural patterns
- Anti-fragility patterns
- Fault treatment & prevention patterns

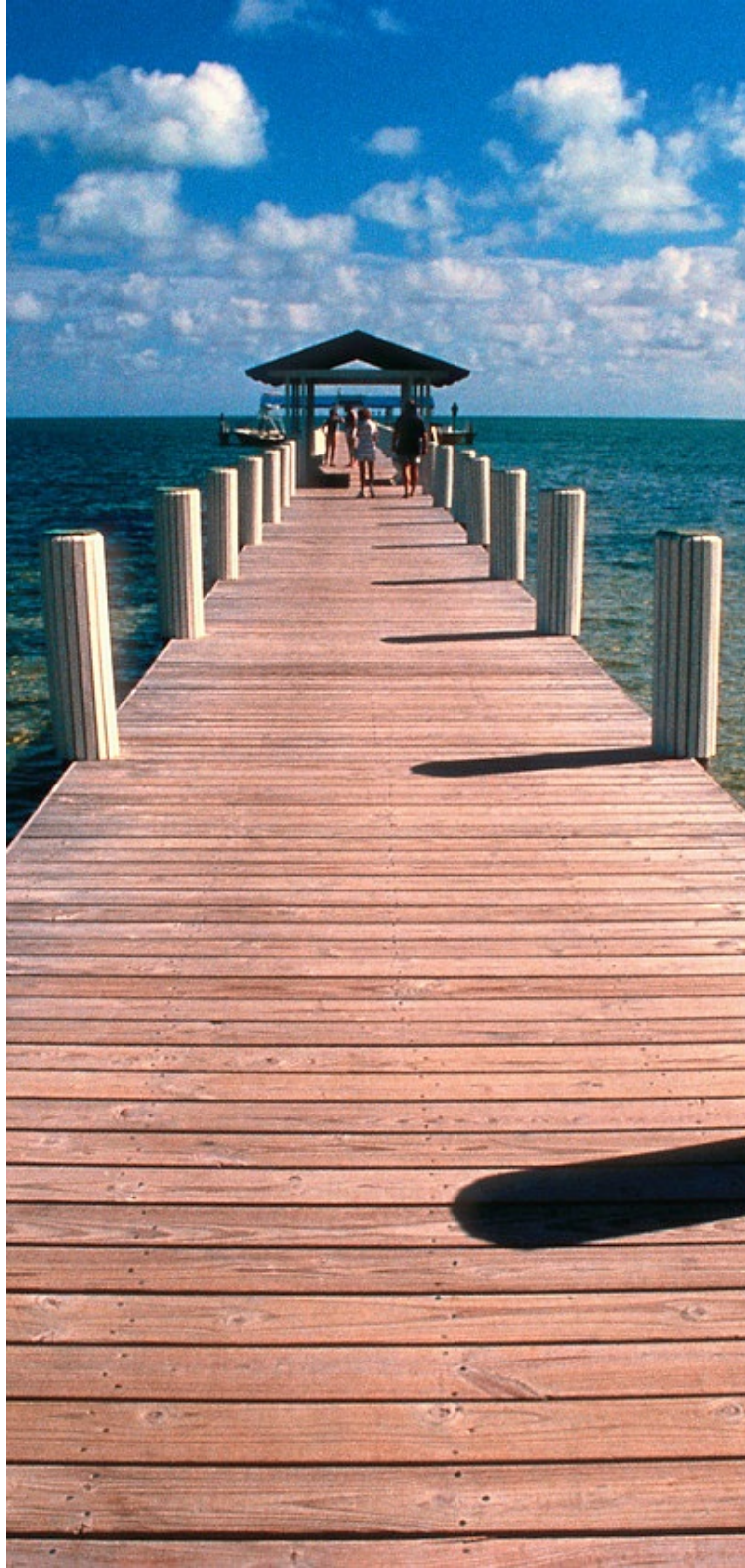
A rich pattern family

Wrap-up

- Today's systems are distributed …
- … and it's getting "worse"

- Failures are the normal case
- Failures are not predictable

- Resilient software design needed
- Rich pattern language
- Isolation is a good starting point



Do not avoid failures. Embrace them!

@ufried

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