



Resilient software design applied

A hands-on course for the undaunted developer

Uwe Friedrichsen & Ansgar Fitz (codecentric AG) – JAX – Mainz, 22. April 2016

Setting up the environment

With internet connection

- `git clone https://github.com/ufried/resilience-tutorial.git`
- `cd resilience-tutorial/recommendation-service`
- `mvn clean test`

Without internet connection

- *copy tutorial folder from USB stick to local directory*
- *copy repo folder from USB stick to local maven repository (~/.m2/repository)*
 - *Alternatively point local repository in **settings.xml** to copied repository folder*
- `cd <local tutorial folder>/recommendation-service`
- `mvn clean test`



@ufried

Uwe Friedrichsen

uwe.friedrichsen@codecentric.de



@afitz1712

Ansgar Fitz

ansgar.fitz@codecentric.de

Agenda

Introduction	30'
1. Bulkheads	60'
2. Complete parameter checking	50'
3. Timeout	50'
4. Retry	30'
5. Graceful degradation	30'
6. Failover	30'
7. Circuit breaker	50'
Outlook & wrap-up	30'

Breaks based on wall clock



Why resilience?

A perspective view of a long, brightly lit server room aisle. The aisle is flanked by tall, blue server cabinets on both sides. The floor is made of light-colored square tiles. The ceiling is white with several long, rectangular fluorescent light fixtures. In the distance, a person is walking away from the camera, their figure slightly blurred. A semi-transparent white banner with black text is overlaid across the middle of the image.

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

Traditional stability approach

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



Maximize MTTF

(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 today's complex, distributed and interconnected systems are not the exception.

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

Do not try to avoid failures. Embrace them.

Resilience approach

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



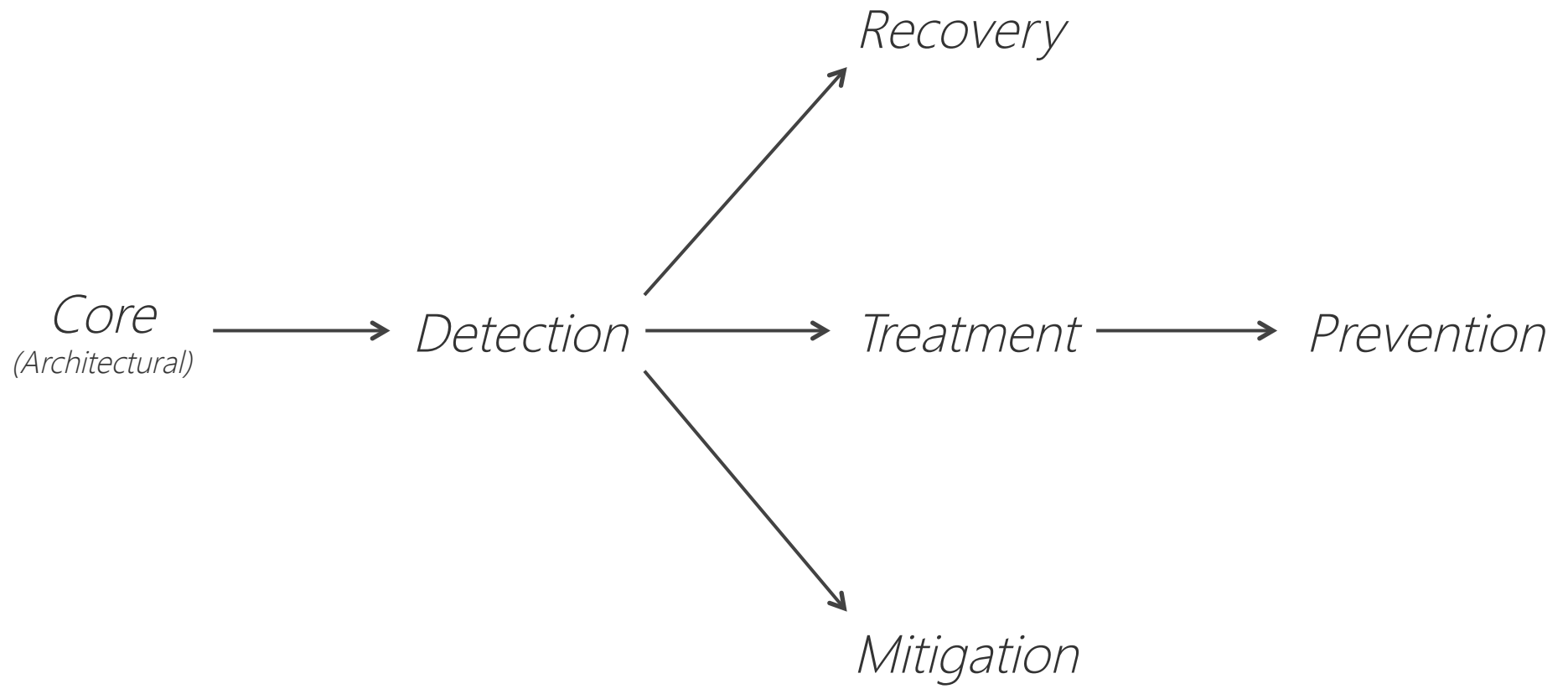
Minimize MTTR

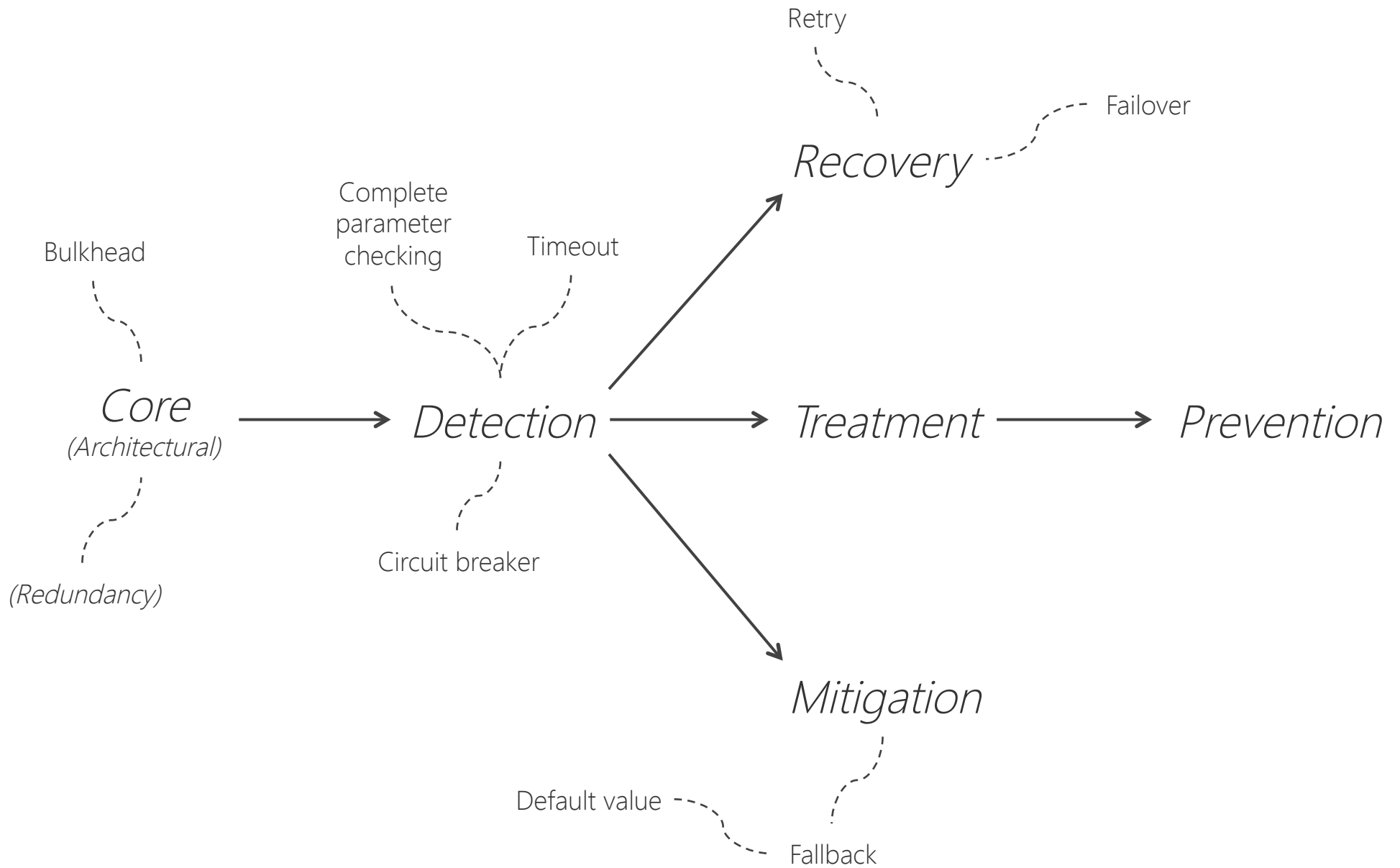
resilience (IT)

the ability of a system to handle unexpected situations

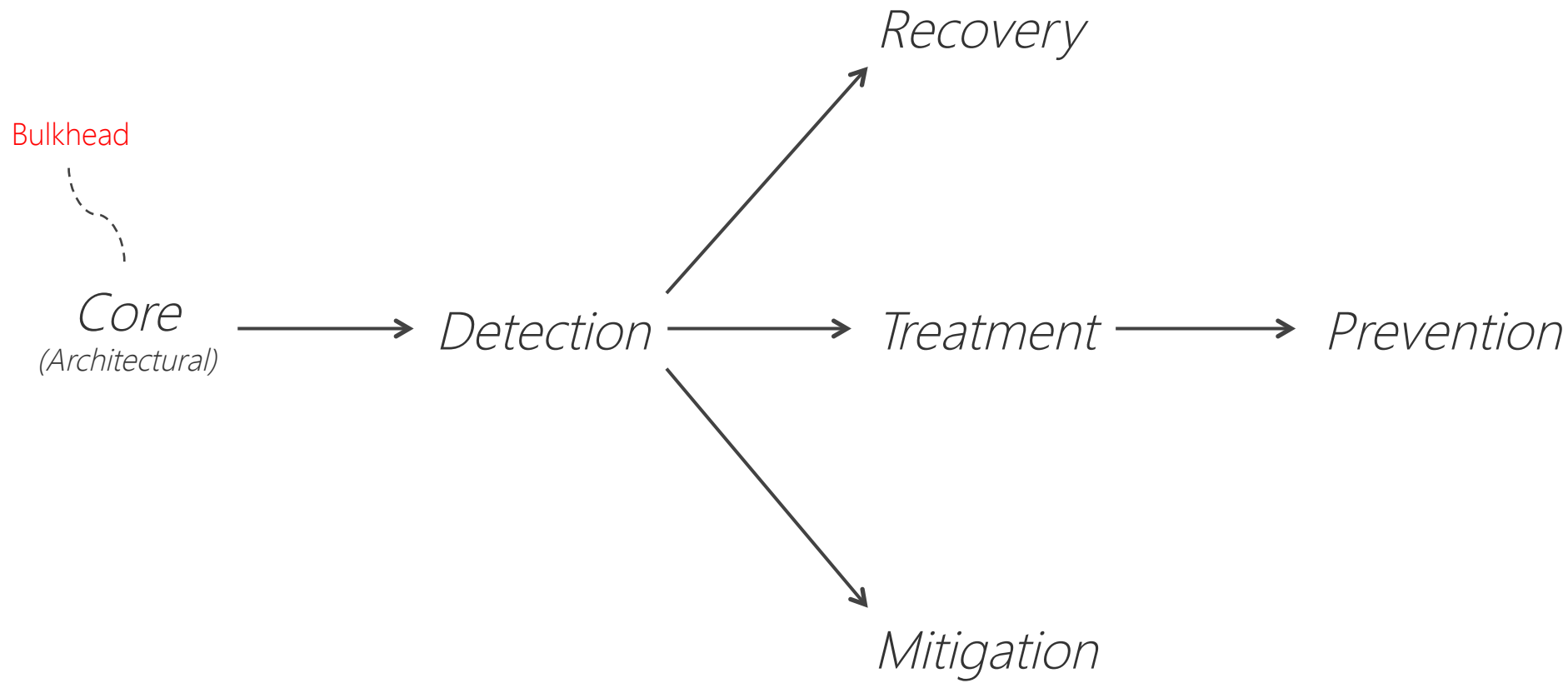
- without the user noticing it (best case)
- with a graceful degradation of service (worst case)

A little taxonomy on resilience ...





Case Study: Awesome, Inc!



Bulkhead

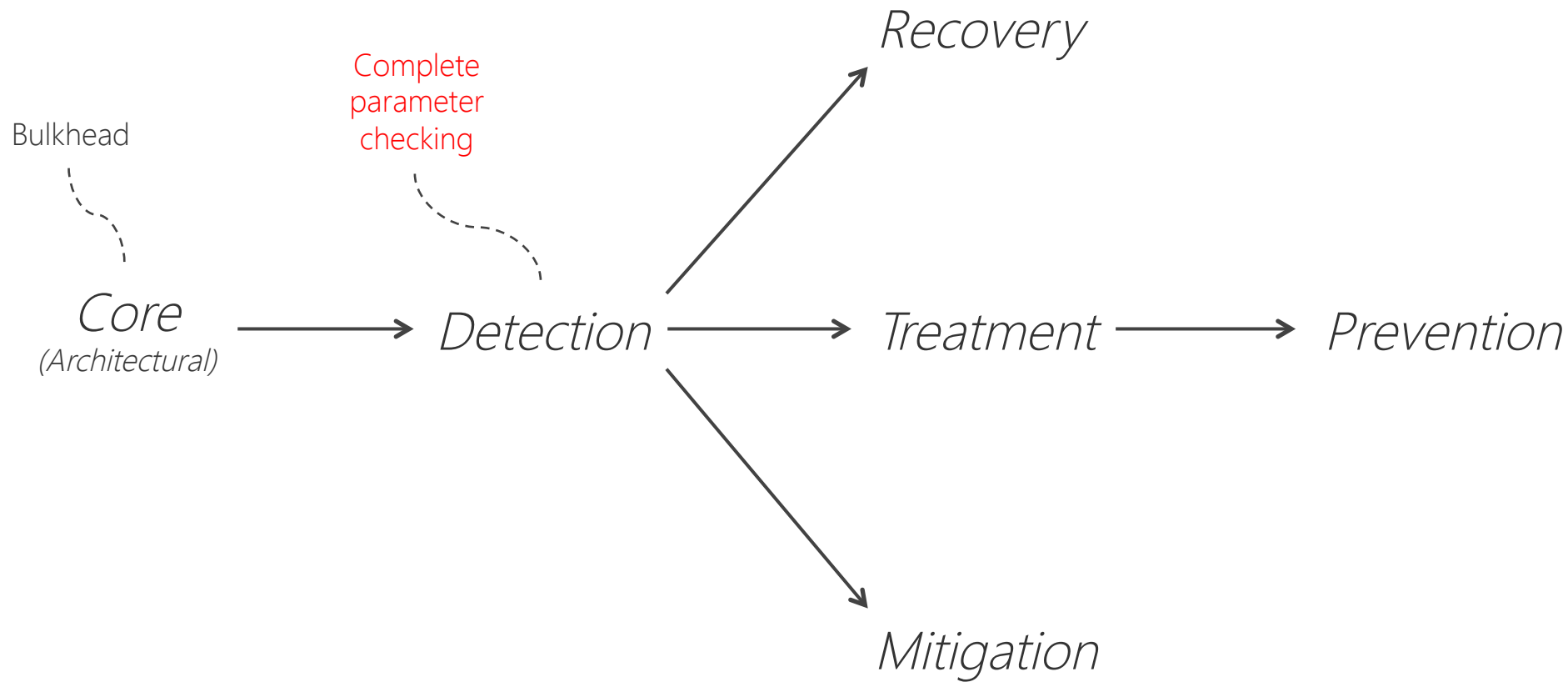
Bulkhead

- Core isolation pattern (a.k.a. “failure units” or “units of mitigation”)
- Shaping good bulkheads is hard (pure design issue)
- Diverse implementation choices available, e.g., μ service, actor, scs, ...
- Implementation choice impacts system and resilience design a lot



Exercise 1: Bulkhead

Case Study: Recommendation service



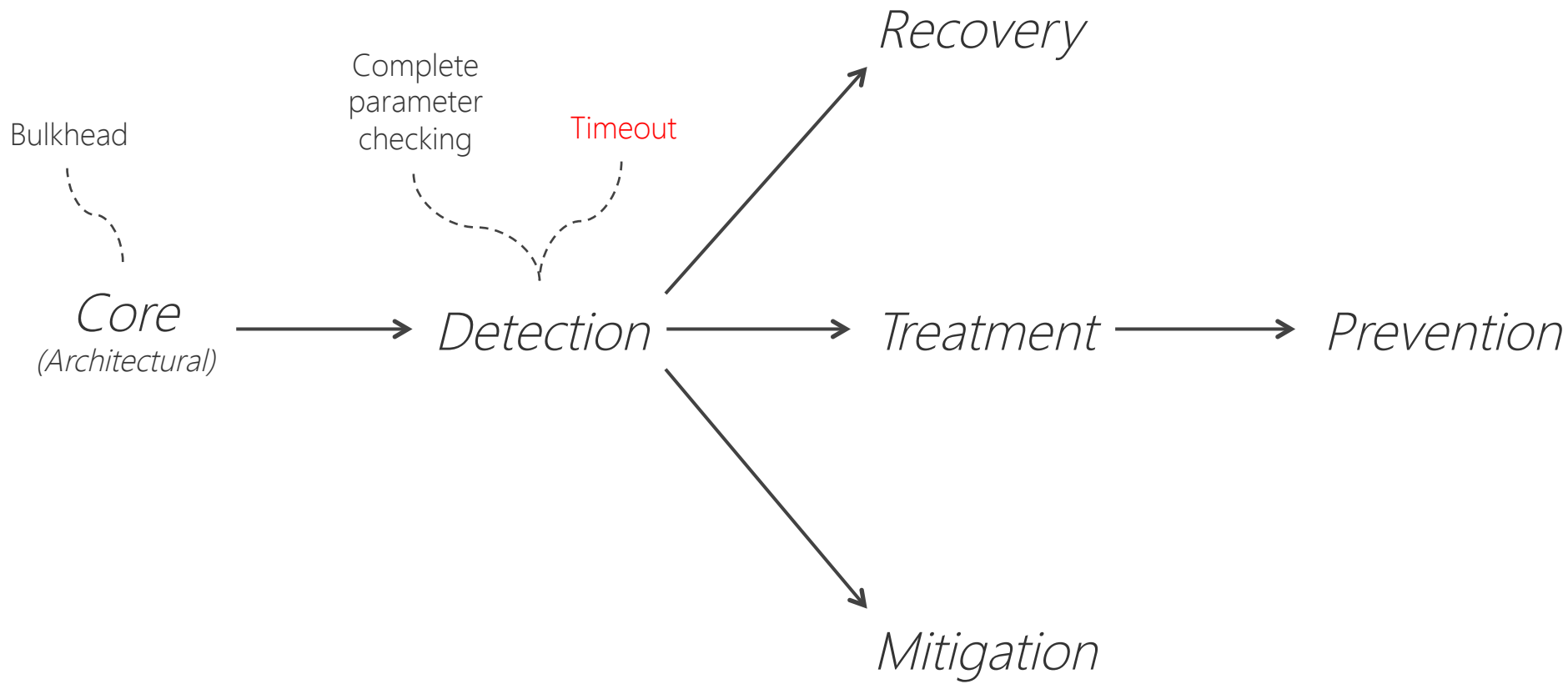
Complete parameter checking

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



Exercise 2: Complete parameter checking



Timeout

Timeout

- 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



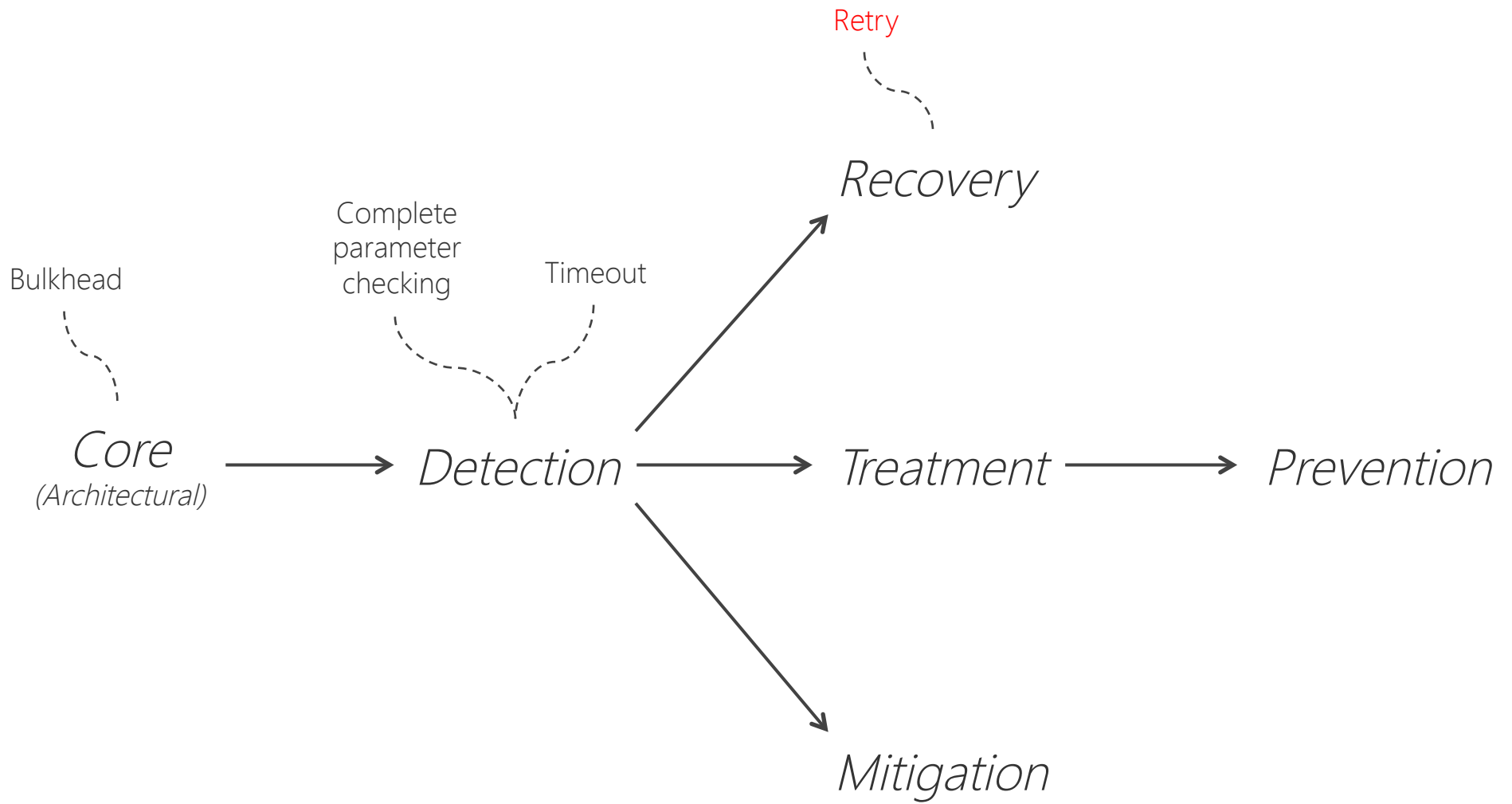
Timeout with standard library means

```
// 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);
}
```

Exercise 3: Timeout



Retry

Retry

- Very basic recovery pattern
- Recover from omission errors
- Limit retries to minimize extra load on an already loaded resource
- Limit retries to avoid recurring errors



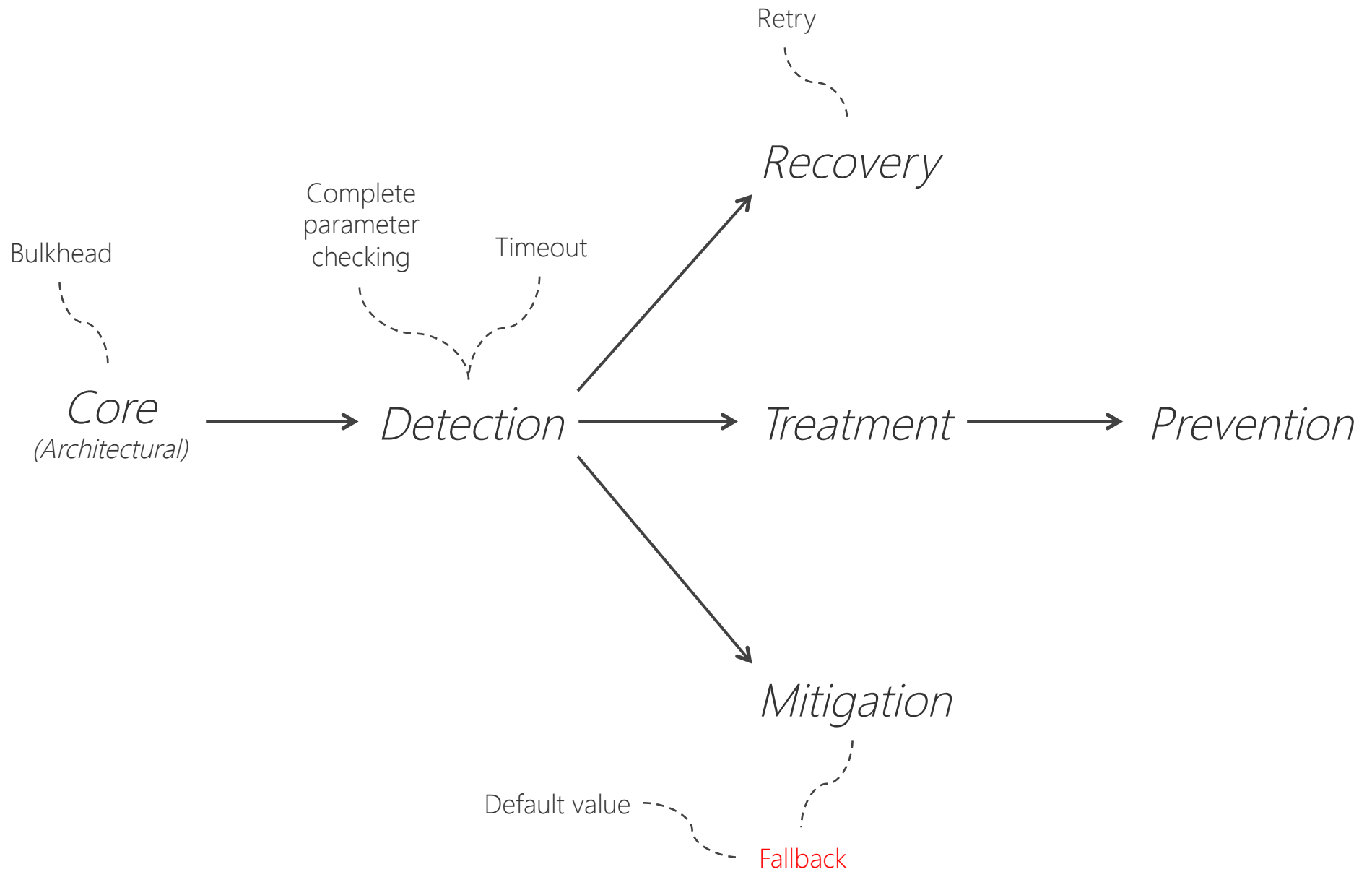
Retry example

```
// doAction returns true if successful, false otherwise
boolean doAction(...) {
    ...
}
```

```
// General pattern
boolean success = false
int tries = 0;
while (!success && (tries < MAX_TRIES)) {
    success = doAction(...);
    tries++;
}
```

```
// Alternative one-retry-only variant
success = doAction(...) || doAction(...);
```

Exercise 4: Retry



Fallback

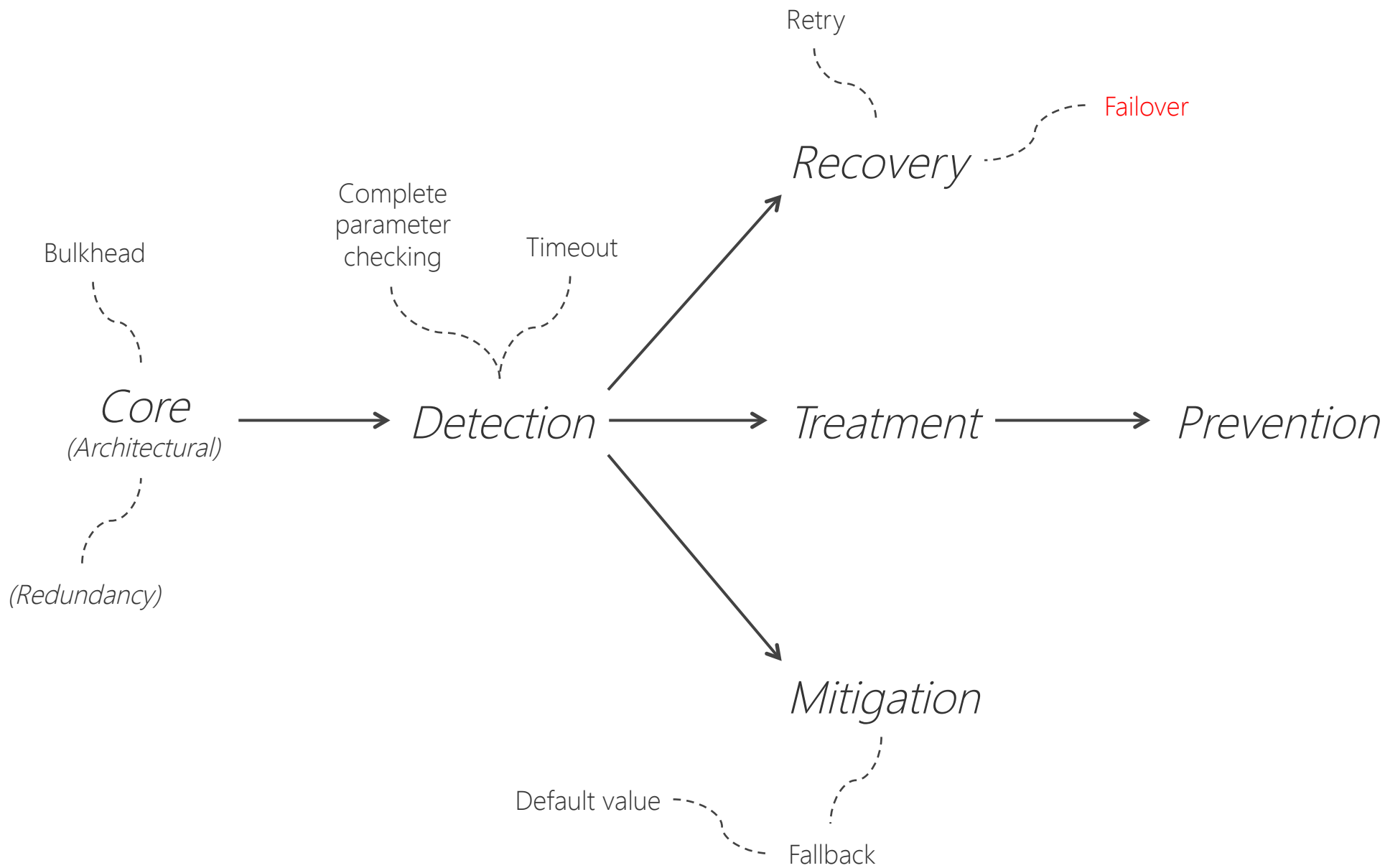
(Graceful degradation of service)

Fallback

- Execute an alternative action if the original action fails
- Basis for most mitigation patterns
- Fail silently – silently ignore the error and continue processing
- Default value – return a predefined default value if an error occurs



Exercise 5: Graceful degradation



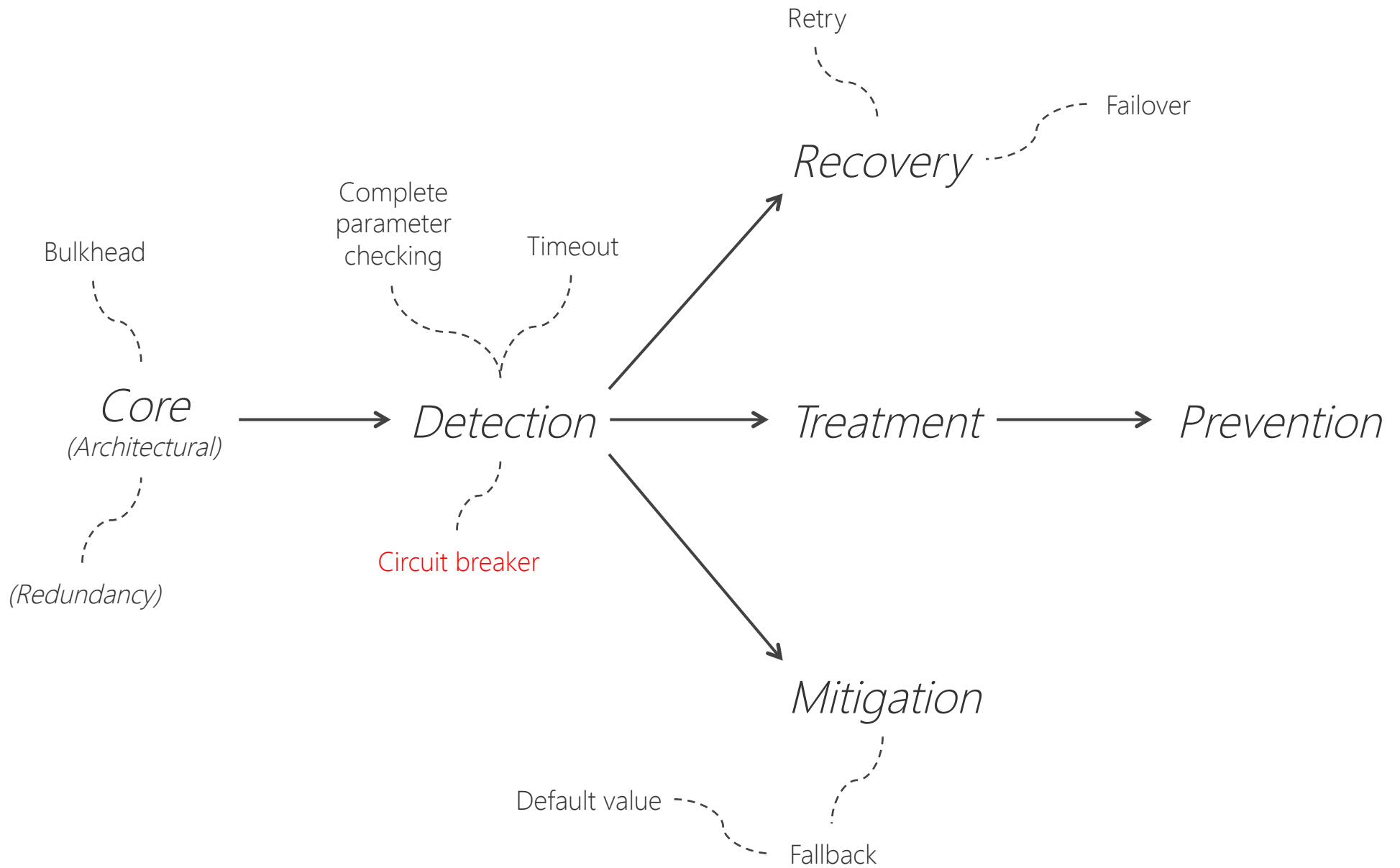
Failover

Failover

- Used as escalation if other measures failed or would take too long
- Requires redundancy – trades resources for availability
- Many implementation variants available, incl. out-of-the-box solutions
- Usually implemented as a monitor-dynamic router combination



Exercise 6: Failover



Circuit breaker

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



PUBLIC



Netflix / Hystrix

★ Star

1,580

🔗 Fork

219

Home

Pages

History

Home

Page History

Clone URL



HYSTRIX

DEFEND YOUR APP

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

- [Home](#)
- [Getting Started](#)
- [How To Use](#)
 - [Hello World!](#)
 - [Synchronous Execution](#)
 - [Asynchronous Execution](#)
 - [Reactive Execution](#)
 - [Fallback](#)
 - [Error Propagation](#)
 - [Command Name](#)
 - [Command Group](#)
 - [Command Thread Pool](#)
 - [Request Cache](#)
 - [Request Collapsing](#)
 - [Request Context Setup](#)
 - [Common Patterns](#)
 - [Migrating to Hystrix](#)
- [How It Works](#)
 - [Execution Flow](#)
 - [Circuit Breaker](#)
 - [Isolation](#)
 - [Request Collapsing](#)
 - [Request Caching](#)



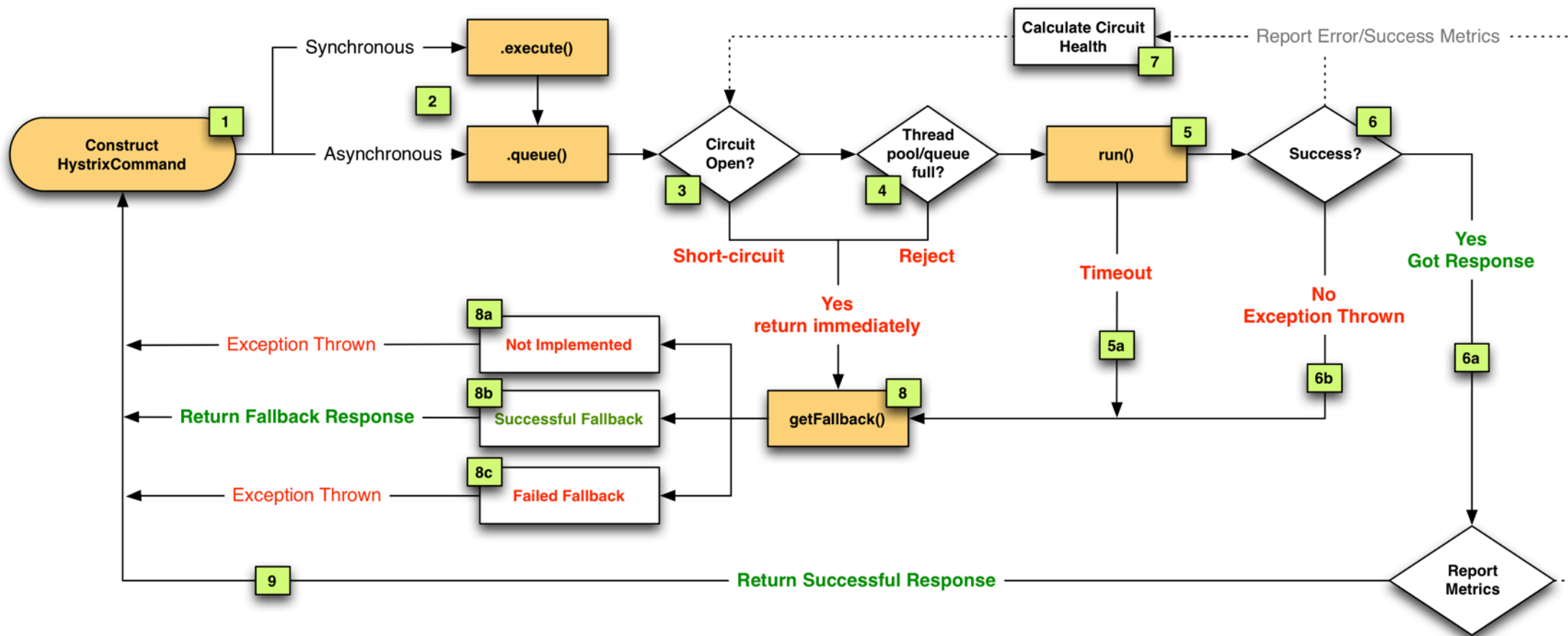

```
// Hystrix "Hello world"

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);
}
```

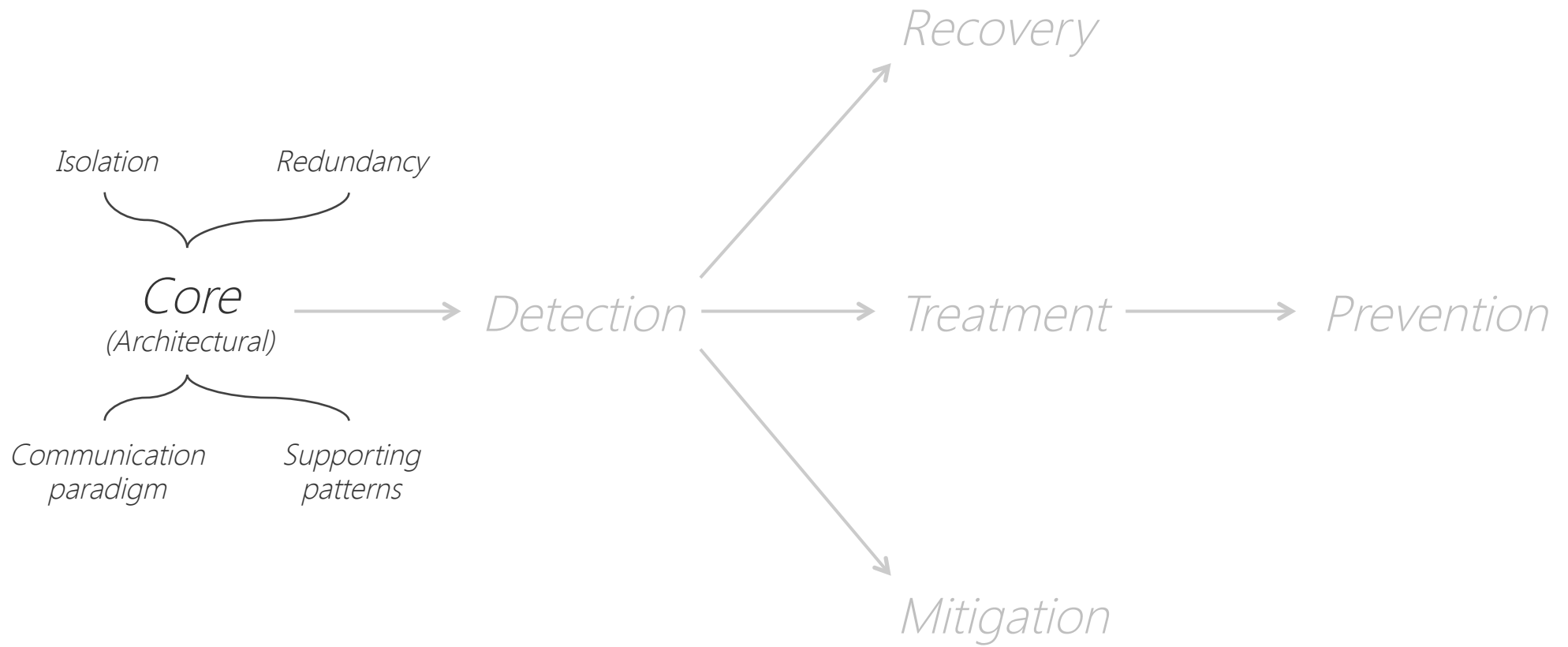


Exercise 7: Circuit breaker

```
// Required super class constructor call for this exercise
```

```
super(HystrixCommand.Setter  
    .withGroupKey(HystrixCommandGroupKey.Factory.asKey("AnalysisService"))  
    .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()  
        .withExecutionTimeoutInMilliseconds(200)  
        .withCircuitBreakerRequestVolumeThreshold(1)  
        .withMetricsRollingStatisticalWindowInMilliseconds(1000)  
        .withMetricsRollingStatisticalWindowBuckets(1)  
        .withMetricsRollingPercentileEnabled(false)));
```

... and that's just the beginning



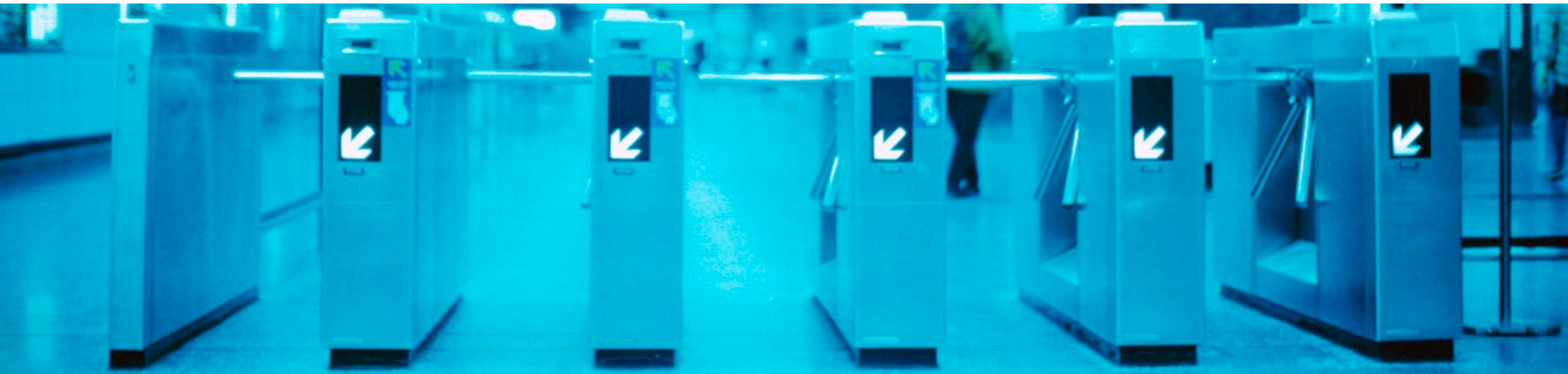
Communication paradigm

- Request-response \leftrightarrow messaging \leftrightarrow events
- Not a pattern, but heavily influences resilience patterns to be used
- Also heavily influences functional bulkhead design
- Very fundamental decision which is often underestimated



Redundancy

- Core resilience concept
- Applicable to all failure types
- Basis for many recovery and mitigation patterns
- Often different variants implemented in a system





Failure types

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

Usage of redundancy

- Patterns
 - Failover
- Schemes
 - Active/Passive
 - Active/Active
 - N+M Redundancy
- Implementation examples
 - Load balancer + health check (e.g., HAProxy)
 - Dynamic routing + health check (e.g., Consul, ZooKeeper)
 - Cluster manager with shared IP (e.g., Pacemaker & Corosync)

Failure types

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

Failure types

Usage of redundancy

- Patterns
 - Backup Request
 - Retry (to different replica)
 - Failover
- Schemes
 - Identical replicas
 - Failover schemes (for failover)
- Implementation examples
 - Client-based routing
 - Load balancer
 - Leaky bucket + dynamic routing

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

Failure types

Usage of redundancy

- Patterns
 - Backup Request
 - Timeout + retry to different replica
 - Timeout + failover
- Schemes
 - Identical replicas
 - Failover schemes (for failover)
- Implementation examples
 - Client-based routing
 - Load balancer
 - Circuit breaker + dynamic routing

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

Failure types

Usage of redundancy

- Patterns
 - Voting
 - Recovery blocks
 - Routine exercise
- Schemes
 - Identical replicas
 - Different replicas (recovery blocks)
- Implementation examples
 - Majority based quorum
 - Adaptive weighted sum
 - Synthetic computation

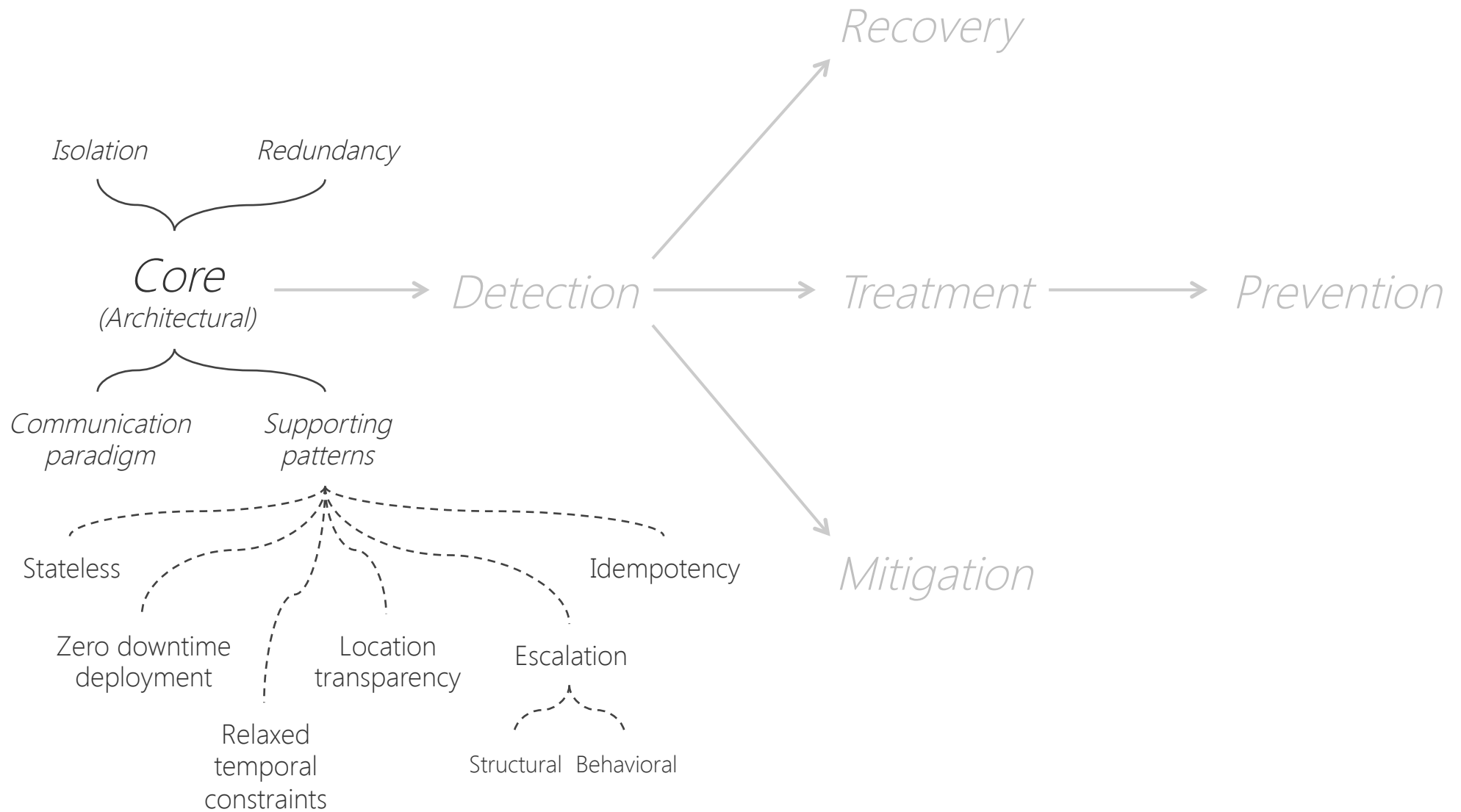
- Crash failure
- Omission failure
- Timing failure
- Response failure
- Byzantine failure

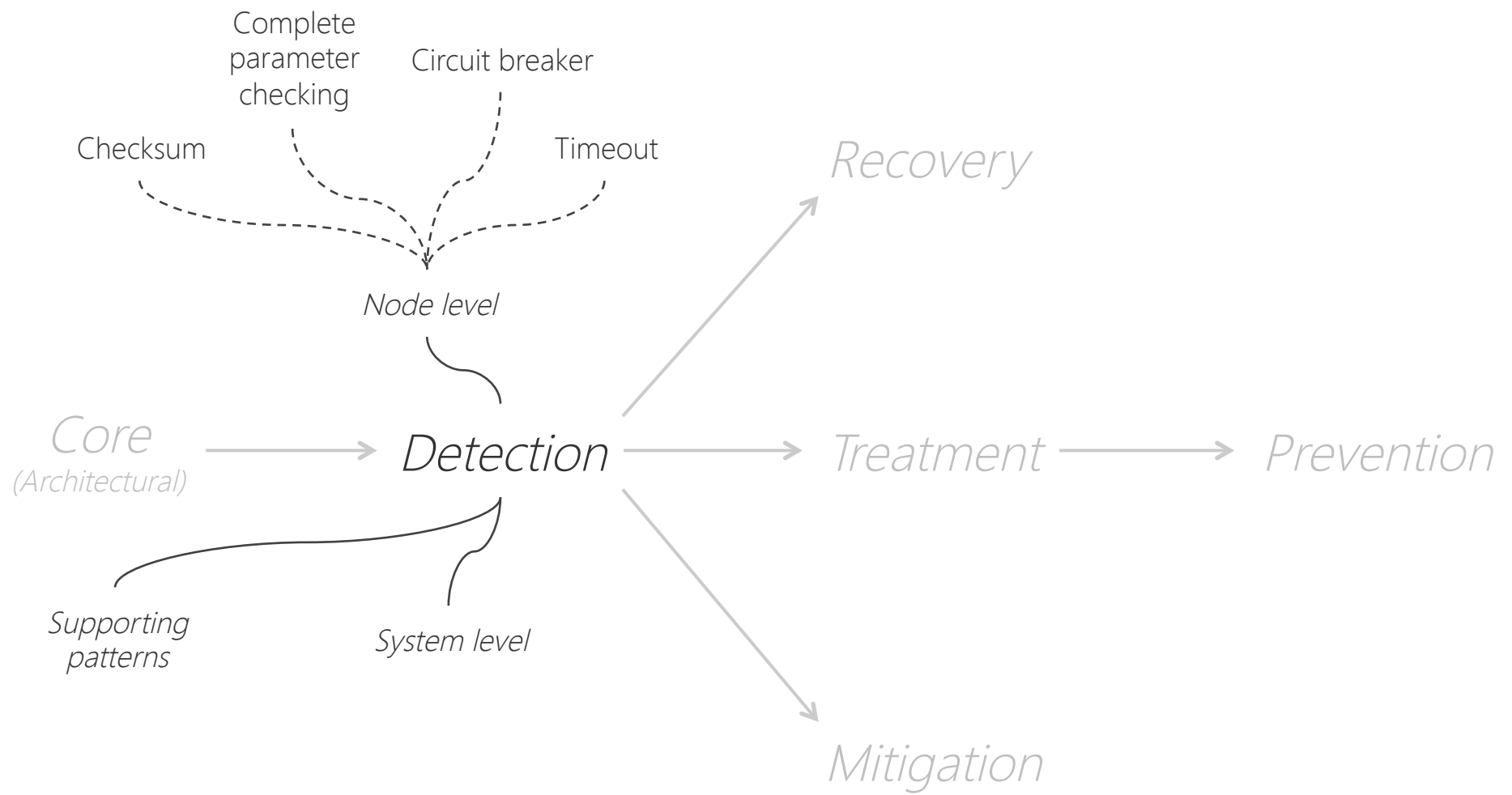
Failure types

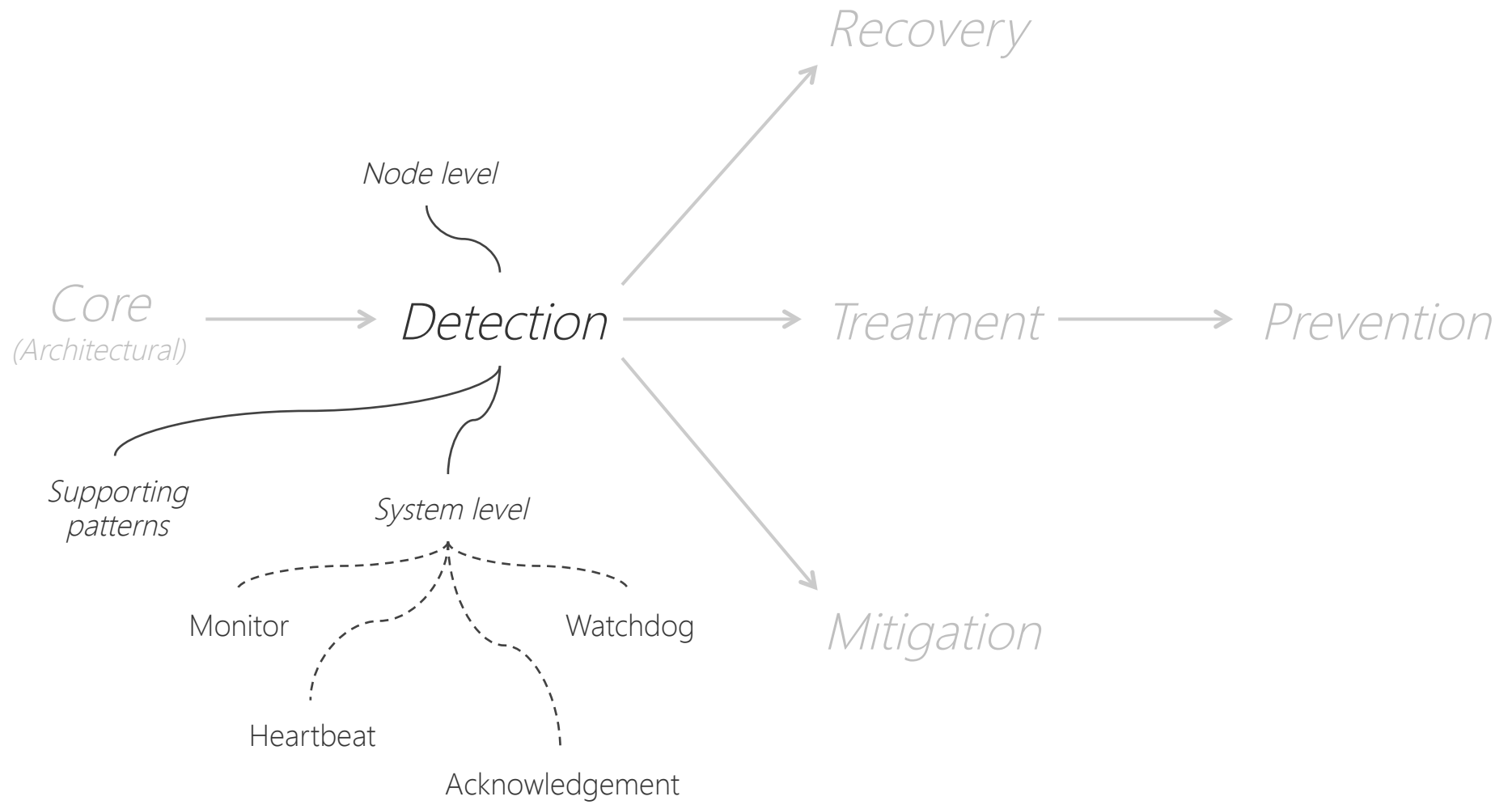
Usage of redundancy

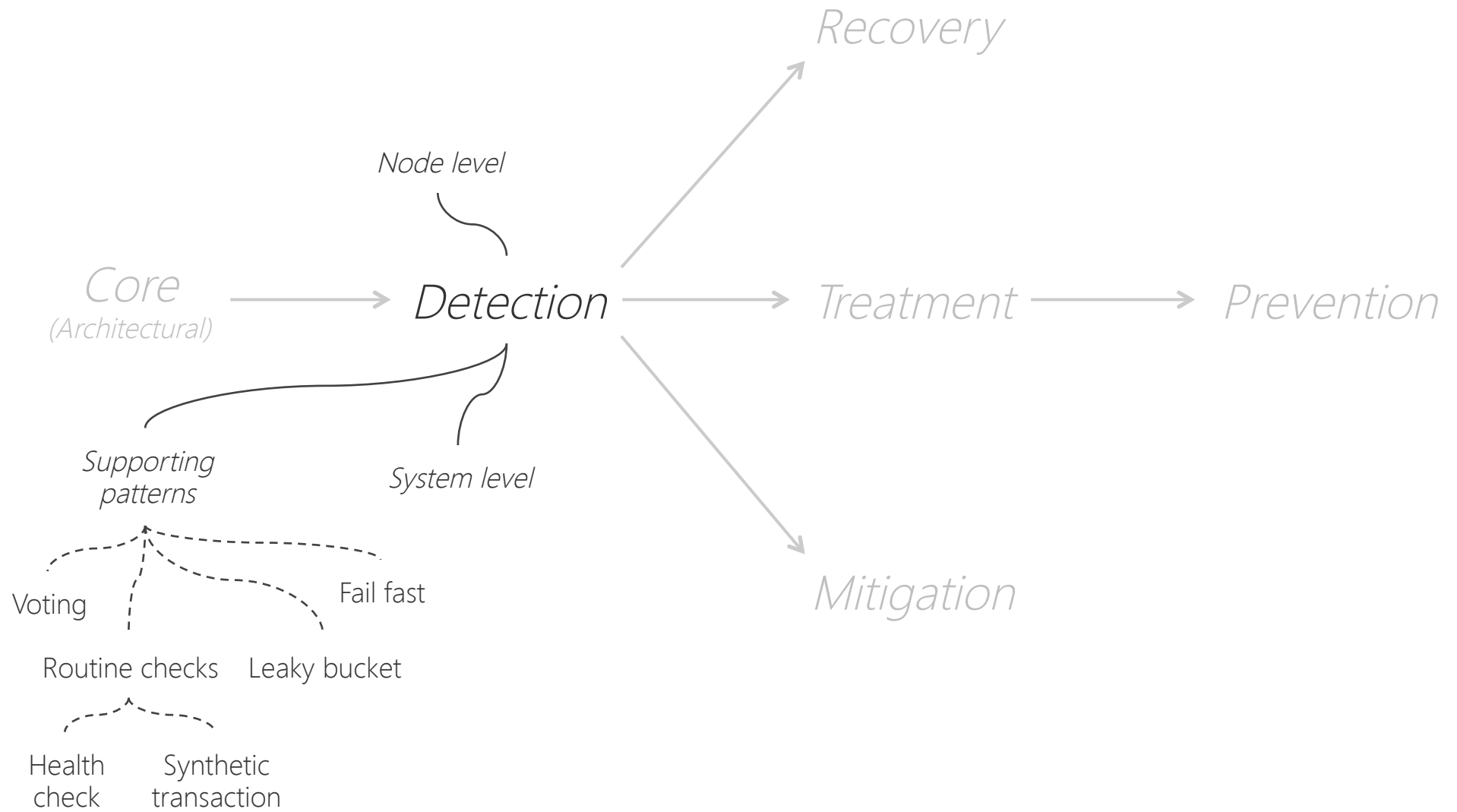
- Patterns
 - Voting
 - Recovery blocks
 - Routine exercise
- Schemes
 - Identical replicas
 - Different replicas (recovery blocks)
- Implementation examples
 - $n > 3t$ quorum
 - Adaptive weighted sum
 - Synthetic computation

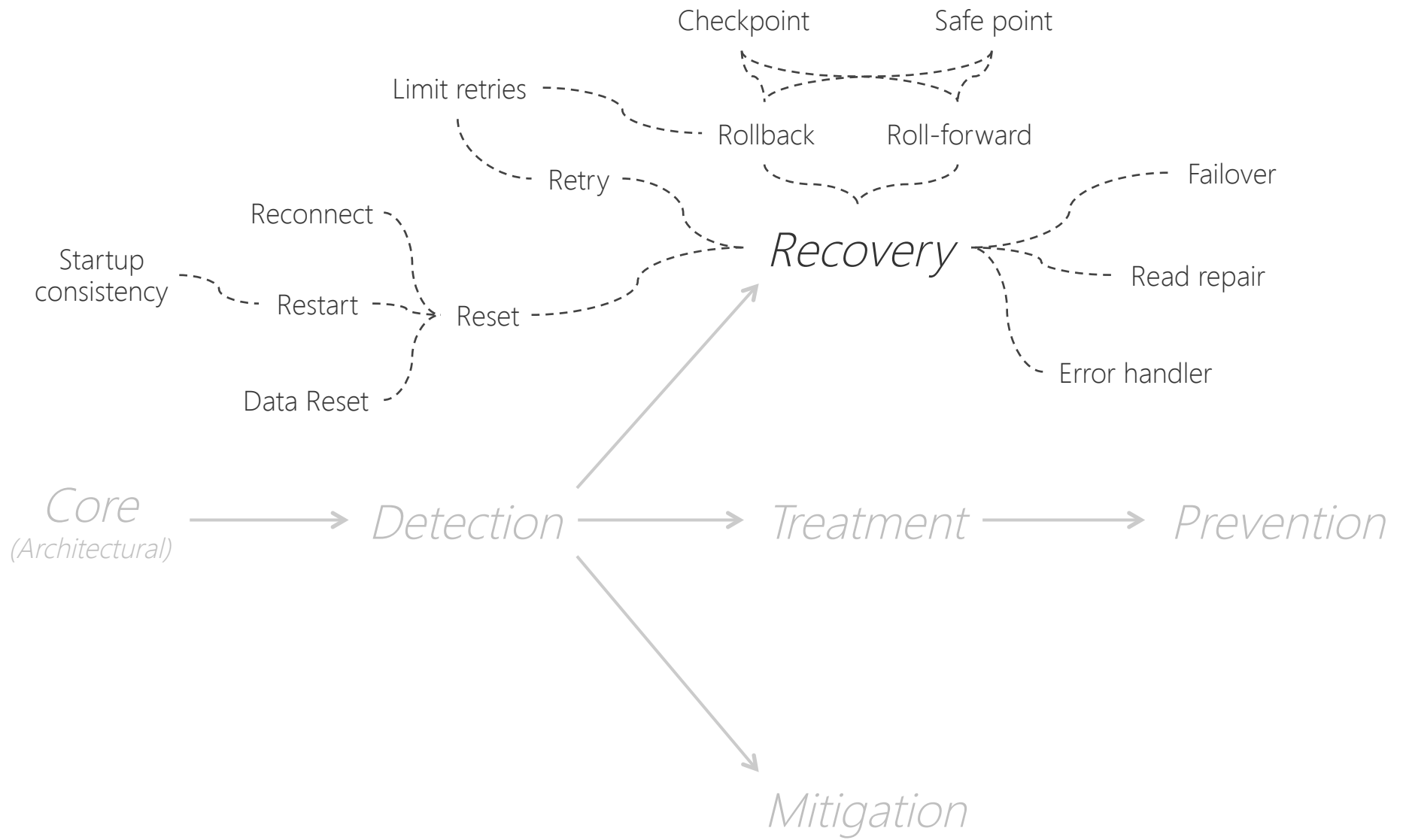
- Crash failure
- Omission failure
- Timing failure
- Response failure
- Byzantine failure

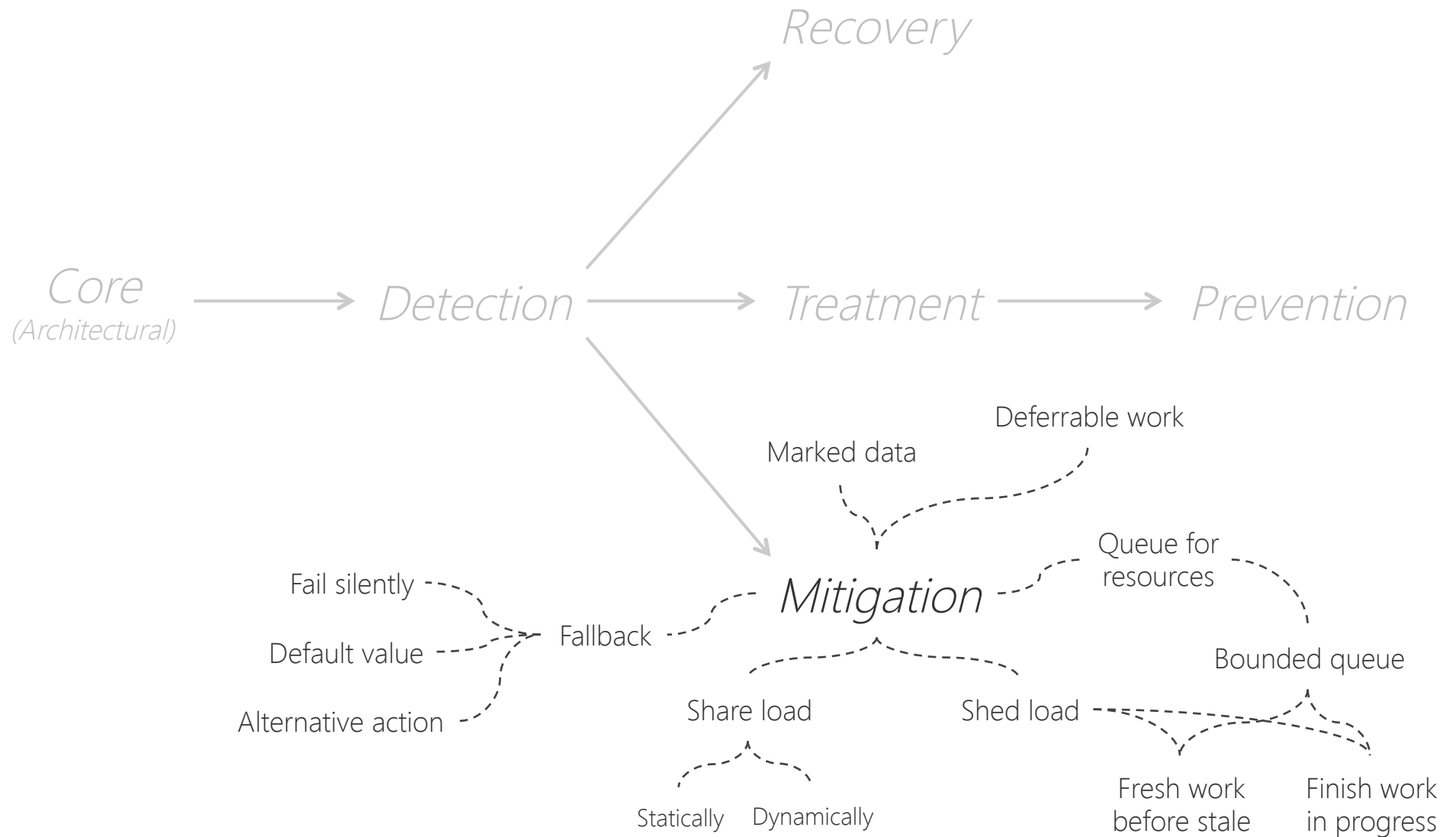


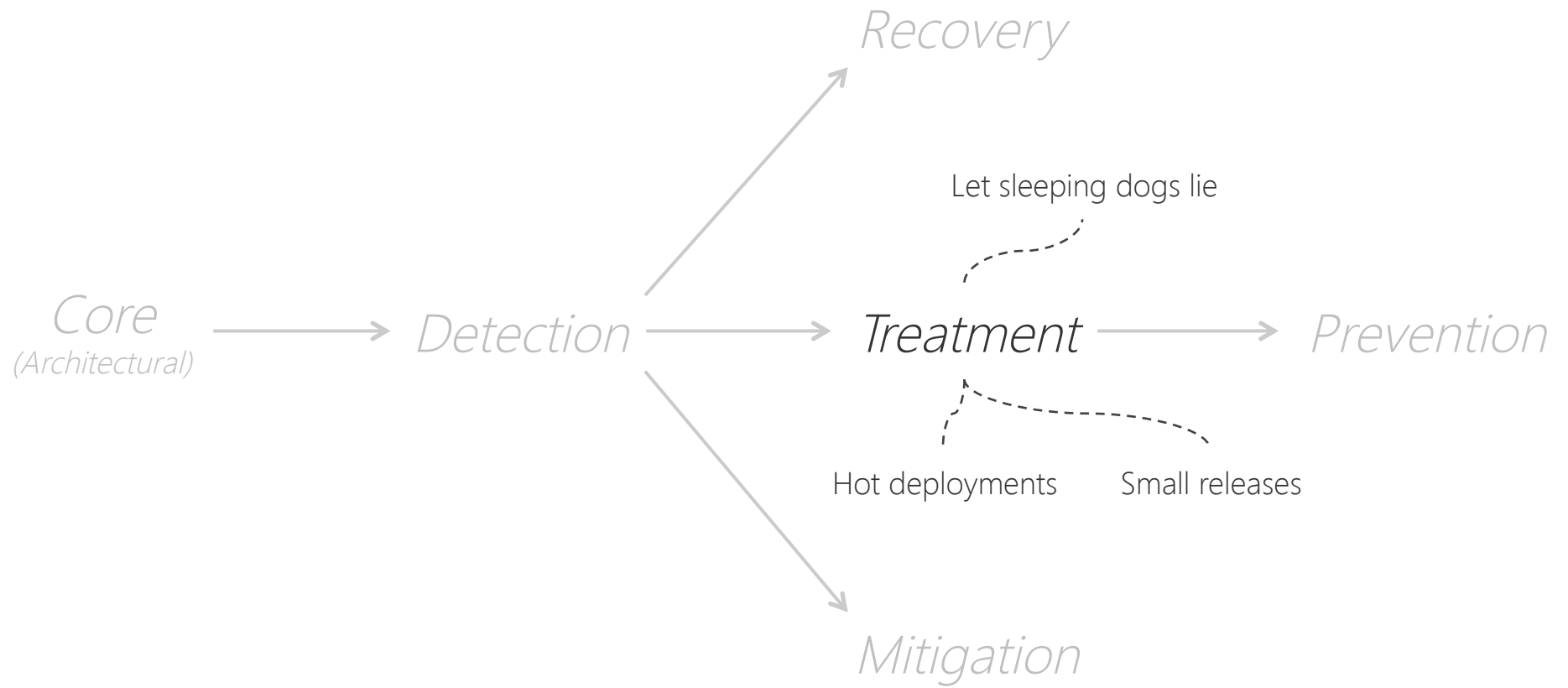


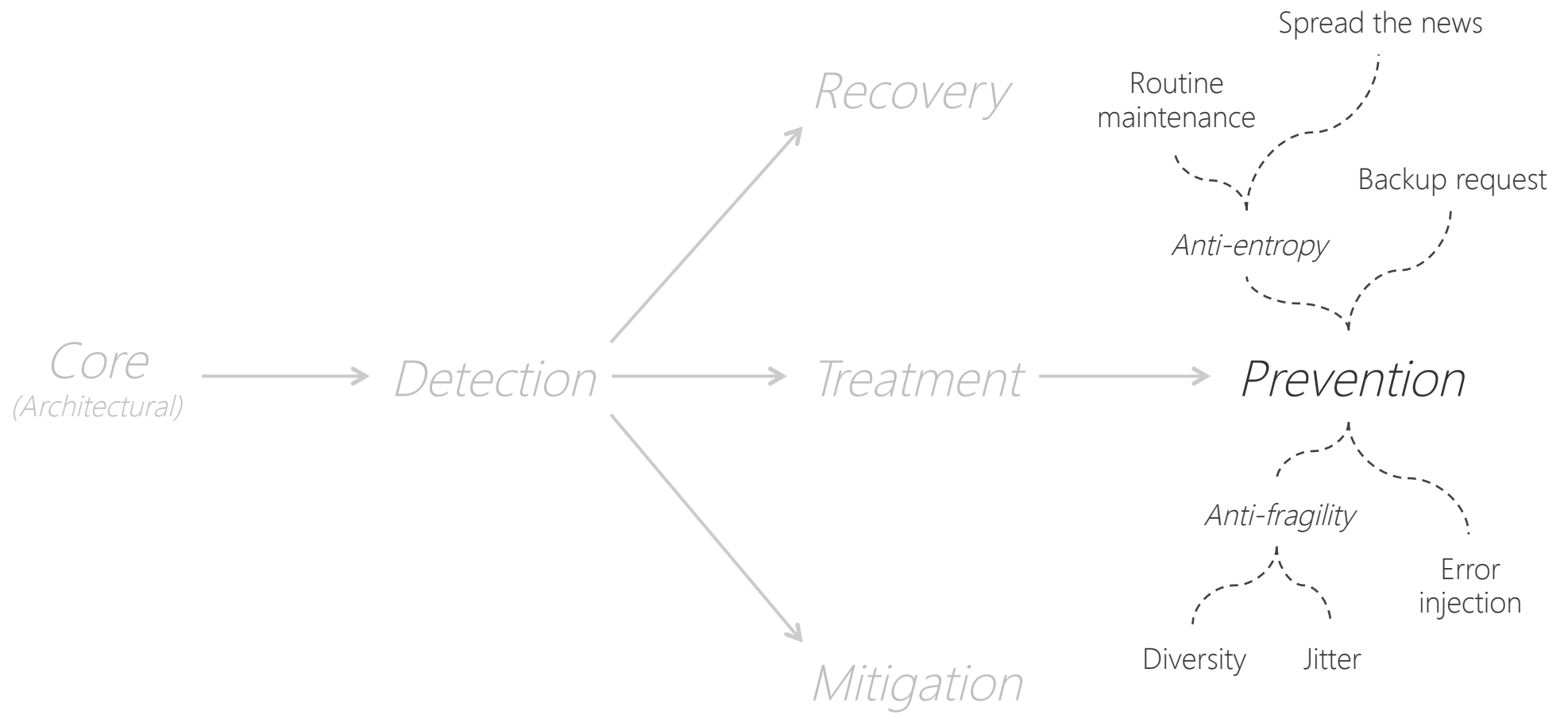










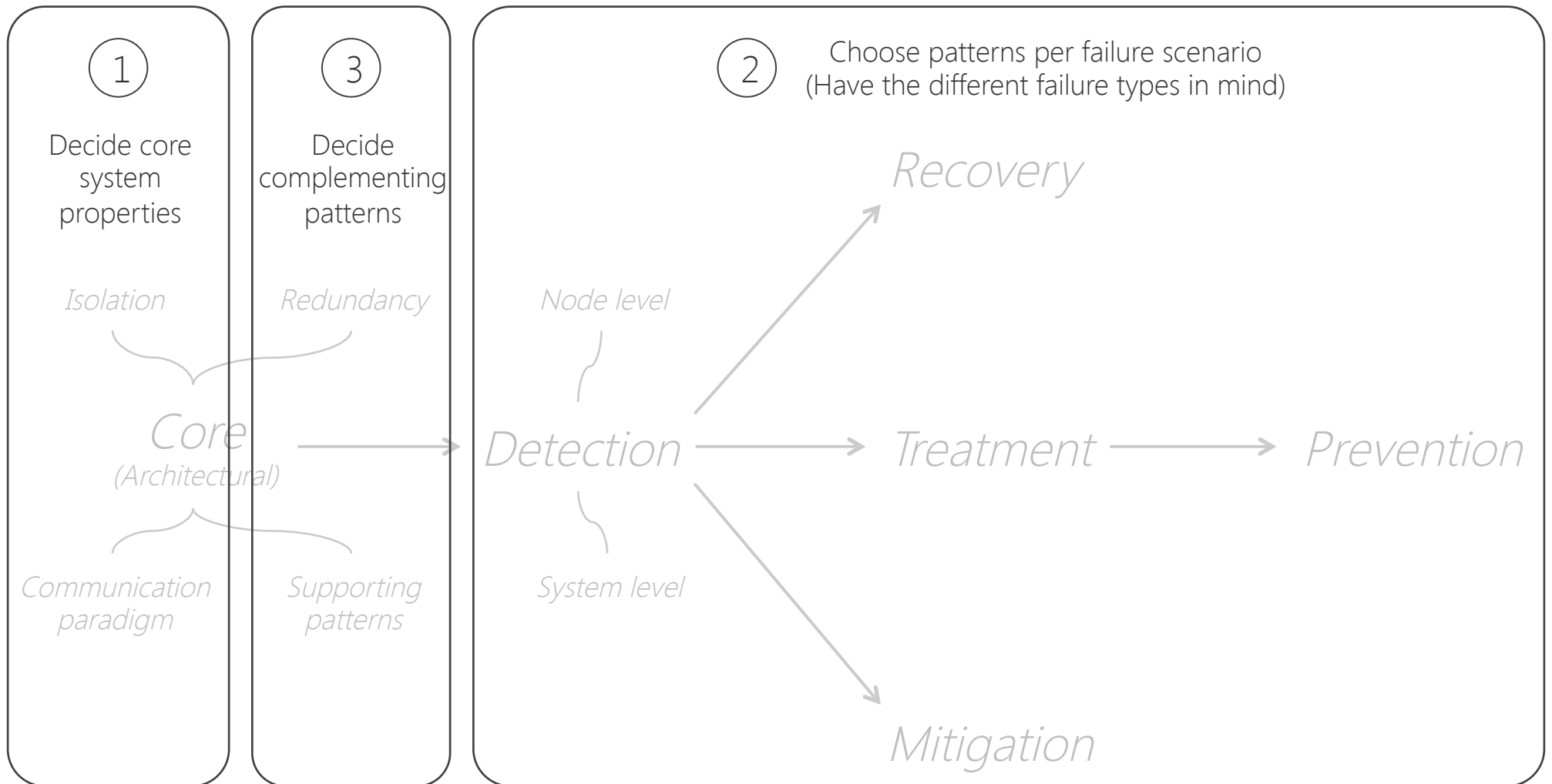


Towards a pattern language ...

Decisions to make

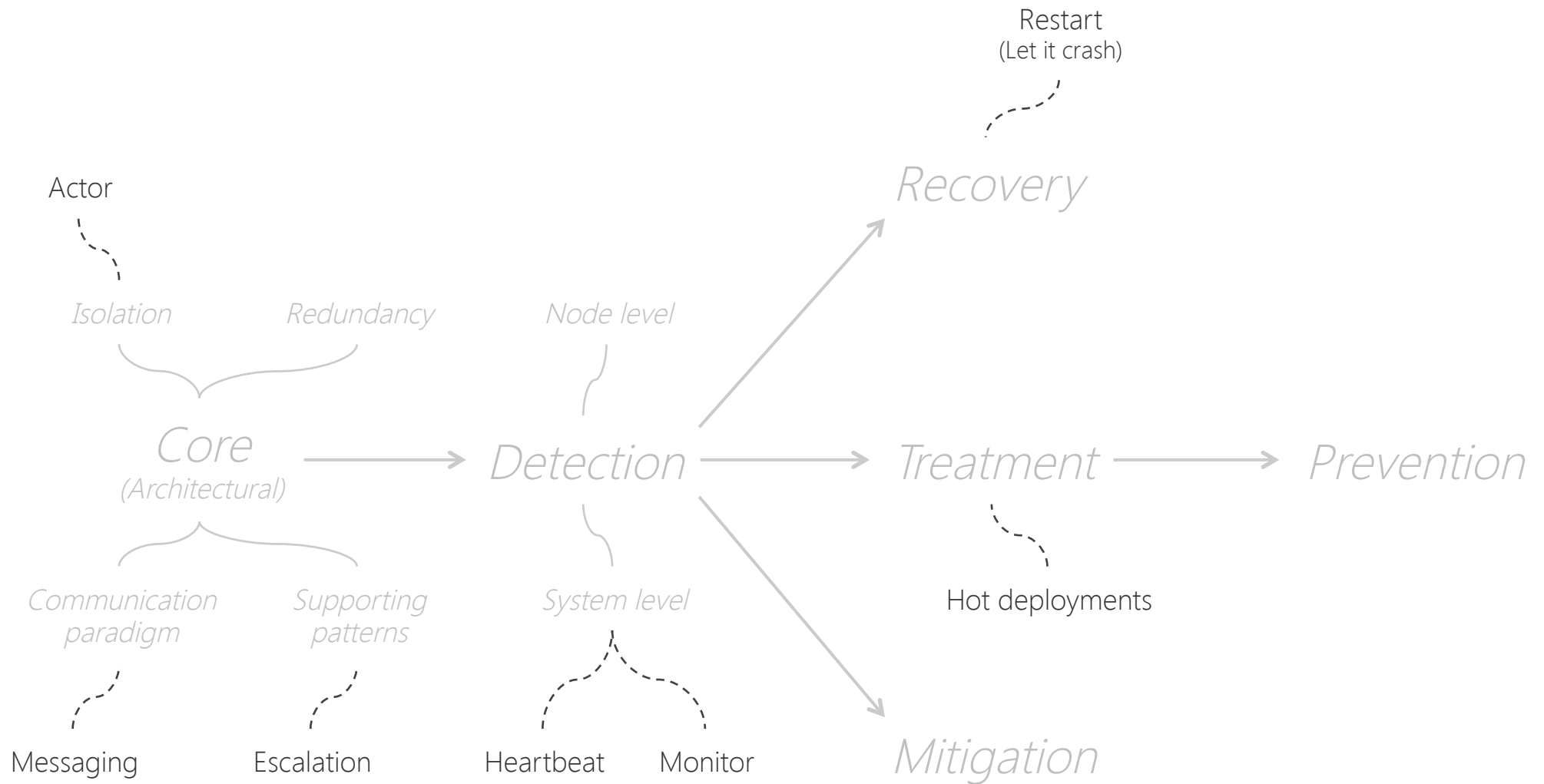
- General decisions
 - Bulkhead type
 - Communication paradigm
- Decisions per failure scenario (repeat)
 - Error detection on node & system level
 - Recovery/mitigation mechanism
 - Supporting treatment mechanism
 - Supporting prevention mechanism
- Complementing decisions
 - Complementing redundancy mechanism(s)
 - Complementing architectural patterns



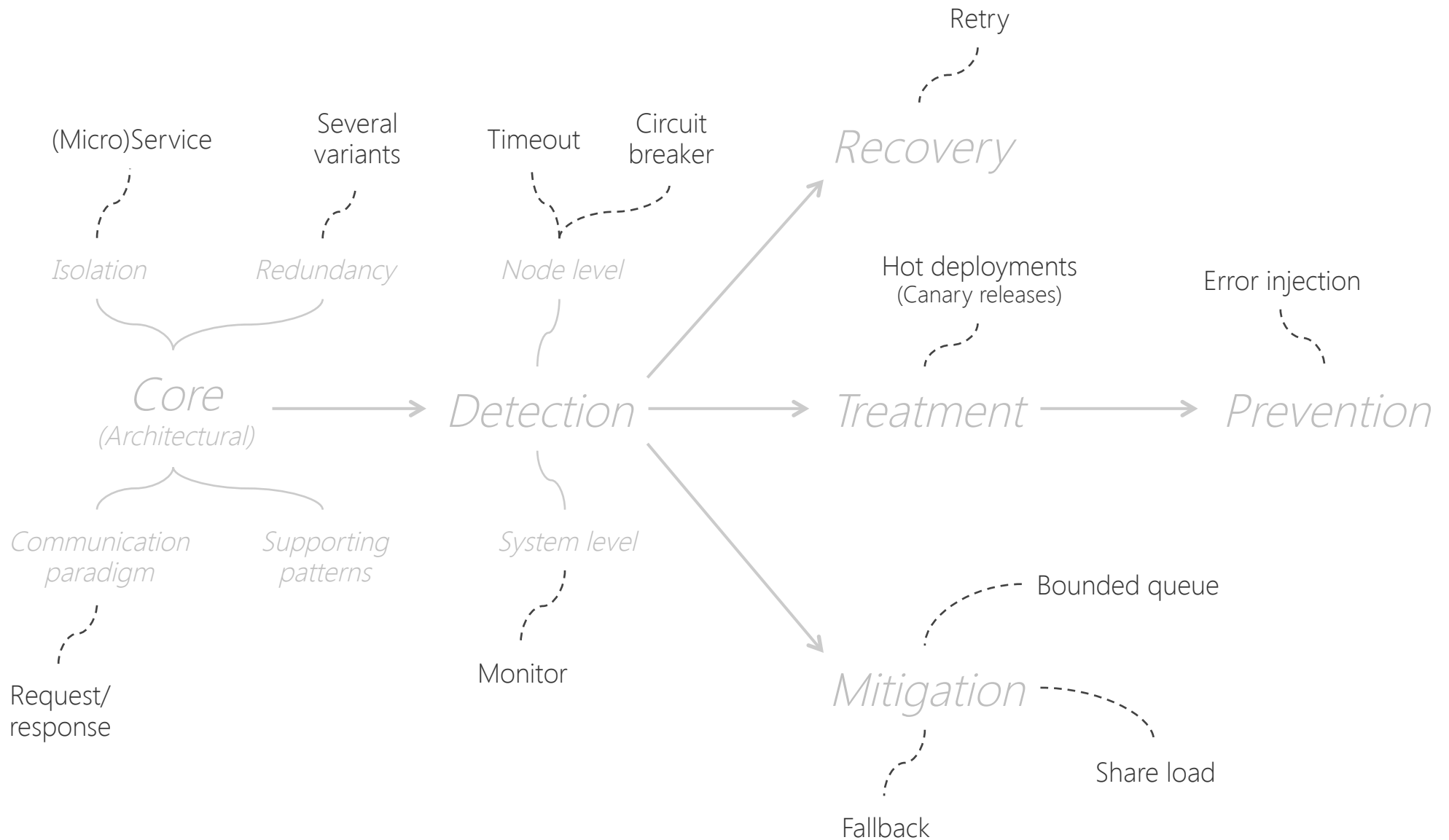


Ongoing

Create and refine system design and functional decomposition. Functionally decouple bulkheads (A good functional decomposition on business level is the prerequisite for an effective resilience)



Example: Erlang (Akka)



Example: Netflix



@ufried

Uwe Friedrichsen

uwe.friedrichsen@codecentric.de



@afitz1712

Ansgar Fitz

ansgar.fitz@codecentric.de

