Towards a resilience pattern language

or how to get resilient software design right

Uwe Friedrichsen (codecentric AG) – Berlin Expert Days – Berlin, 16. September 2016

@ufried

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

Previously on “Resilience” …

Why resilience?


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 todays 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

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

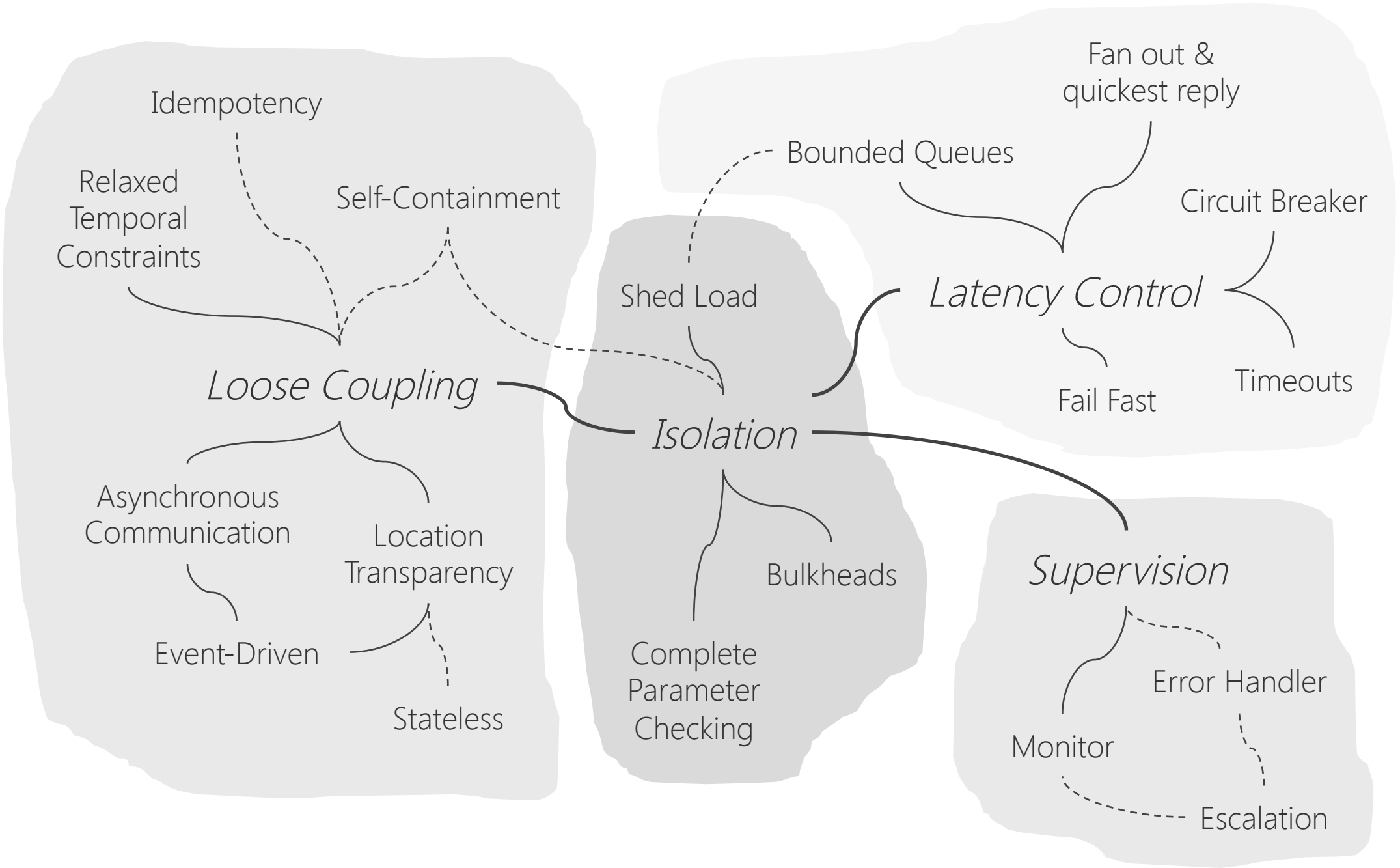
Minimize MTTR

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)

Do not fall for the “100% available” trap!


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

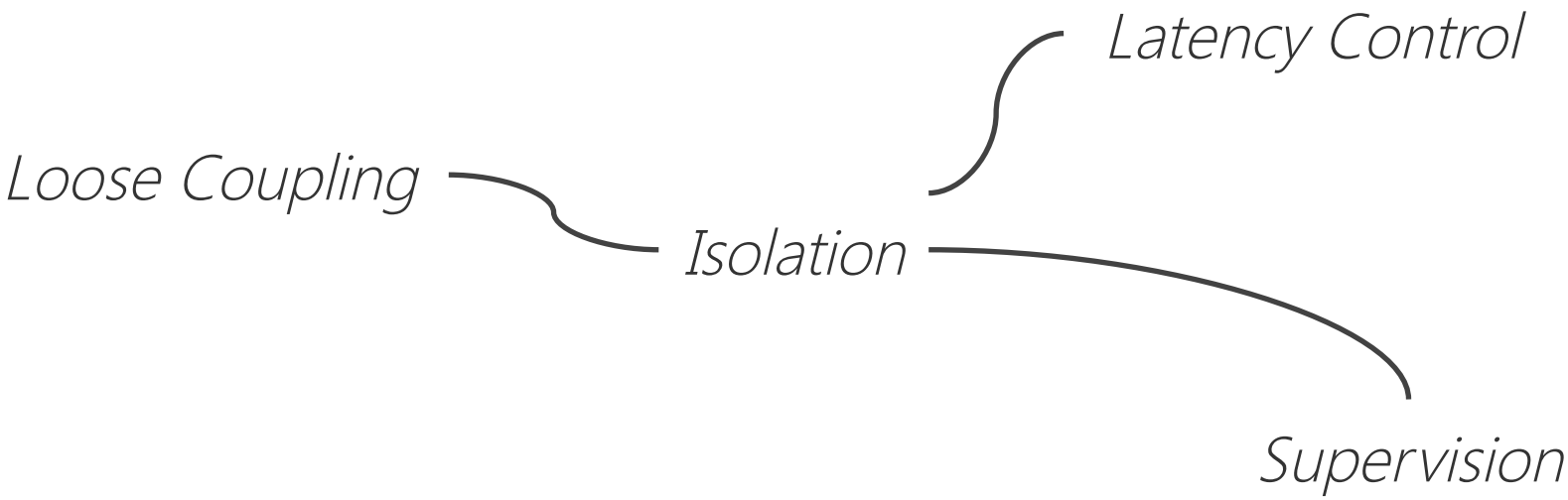
… and there is more

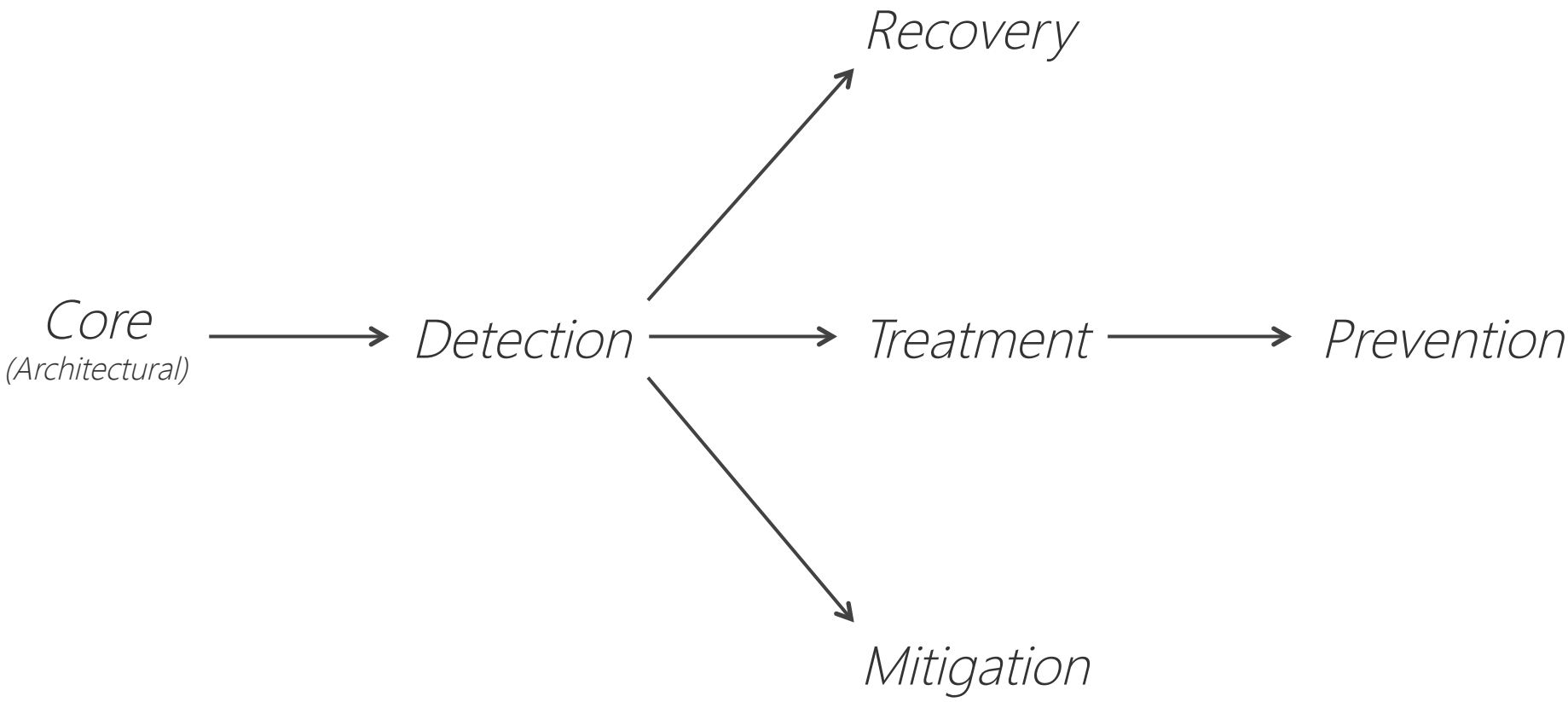
- Recovery & mitigation patterns
- More supervision patterns
- Architectural patterns
- Anti-fragility patterns
- Fault treatment & prevention patterns

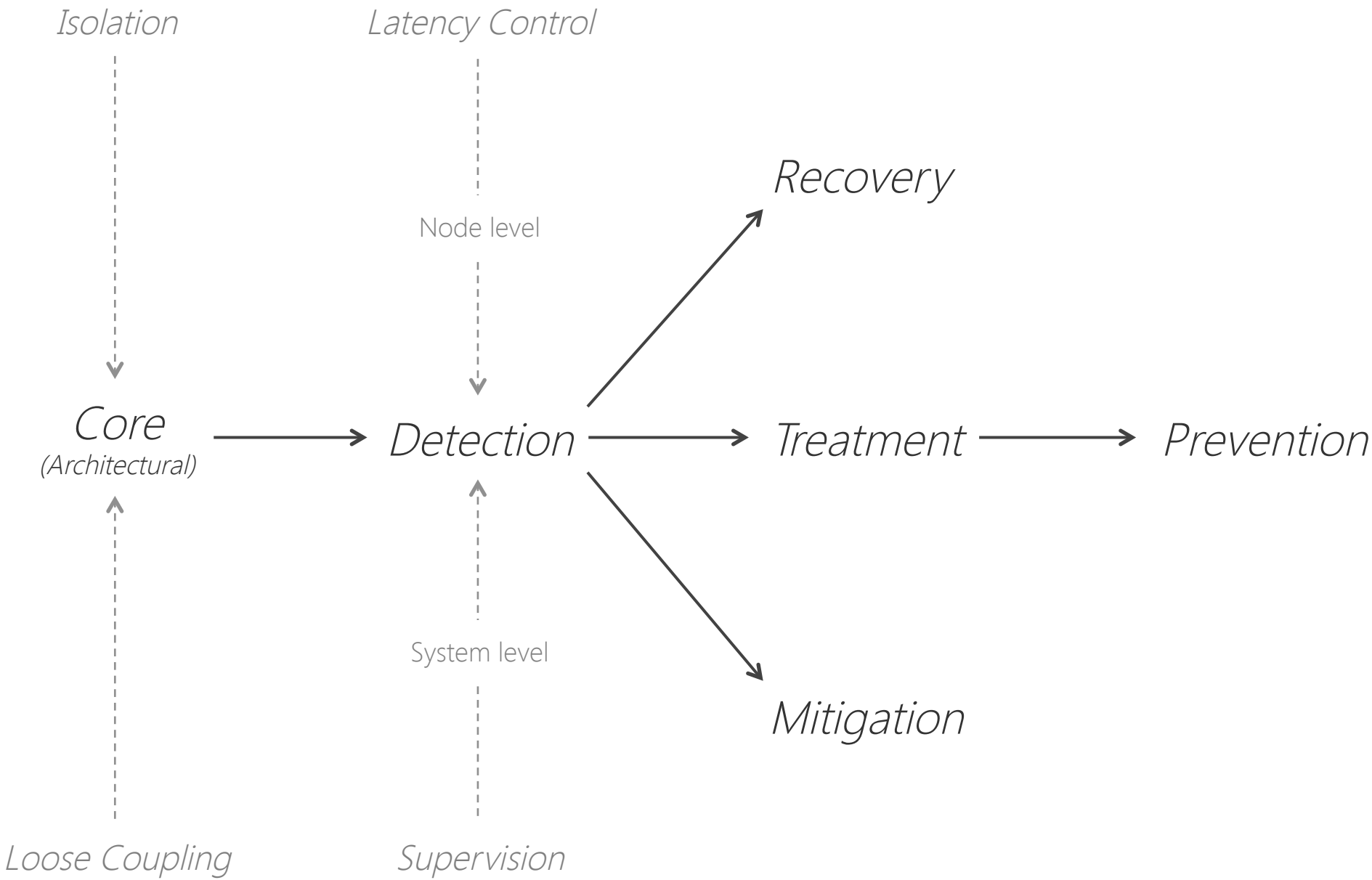
A rich pattern family

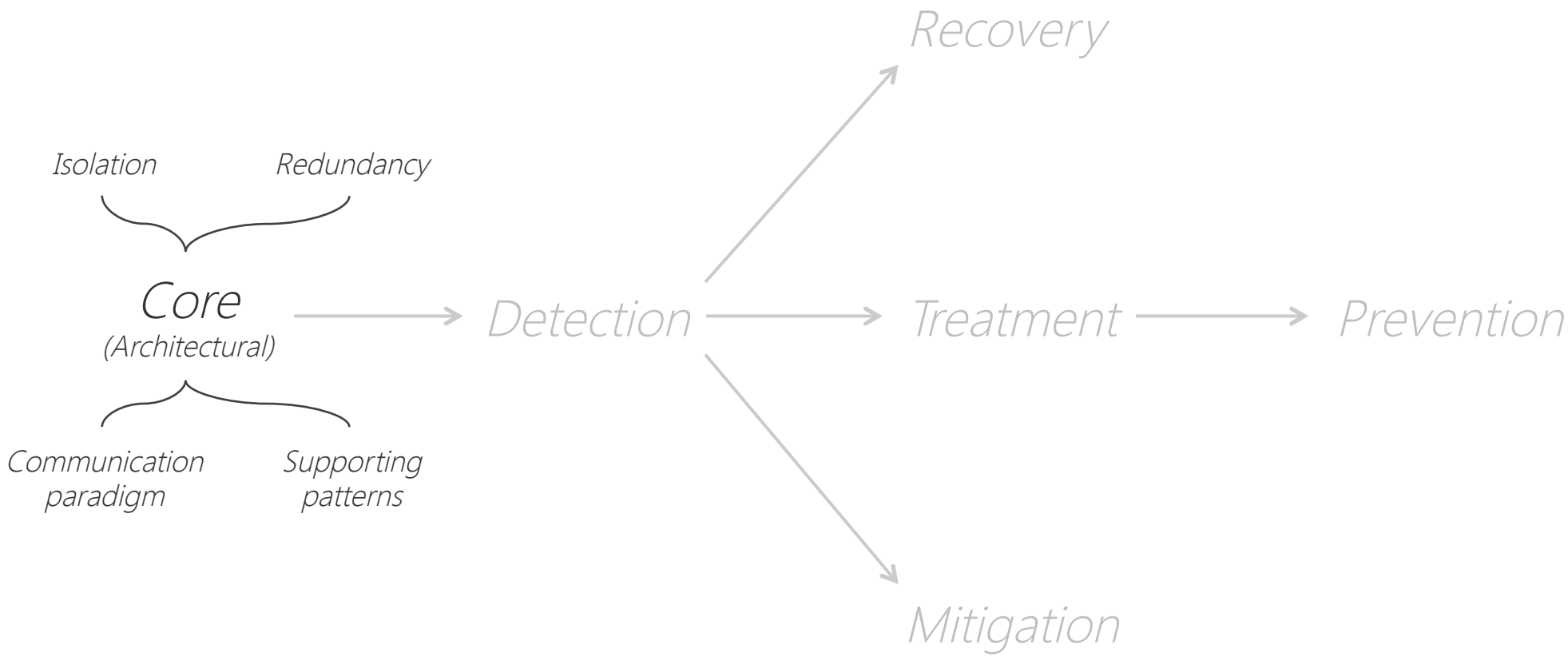
(Title music starts & opening credits shown)

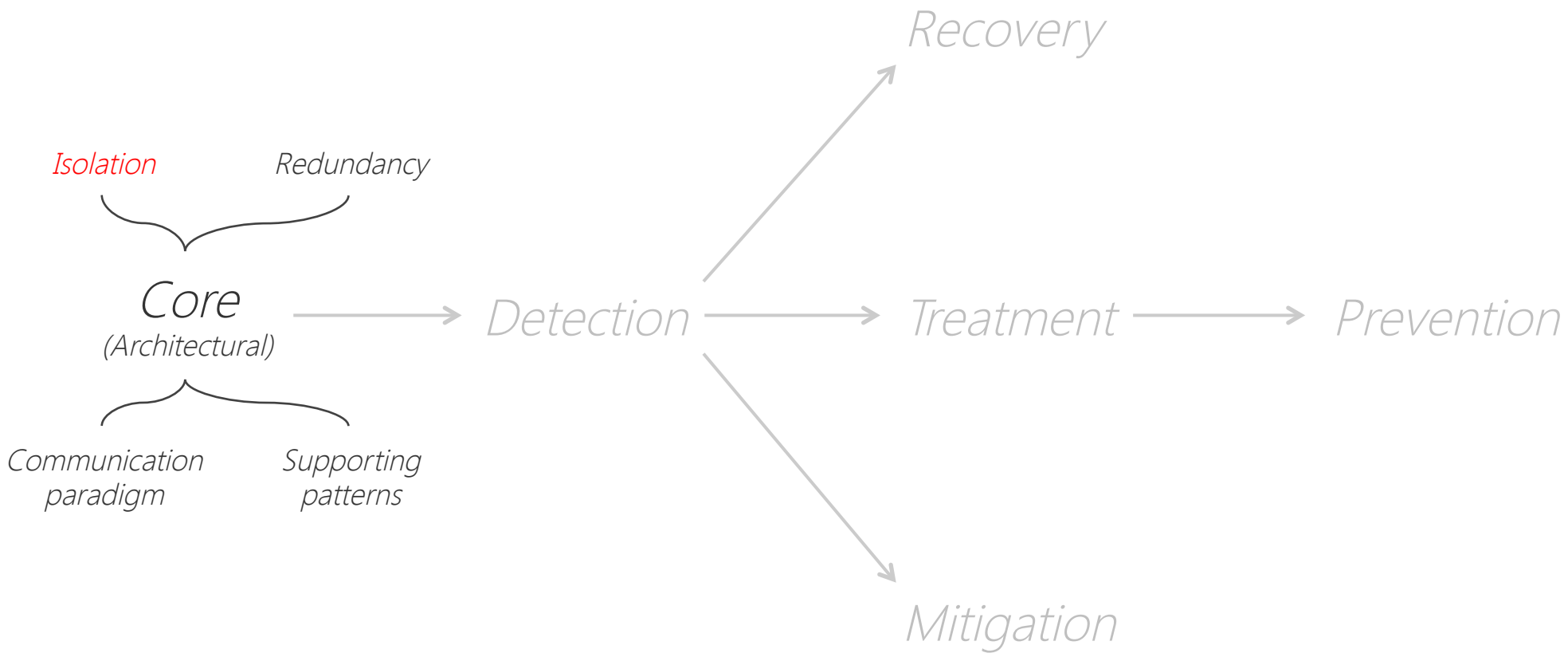
Let’s complete the picture first …

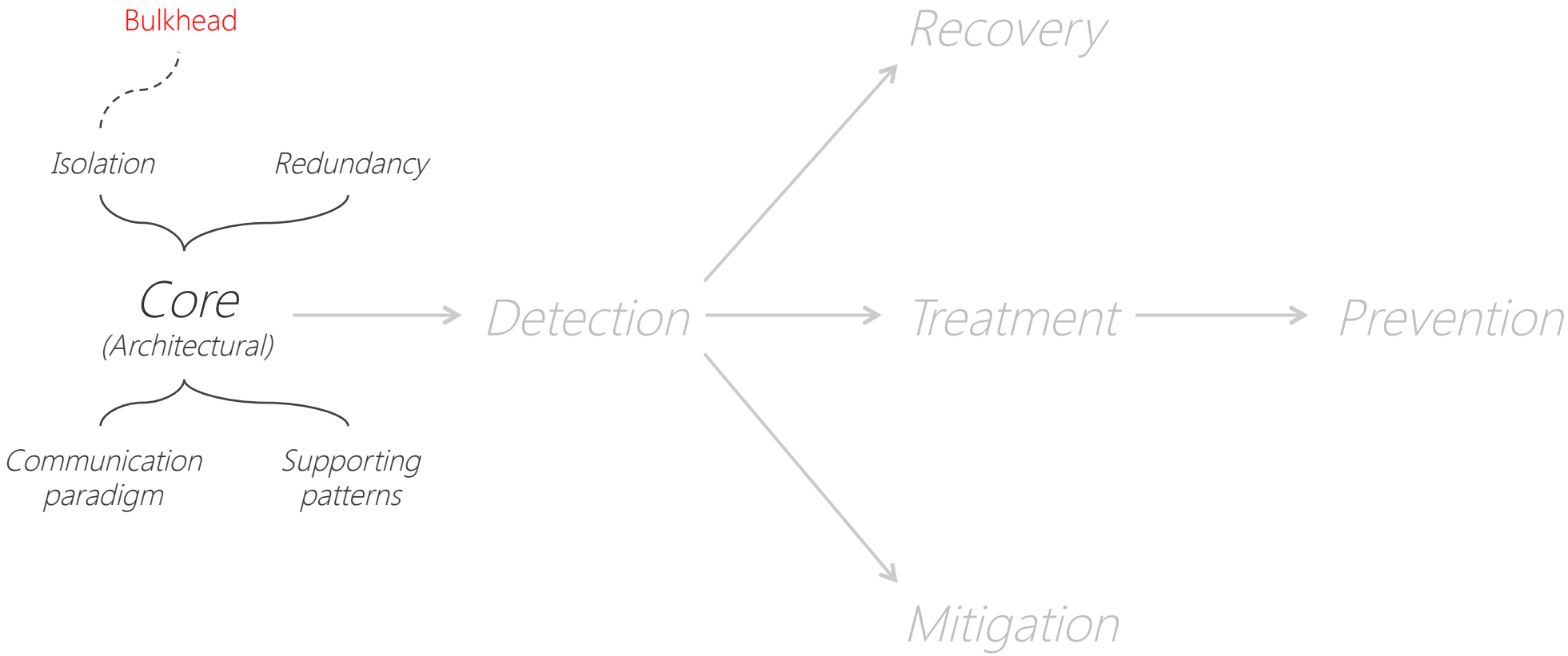

Latency Control
Loose Coupling
Isolation
Supervision


Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation


Isolation
Latency Control
Recovery
Node level
Core
(Architectural)
Detection
Treatment
Prevention
System level
Mitigation
Loose Coupling
Supervision

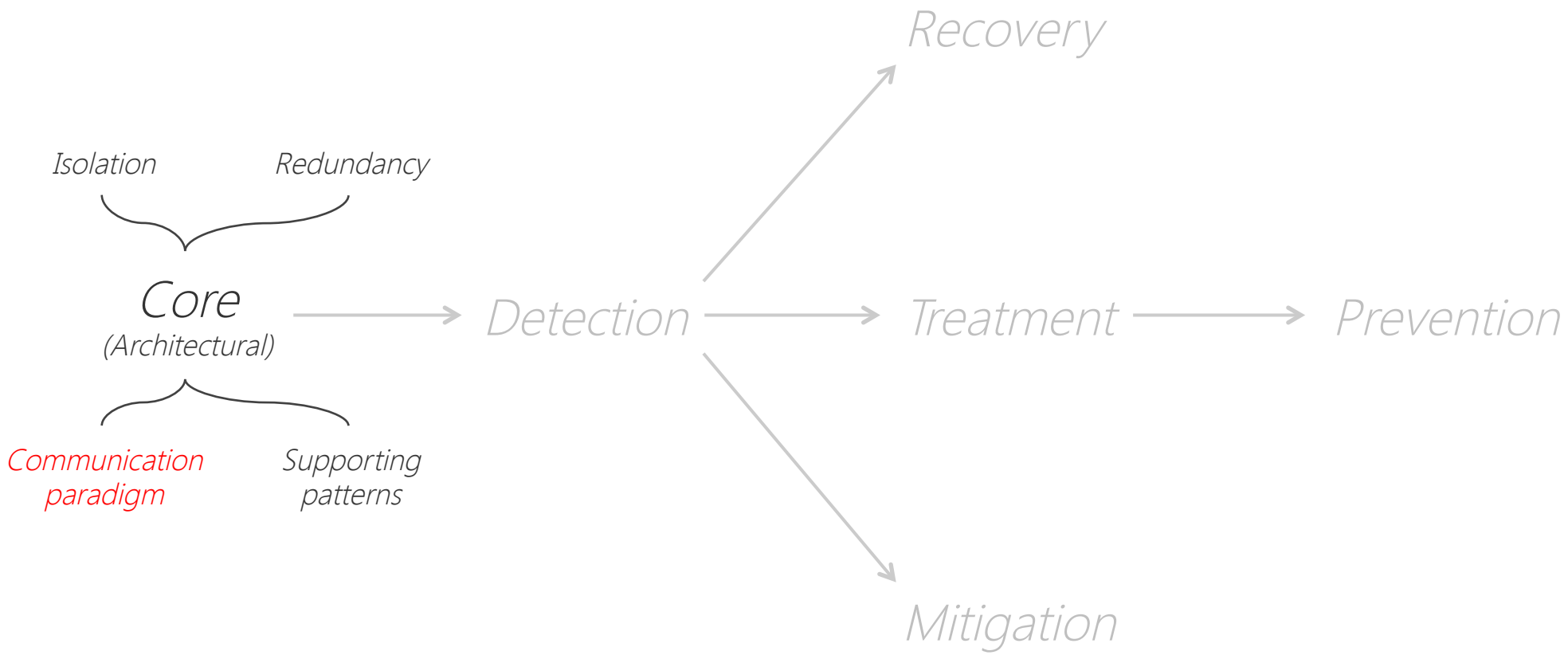

Isolation
Redundancy
Core
(Architectural)
Communication paradigm
Supporting patterns
Detection
Recovery
Treatment
Prevention
Mitigation


Isolation
Redundancy
Core
(Architectural)
Communication paradigm
Supporting patterns
Detection
Recovery
Treatment
Prevention
Mitigation


Bulkhead
Isolation
Redundancy
Core
(Architectural)
Communication paradigm
Supporting patterns
Detection
Recovery
Treatment
Prevention
Mitigation

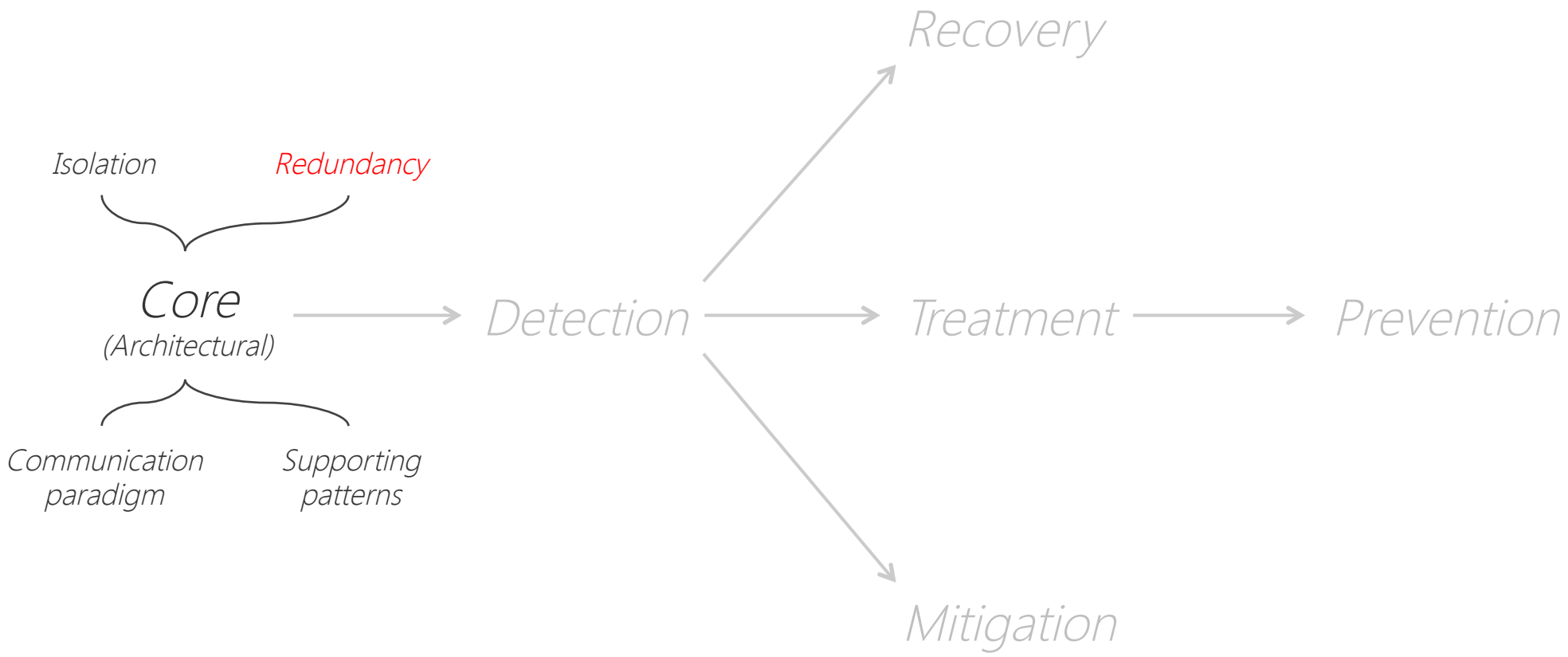
Bulkheads

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


Recovery
Isolation
Redundancy
Core
(Architectural)
Detection
Treatment
Prevention
Communication paradigm
Supporting patterns
Mitigation

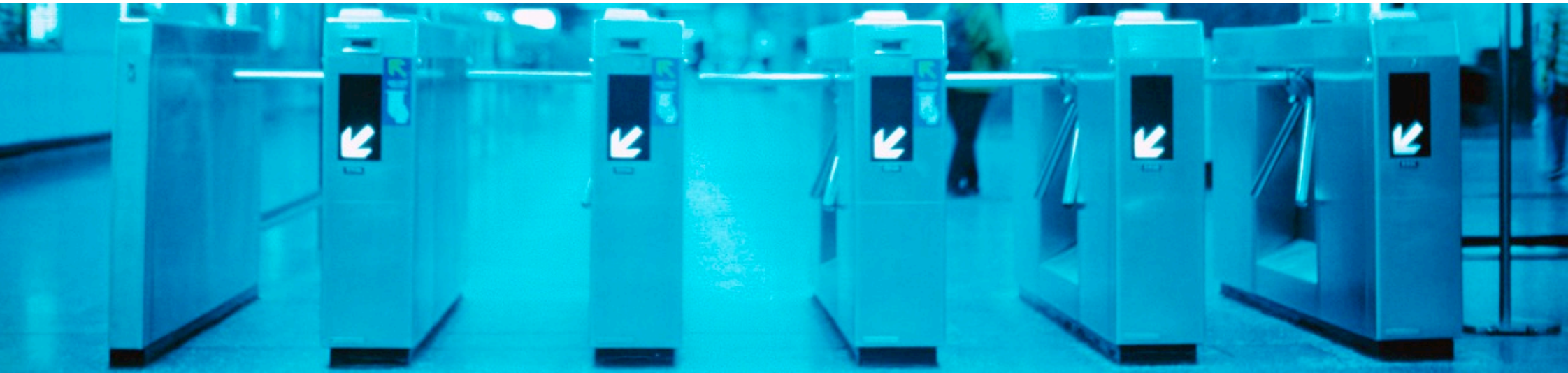
Communication paradigm

- Request-response <-> messaging <-> 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


Isolation
Redundancy
Core
(Architectural)
Communication paradigm
Supporting patterns
Detection
Recovery
Treatment
Prevention
Mitigation

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

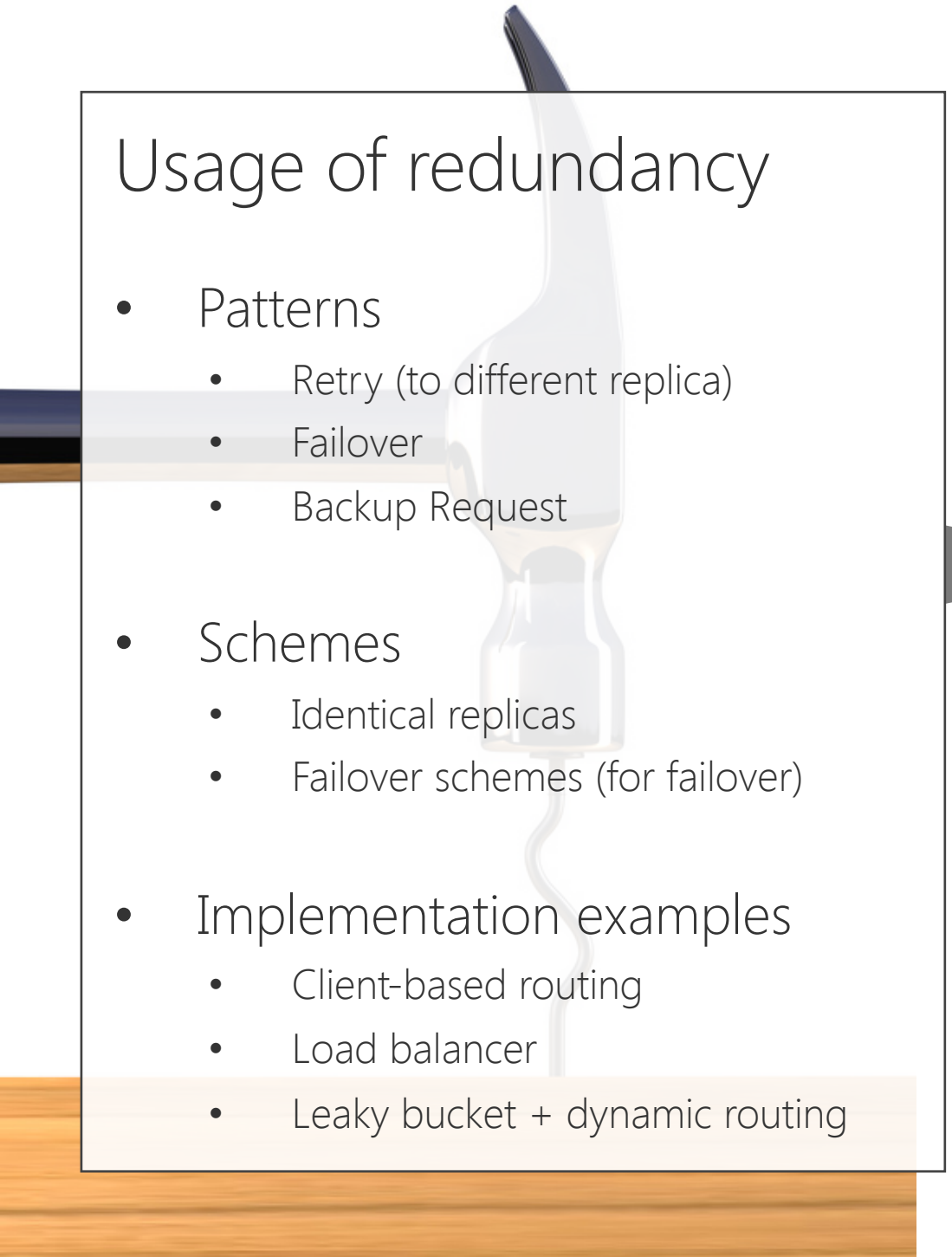
- Crash failure
- Omission failure
- Timing failure
- Response failure
- Byzantine failure

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

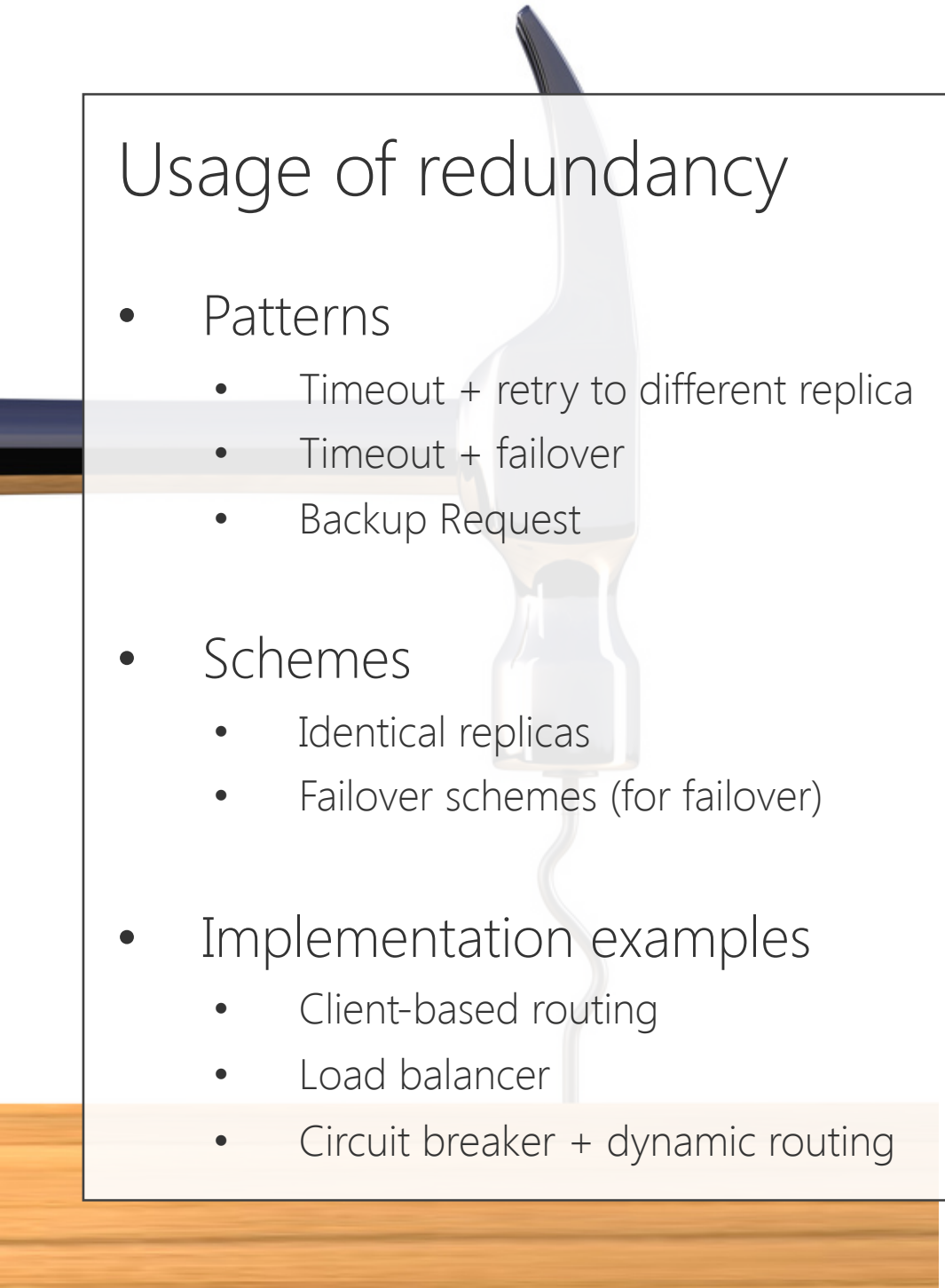


Usage of redundancy

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

Failure types

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

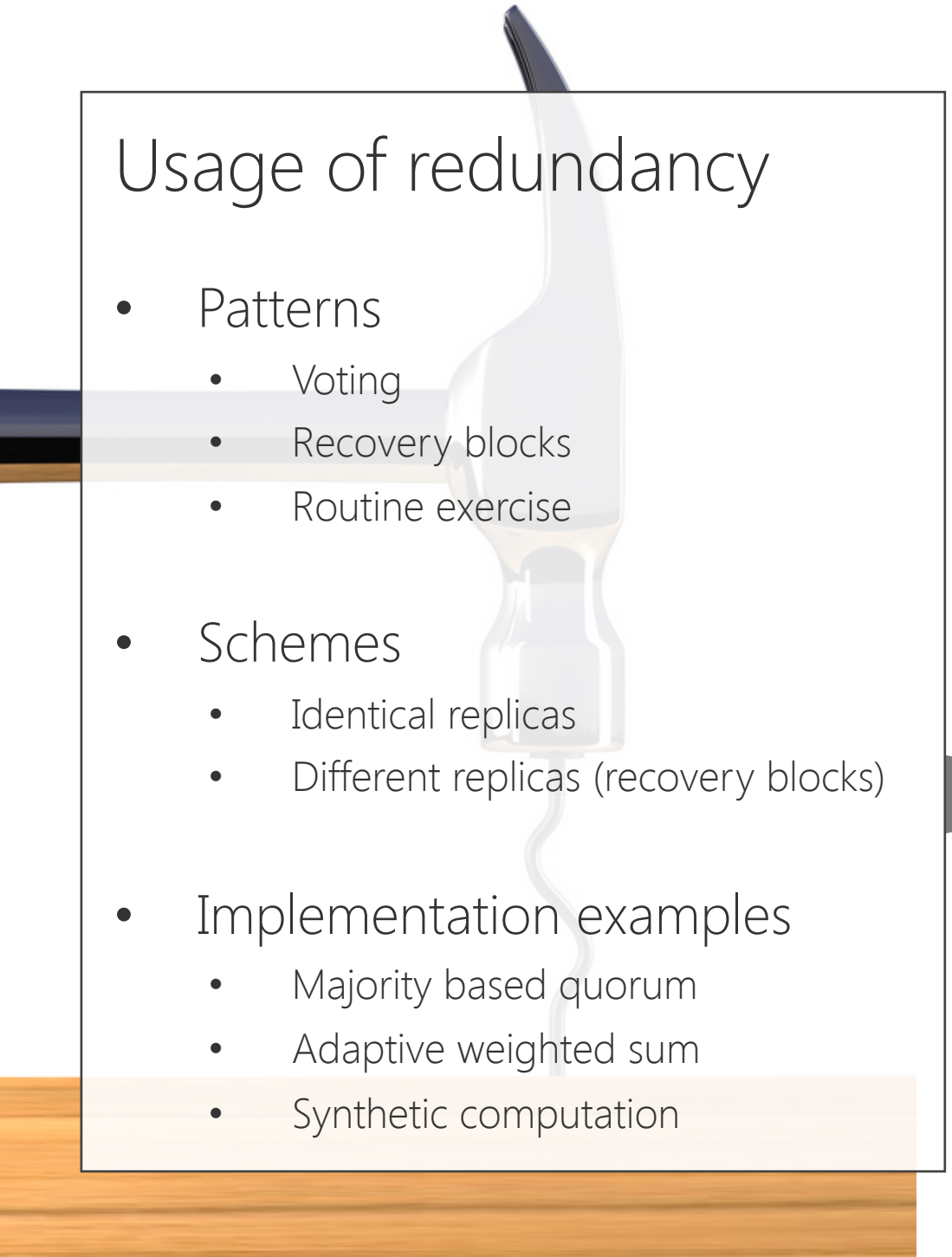


Usage of redundancy

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

Failure types

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

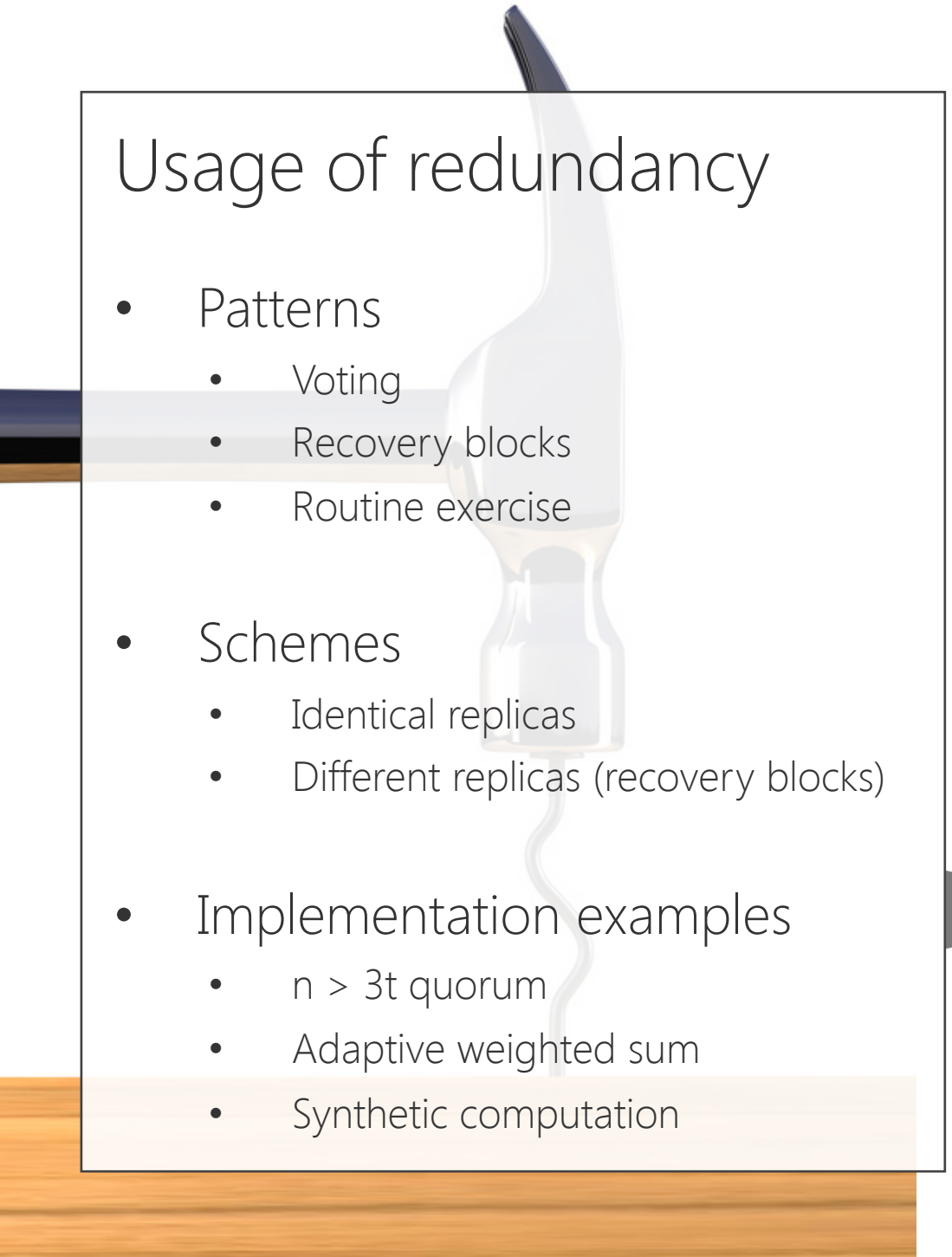


Usage of redundancy

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

Failure types

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

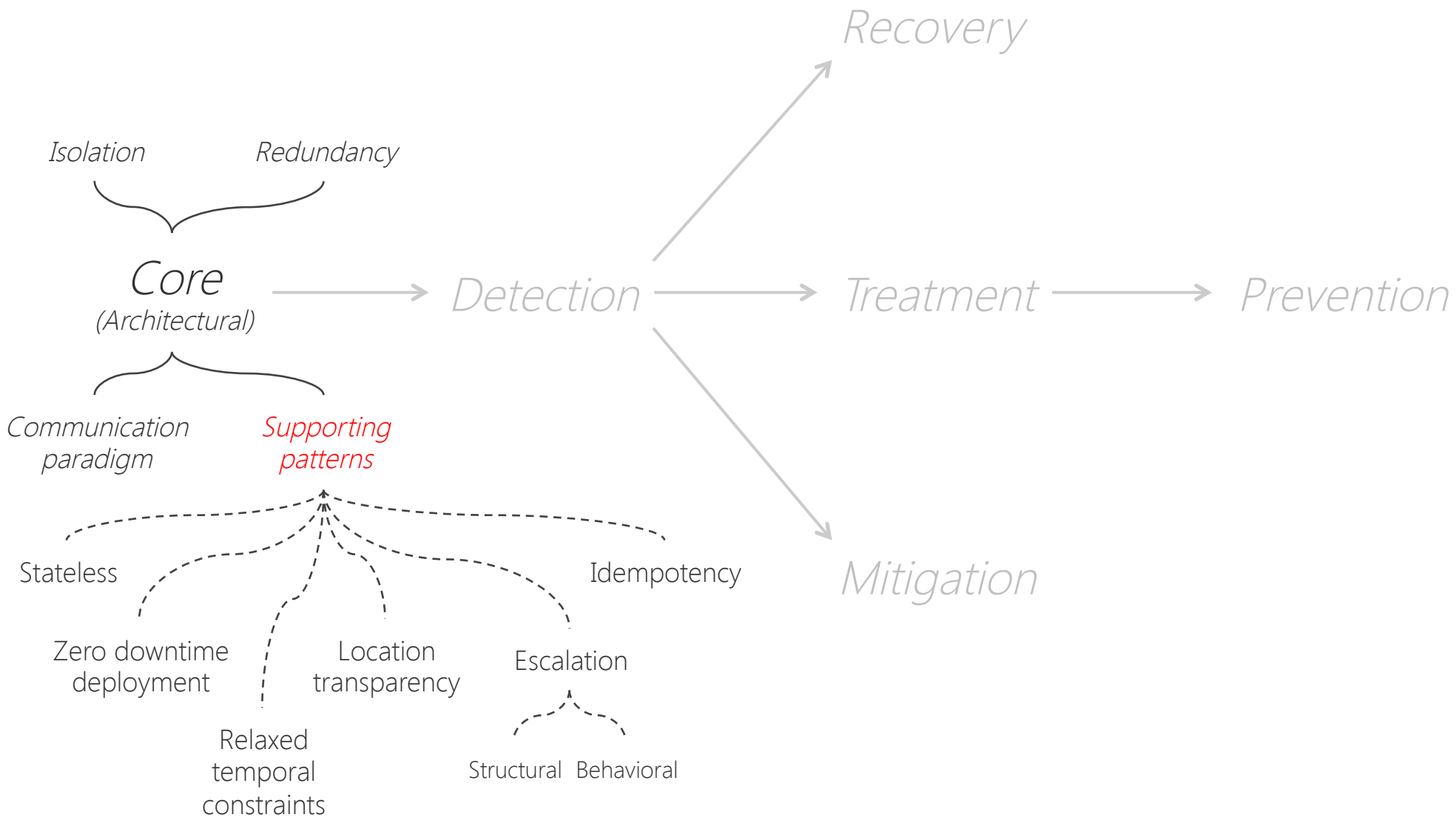


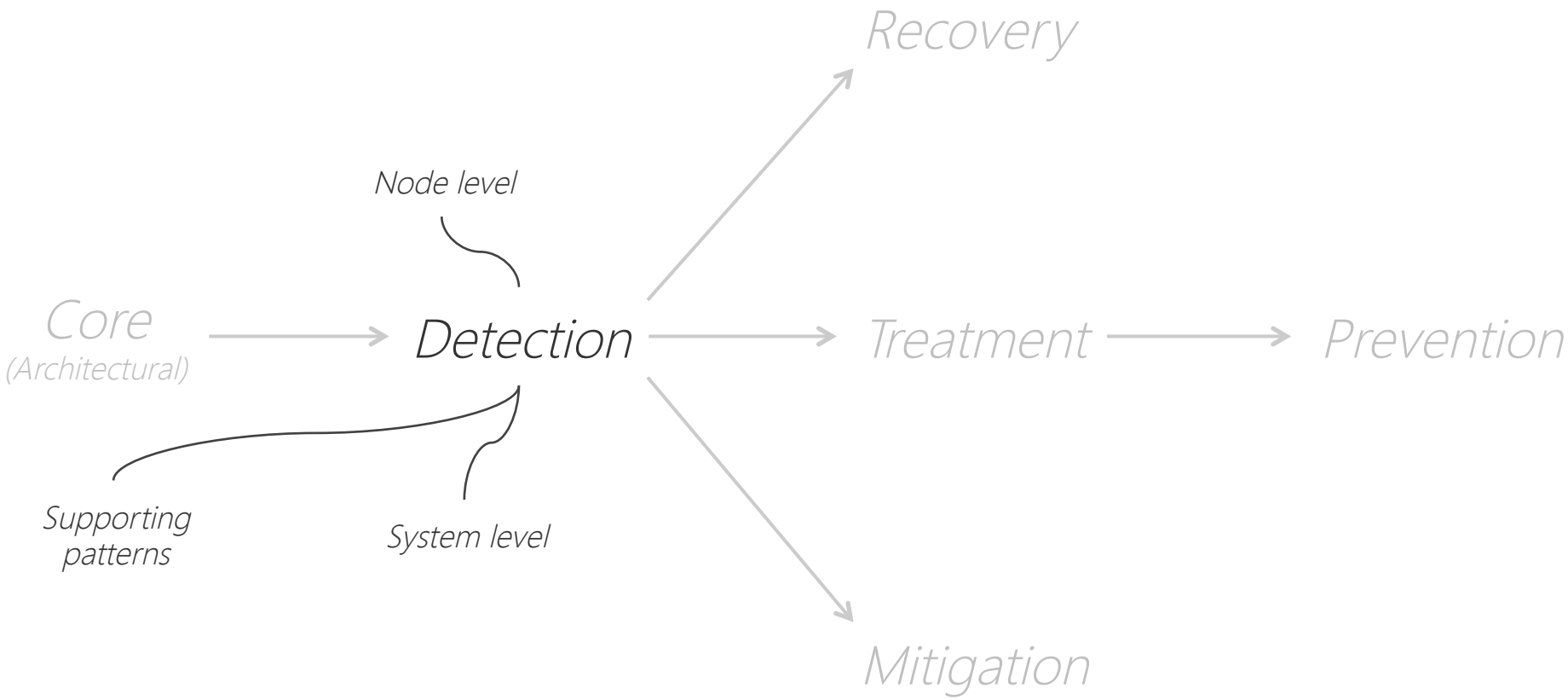
Usage of redundancy

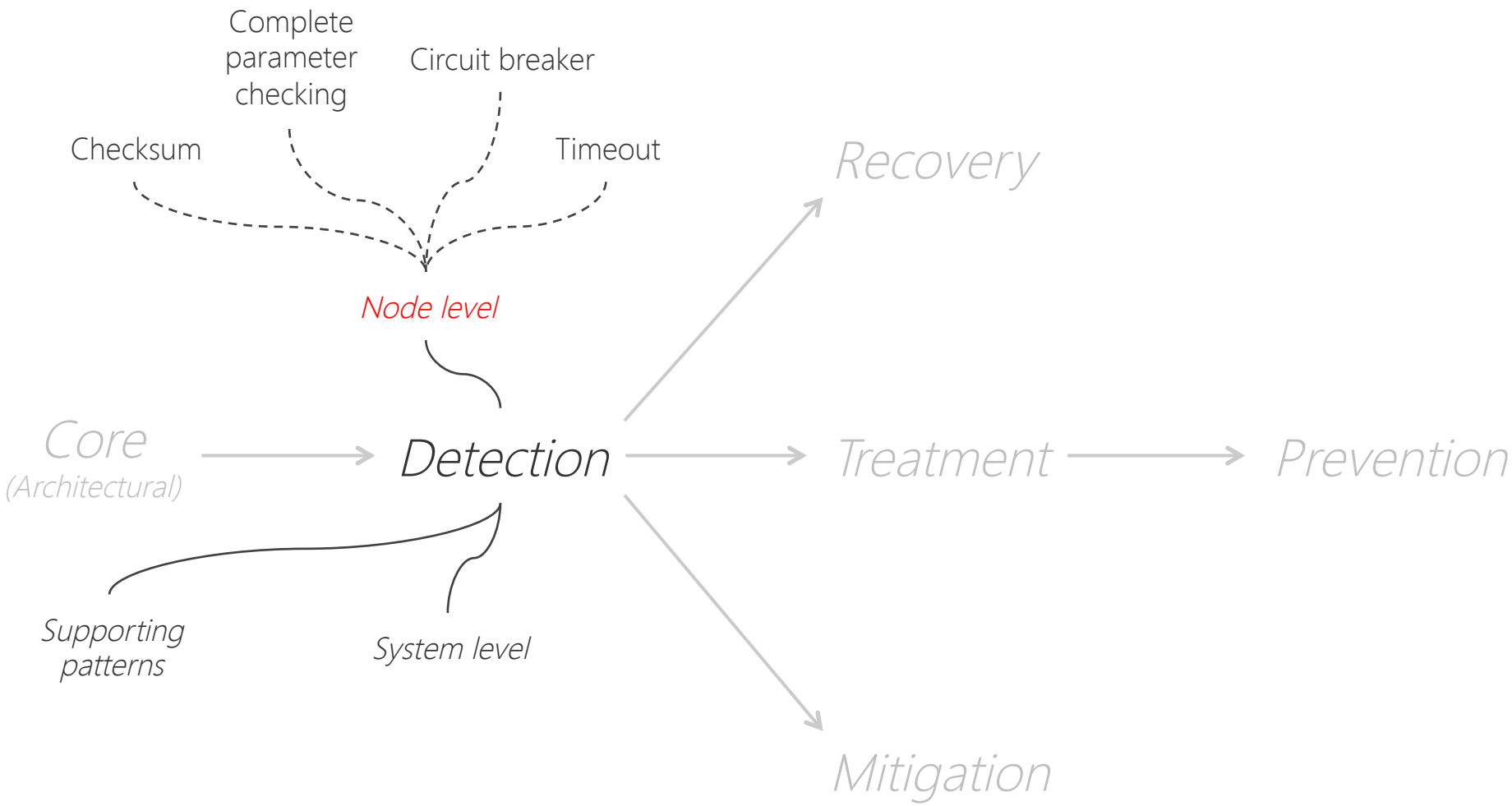
- Patterns
  - Voting
  - Recovery blocks
  - Routine exercise
- Schemes
  - Identical replicas
  - Different replicas (recovery blocks)
- Implementation examples
  - n > 3t quorum
  - Adaptive weighted sum
  - Synthetic computation

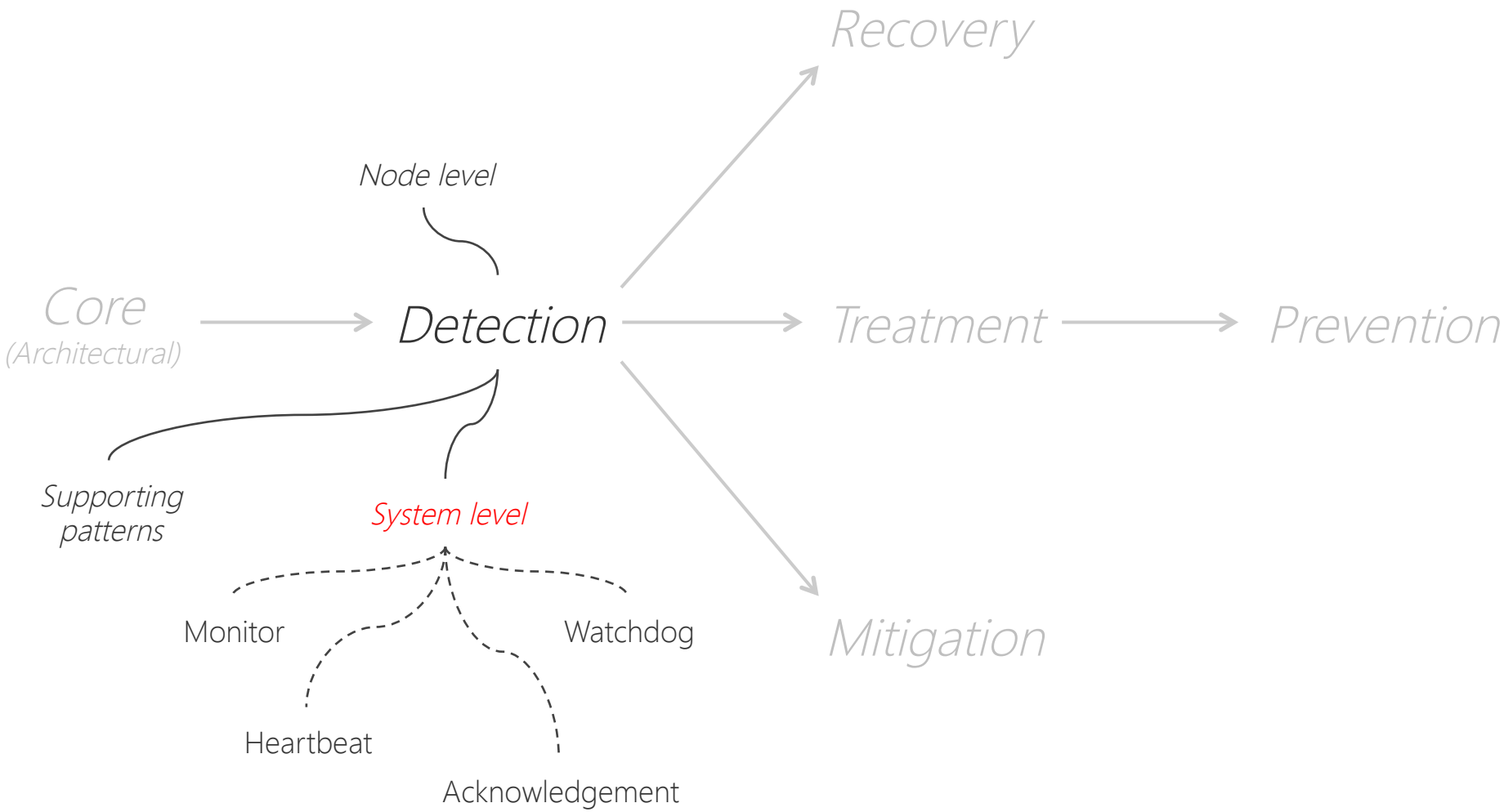
Failure types

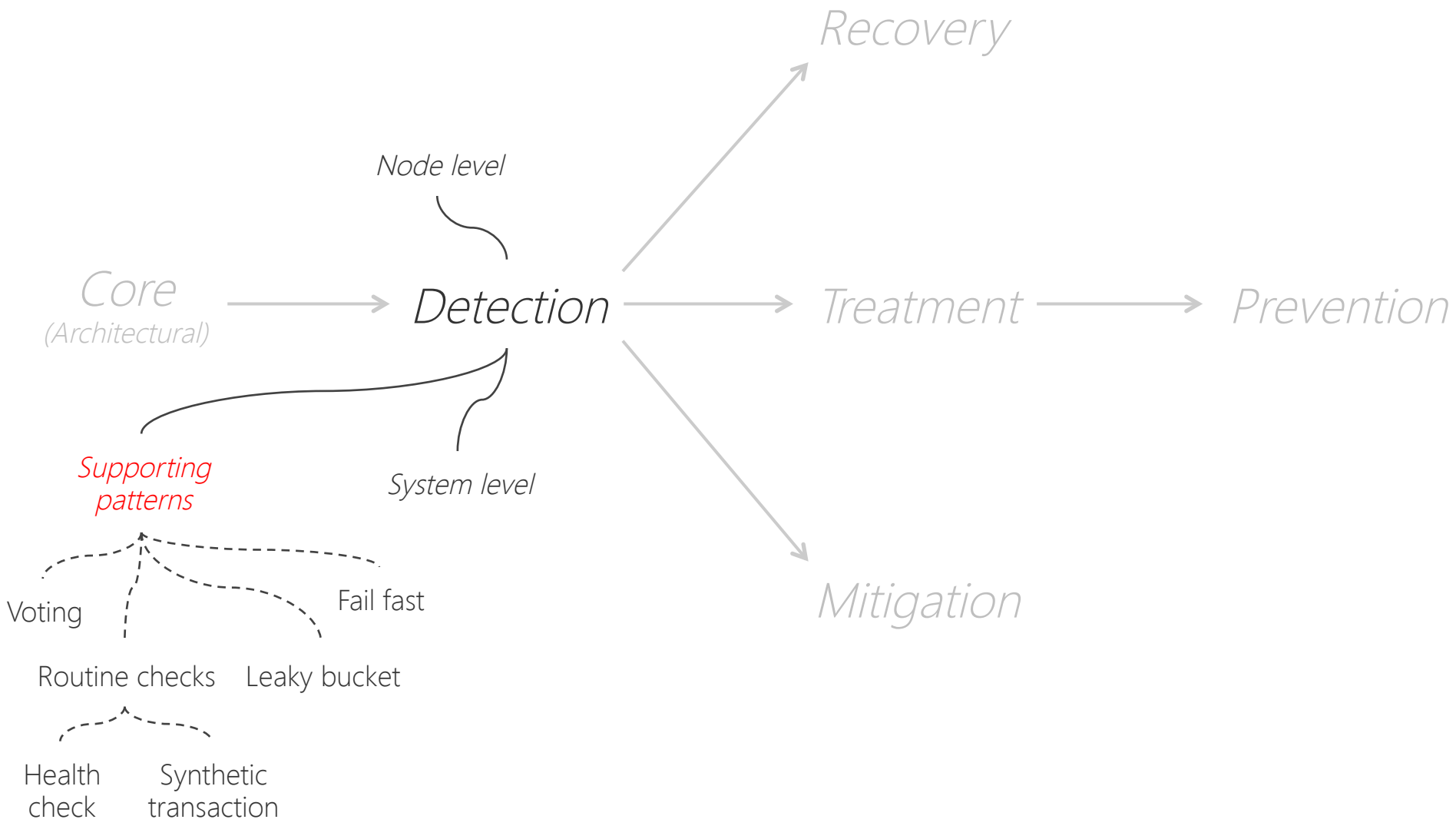
- Crash failure
- Omission failure
- Timing failure
- Response failure
- Byzantine failure

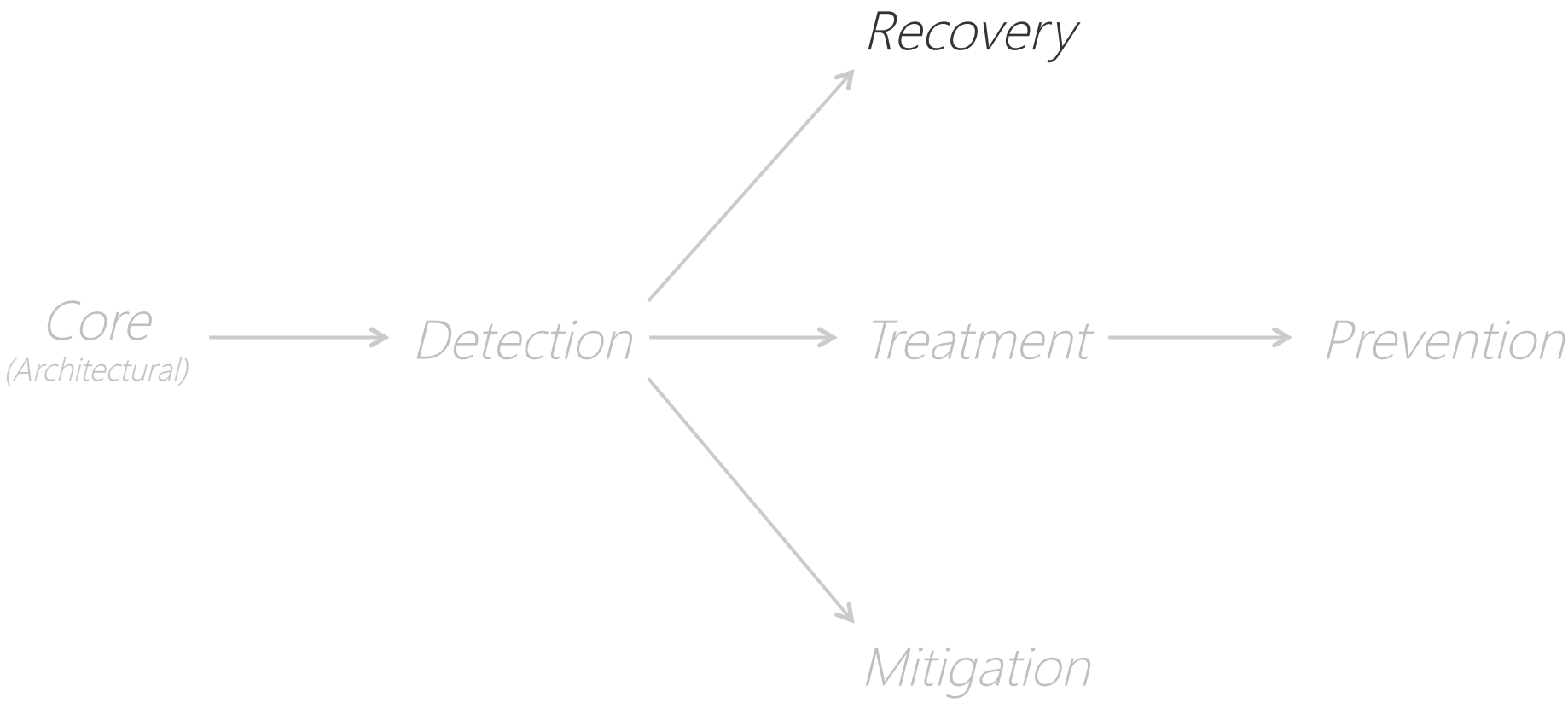

Isolation
Redundancy
Core
(Architectural)
Communication paradigm
Supporting patterns
Stateless
Idempotency
Zero downtime deployment
Location transparency
Escalation
Relaxed temporal constraints
Structural
Behavioral
Detection
Recovery
Treatment
Mitigation
Prevention

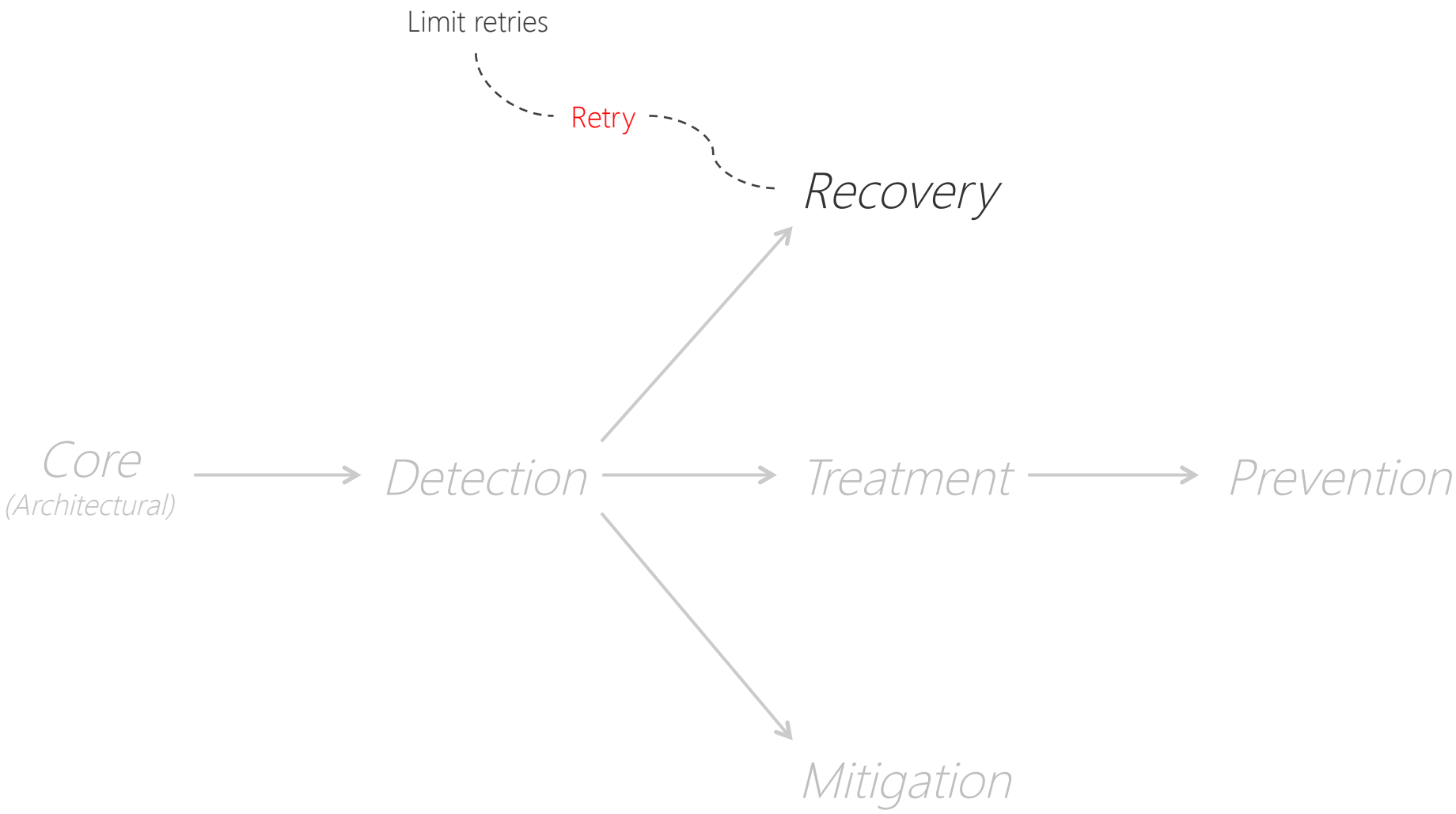

Recovery
Node level
Core
(Architectural)
Detection
Treatment
Prevention
Supporting
patterns
System level
Mitigation


Complete parameter checking
Circuit breaker
Checksum
Timeout
Node level
Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Supporting patterns
System level
Mitigation


Recovery
Node level
Core
(Architectural)
Detection
Treatment
Prevention
Supporting
patterns
System level
Monitor
Watchdog
Mitigation
Heartbeat
Acknowledgement


Recovery
Node level
Core
(Architectural)
Detection
Treatment
Prevention
Supporting
patterns
System level
Mitigation
Voting
Fail fast
Routine checks
Leaky bucket
Health
check
Synthetic
transaction


Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation

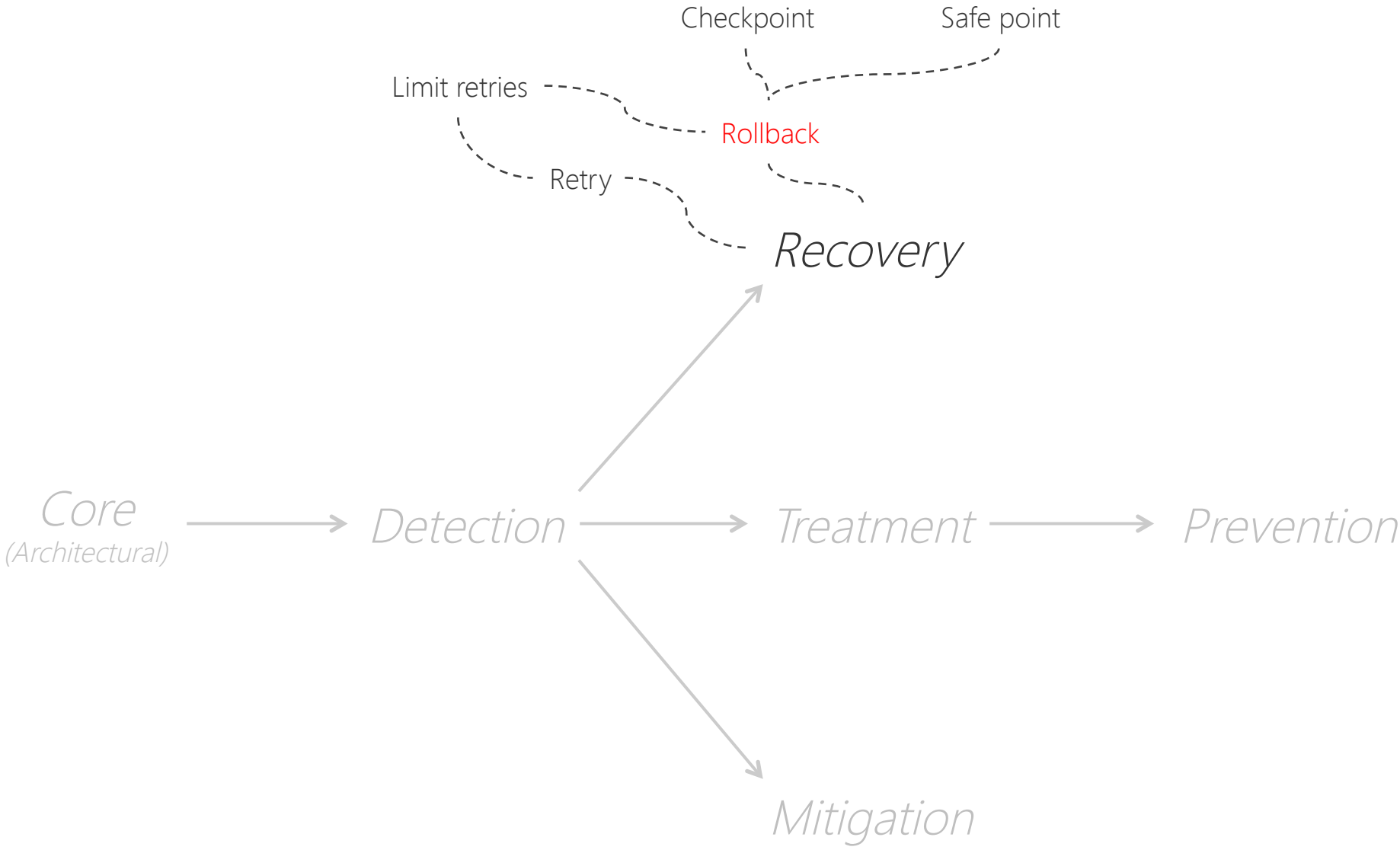

Limit retries
Retry
Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation

Retry

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

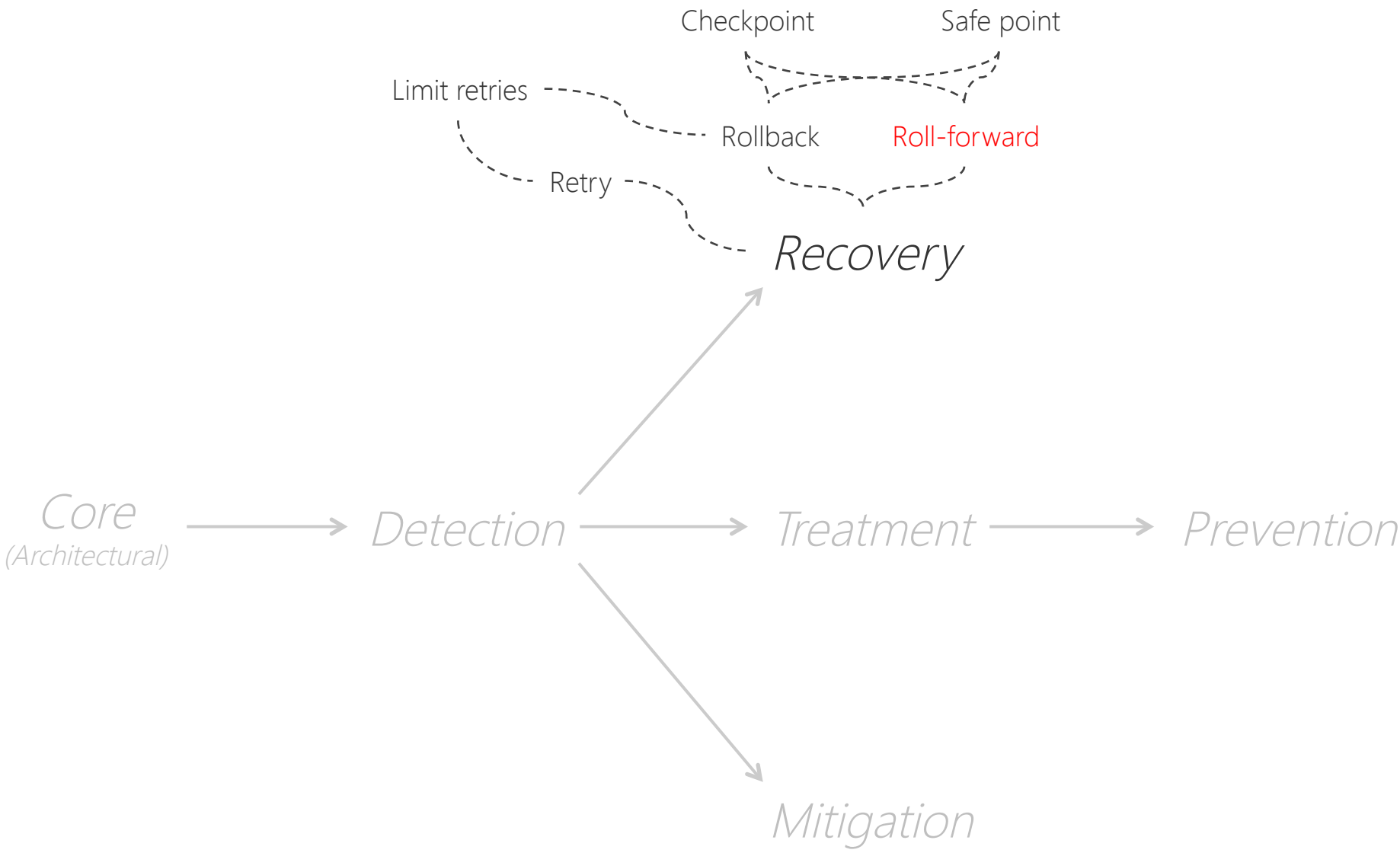
Retry example

```
// 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(...);
```


Checkpoint
Safe point
Limit retries
Rollback
Retry
Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation

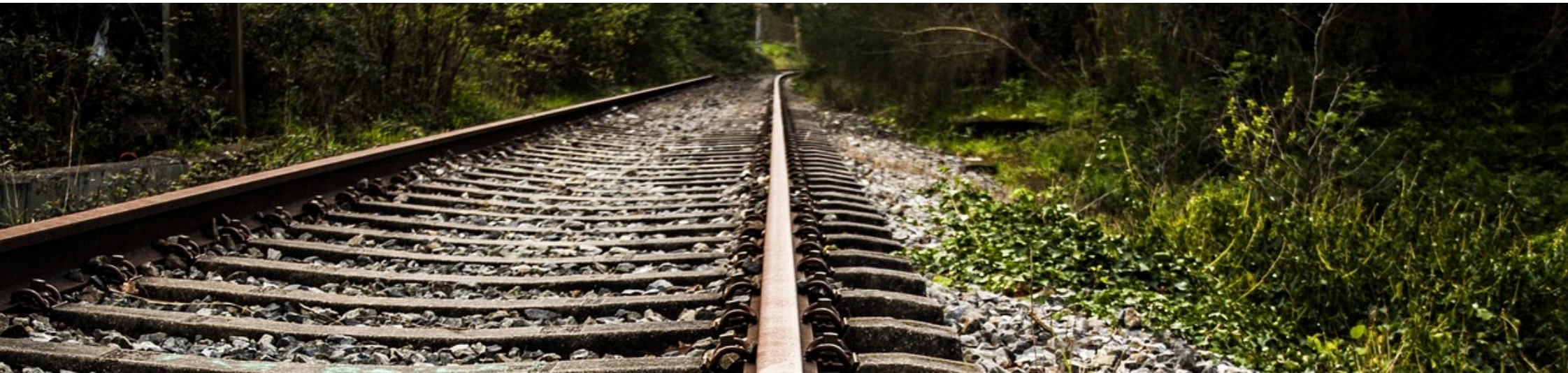
Rollback

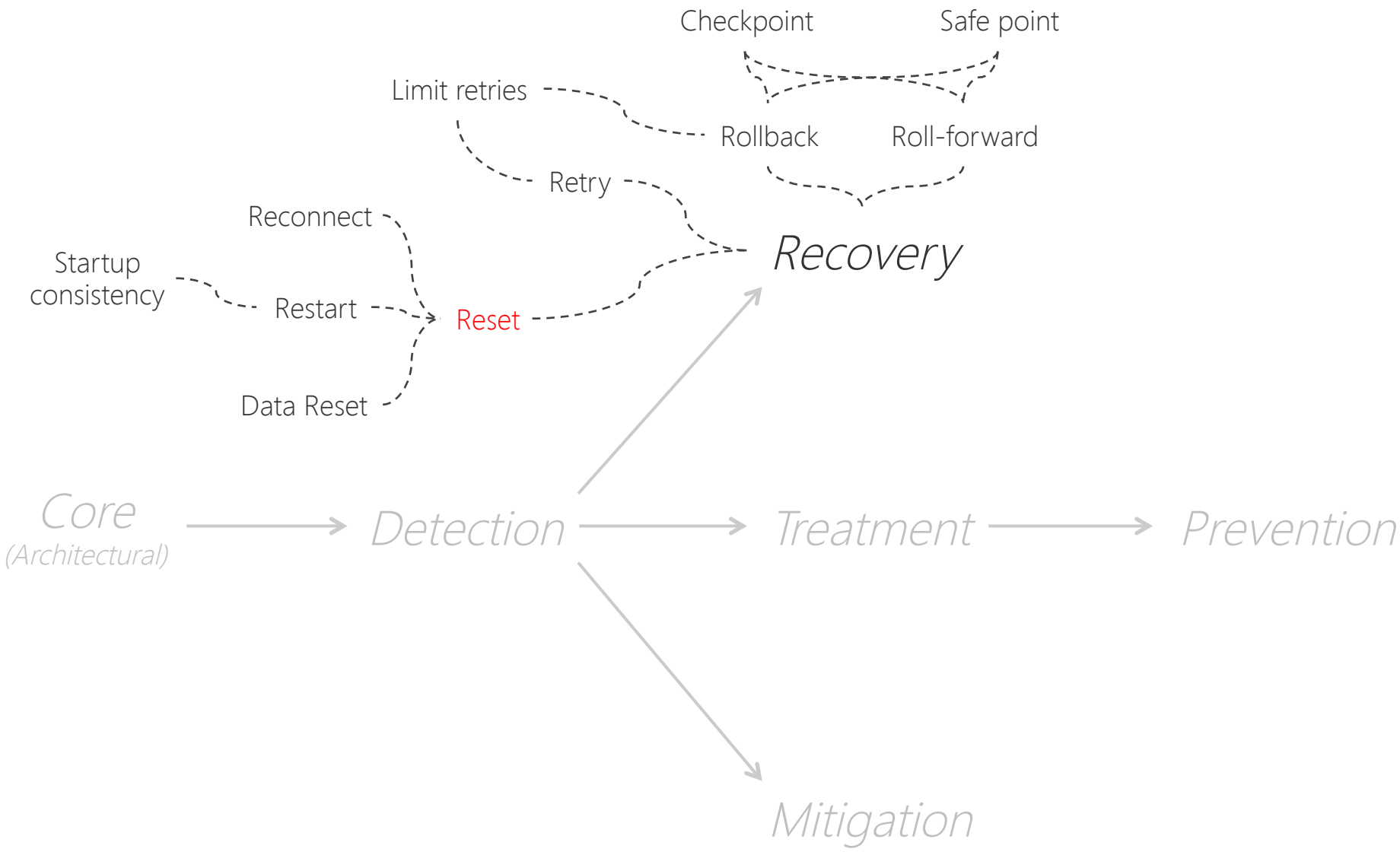
- Roll back state and/or execution path to a defined safe state
- Recover from internal errors caused by external failures
- Use checkpoints and safe points to provide safe rollback points
- Limit retries to avoid recurring errors


Checkpoint
Safe point
Limit retries
Rollback
Roll-forward
Retry
Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation

Roll-forward

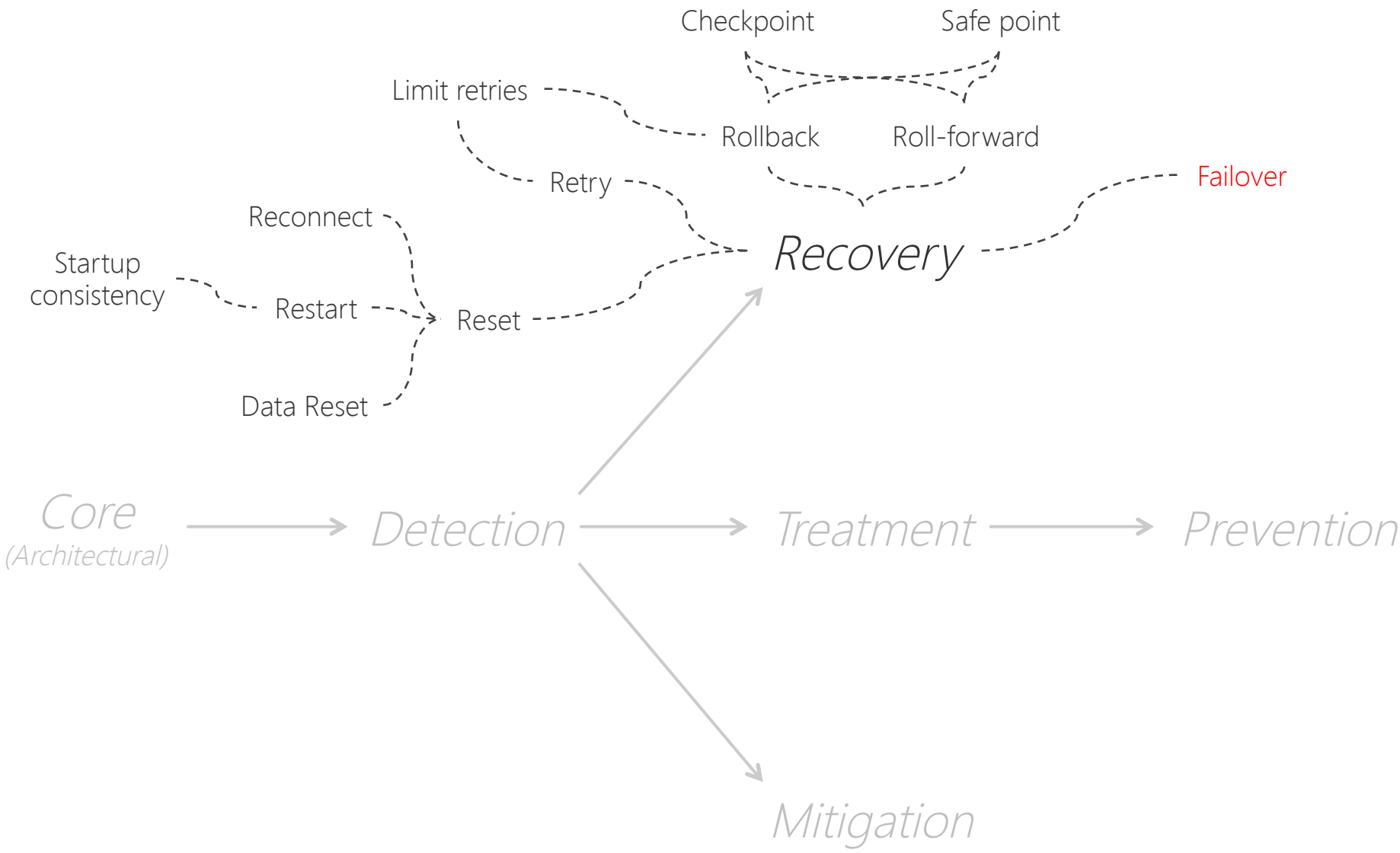
- Advance execution past the point of error
- Often used as escalation if retry or rollback do not succeed
- Not applicable if skipped activity is essential
- Use checkpoints and safe points to provide safe roll-forward points




Checkpoint
Safe point
Limit retries
Rollback
Roll-forward
Retry
Reconnect
Recovery
Startup consistency
Restart
Reset
Data Reset
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation

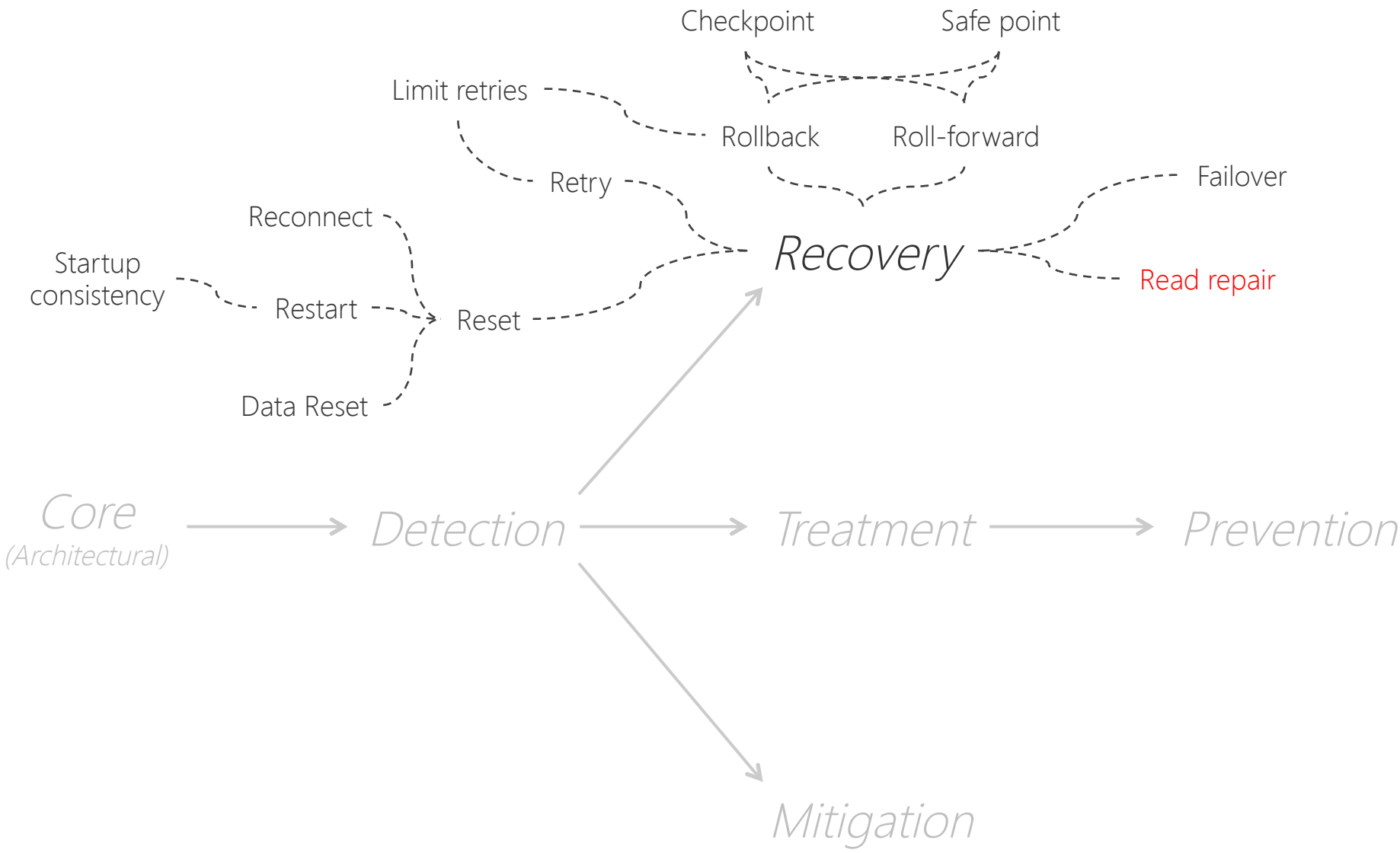
Reset

- Often used as radical escalation if all other measures failed
- Restart service – do not forget to provide a consistent startup state
- Reset data to a guaranteed consistent state if nothing else helps
- Sometimes simply trying to reconnect helps (often forgotten)


Checkpoint
Safe point
Limit retries
Rollback
Roll-forward
Retry
Failover
Reconnect
Recovery
Startup consistency
Restart
Reset
Data Reset
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation

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


Checkpoint
Safe point
Limit retries
Rollback
Roll-forward
Retry
Failover
Reconnect
Recovery
Startup consistency
Restart
Reset
Read repair
Data Reset
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation

Read repair

- Handle response failures due to relaxed temporal constraints
- Requires redundancy – trades resources for availability
- Decides correct state based on conflicting siblings
- Often implemented in NoSQL databases (but not always accessible)

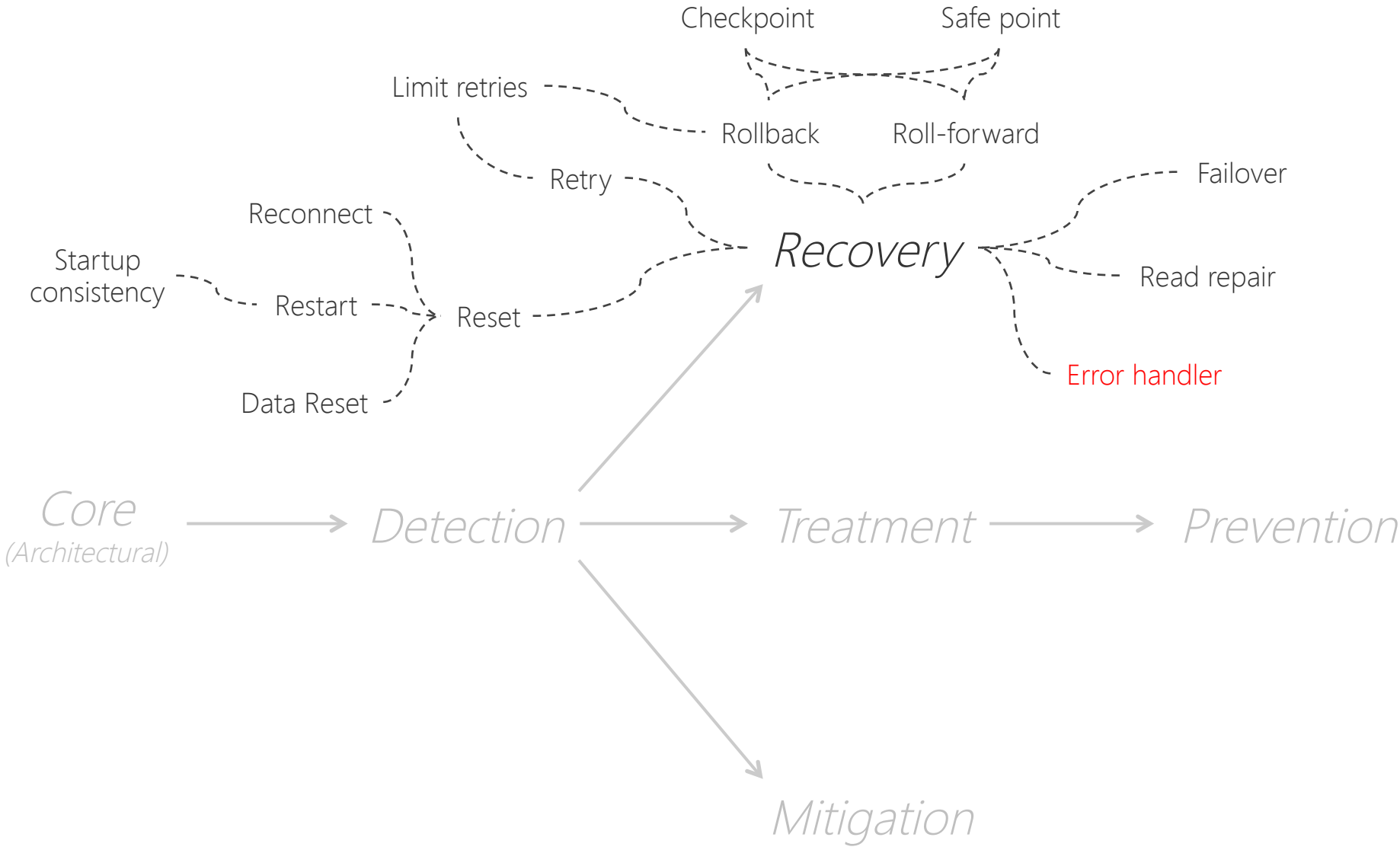
Read repair example (Riak, Java) 1/2

```
public class FooResolver implements ConflictResolver<Foo> {
    @Override
    public Foo resolve(List<Foo> siblings) {
        // Insert your sibling resolution logic here
    }
}
```

```
public class Buddy {
    public String name;
    public Set<String> nicknames;

    public Buddy(String name, Set<String> nicknames) {
        this.name = name;
        this.nicknames = nicknames;
    }
}
```

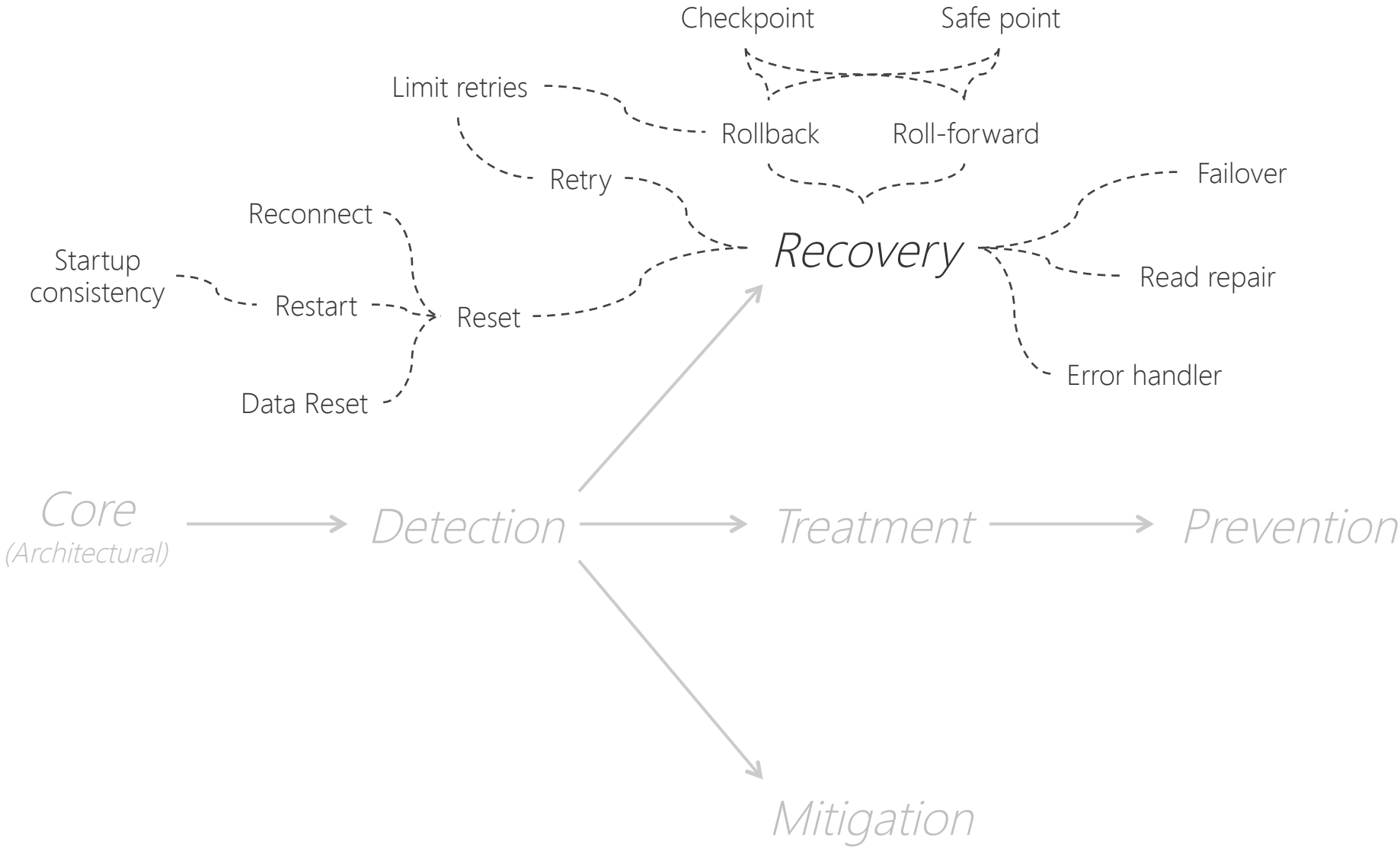
Read repair example (Riak, Java) 2/2

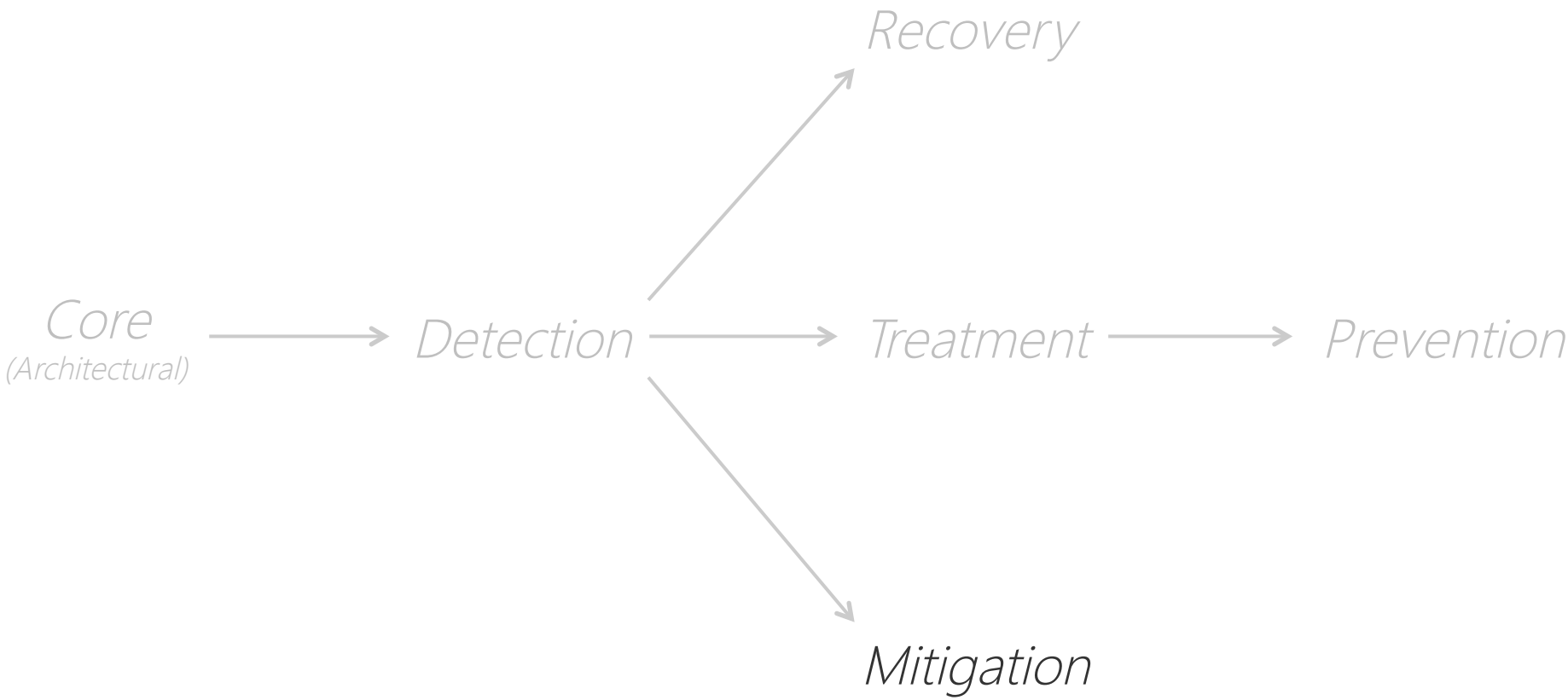
```
public class BuddyResolver implements ConflictResolver<Buddy> {
    @Override
    public Buddy resolve(List<Buddy> siblings) {
        if (siblings.size == 0) {
            return null;
        } else if (siblings.size == 1) {
            return siblings.get(0);
        } else {
            // Name is also used as key. Thus, all siblings have the same name
            String name = siblings.get(0).name;

            Set<String> mergedNicknames = new HashSet<String>();
            for (Buddy buddy : siblings) {
                mergedNicknames.addAll(buddy.nicknames);
            }

            return new Buddy(name, mergedNicknames);
        }
    }
}
```

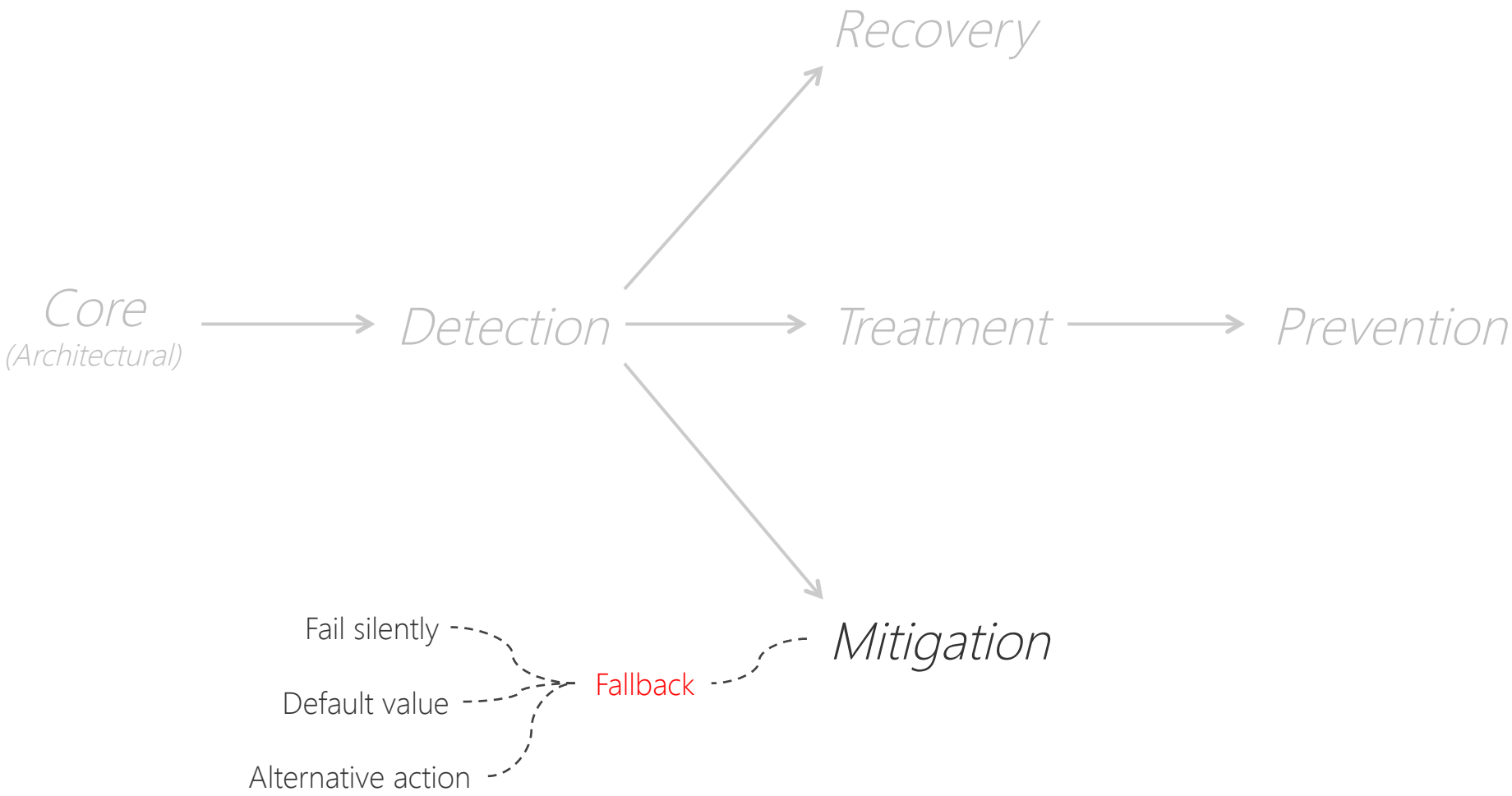

Checkpoint
Safe point
Limit retries
Rollback
Roll-forward
Retry
Failover
Reconnect
Recovery
Startup consistency
Restart
Reset
Read repair
Error handler
Data Reset
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation

Error Handler

- Separate business logic and error handling
- Business logic just focuses on getting the task done
- Error handler focuses on recovering from errors
- Easier to maintain – can be extended to structural escalation


Checkpoint
Safe point
Limit retries
Rollback
Roll-forward
Retry
Failover
Reconnect
Recovery
Startup consistency
Restart
Reset
Read repair
Error handler
Data Reset
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation


Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation


Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Fail silently
Default value
Alternative action
Fallback
Mitigation

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

Fail silently example (Hystrix, Java) 1/2

```
public class FailSilentlyCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = "default";
    private final boolean preCondition;

    public FailSilentlyCommand(boolean preCondition) {
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));
        this.preCondition = preCondition;
    }

    @Override
    protected String run() throws Exception {
        if (!preCondition)
            throw new RuntimeException((”Action failed"));
        return ”I am a result";
    }

    @Override
    protected String getFallback() {
        return null;  // Turn into silent failure
    }
}
```

Fail silently example (Hystrix, Java) 2/2

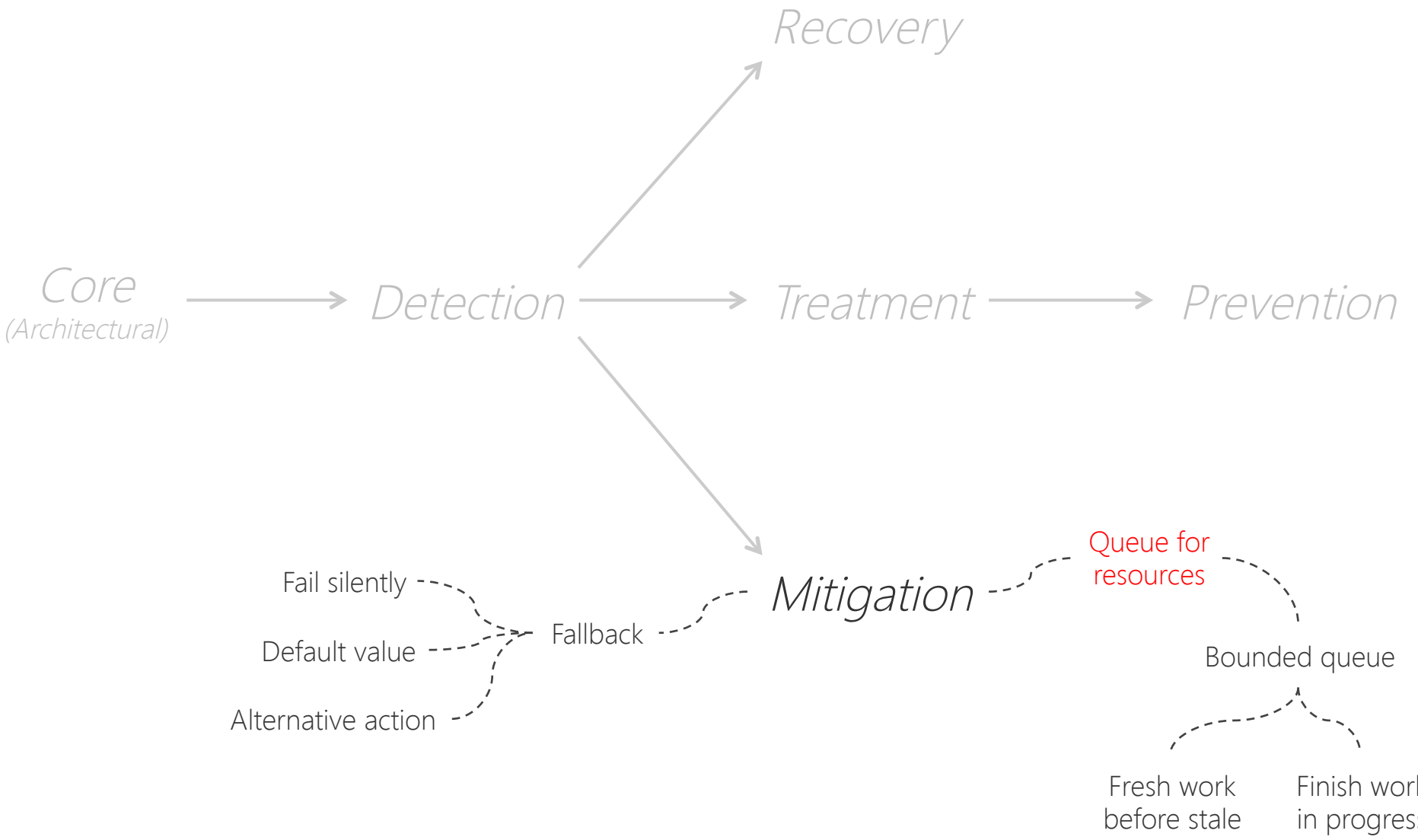
```
@Test
public void shouldSucceed() {
    FailSilentlyCommand command = new FailSilentlyCommand(true);
    String s = command.execute();

    assertEquals(”I am a result", s);
}


@Test
public void shouldFailSilently() {
    FailSilentlyCommand command = new FailSilentlyCommand(false);
    String s = ”Dummy";
    try {
        s = command.execute();
    } catch (Exception e) {
        fail("Did not fail silently");
    }
    assertNull(s);
}
```

Default value example (Hystrix, Java) 1/2

```
public class DefaultValueCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = "default”;
    private final boolean preCondition;

    public DefaultValueCommand(boolean preCondition) {
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));
        this.preCondition = preCondition;
    }

    @Override
    protected String run() throws Exception {
        if (!preCondition)
            throw new RuntimeException((”Action failed"));
        return ”I am a smart result";
    }

    @Override
    protected String getFallback() {
        return ”I am a default value";  // Return default value if action fails
    }
}
```

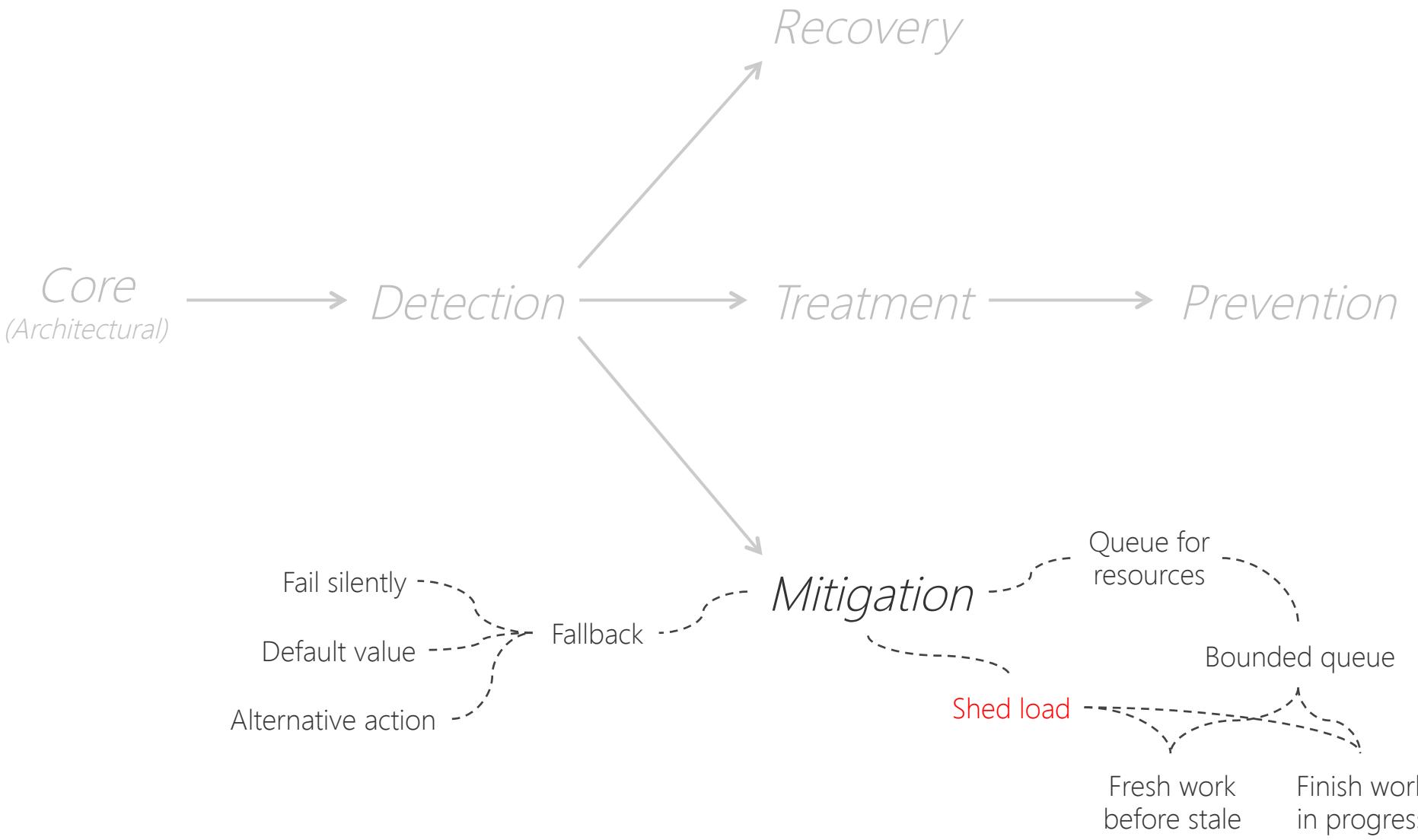
Default value example (Hystrix, Java) 2/2

```
@Test
public void shouldSucceed() {
    DefaultValueCommand command = new DefaultValueCommand(true);
    String s = command.execute();

    assertEquals(”I am a smart result", s);
}


@Test
public void shouldProvideDefaultValue () {
    DefaultValueCommand command = new DefaultValueCommand(false);
    String s = null;
    try {
        s = command.execute();
    } catch (Exception e) {
        fail("Did not return default value");
    }
    assertEquals(”I am a default value", s);
}
```


Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation
Queue for resources
Bounded queue
Fresh work before stale
Finish work in progress
Fallback
Fail silently
Default value
Alternative action

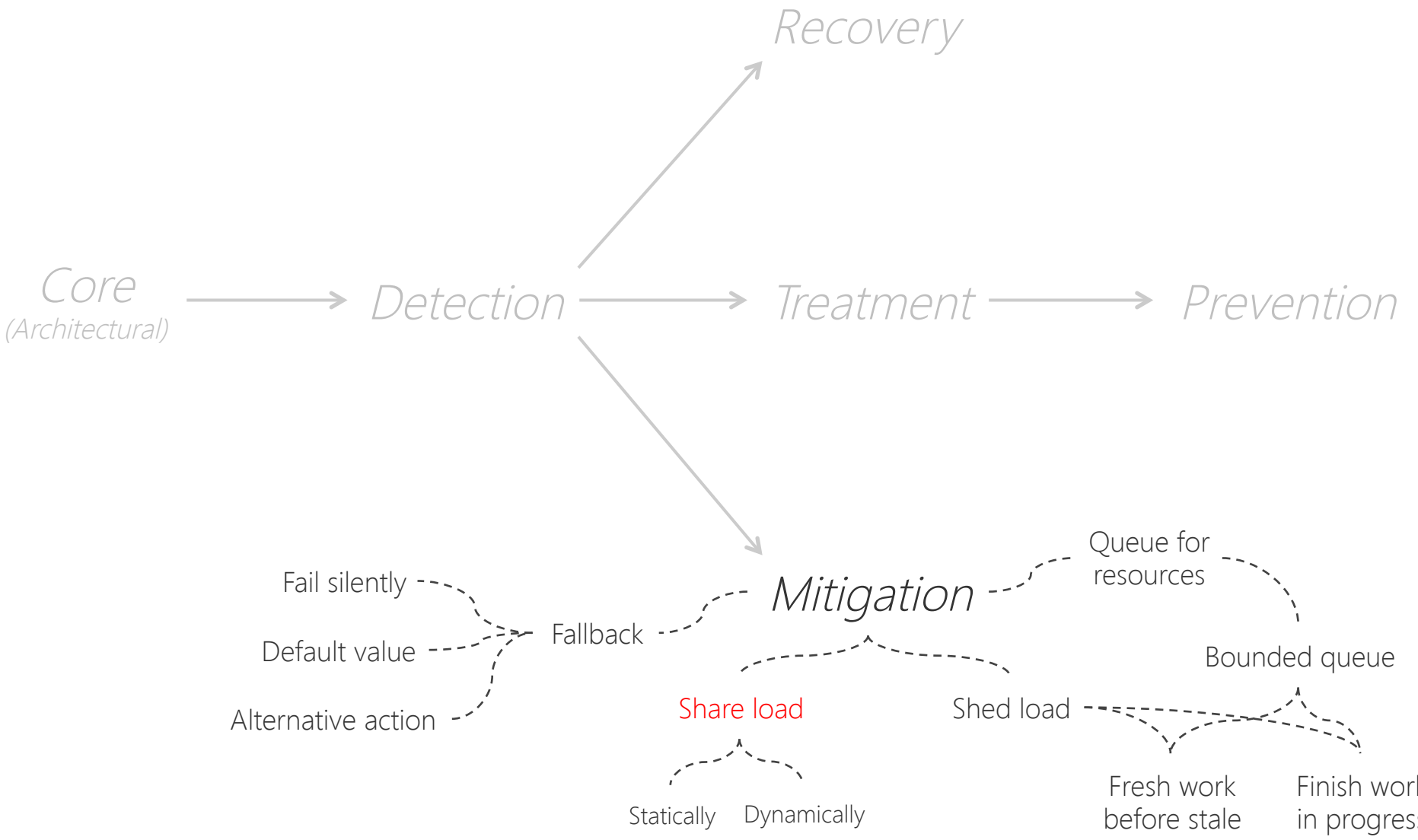
Queues for resources

- Protect resource from temporary overload situations
- Limit queue size to limit latency at longer-lasting overload
- Finish work in progress – Create pushback on the callers
- Fresh work before stale – Discard old entries


Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Fail silently
Default value
Alternative action
Fallback
Mitigation
Queue for resources
Bounded queue
Shed load
Fresh work before stale
Finish work in progress

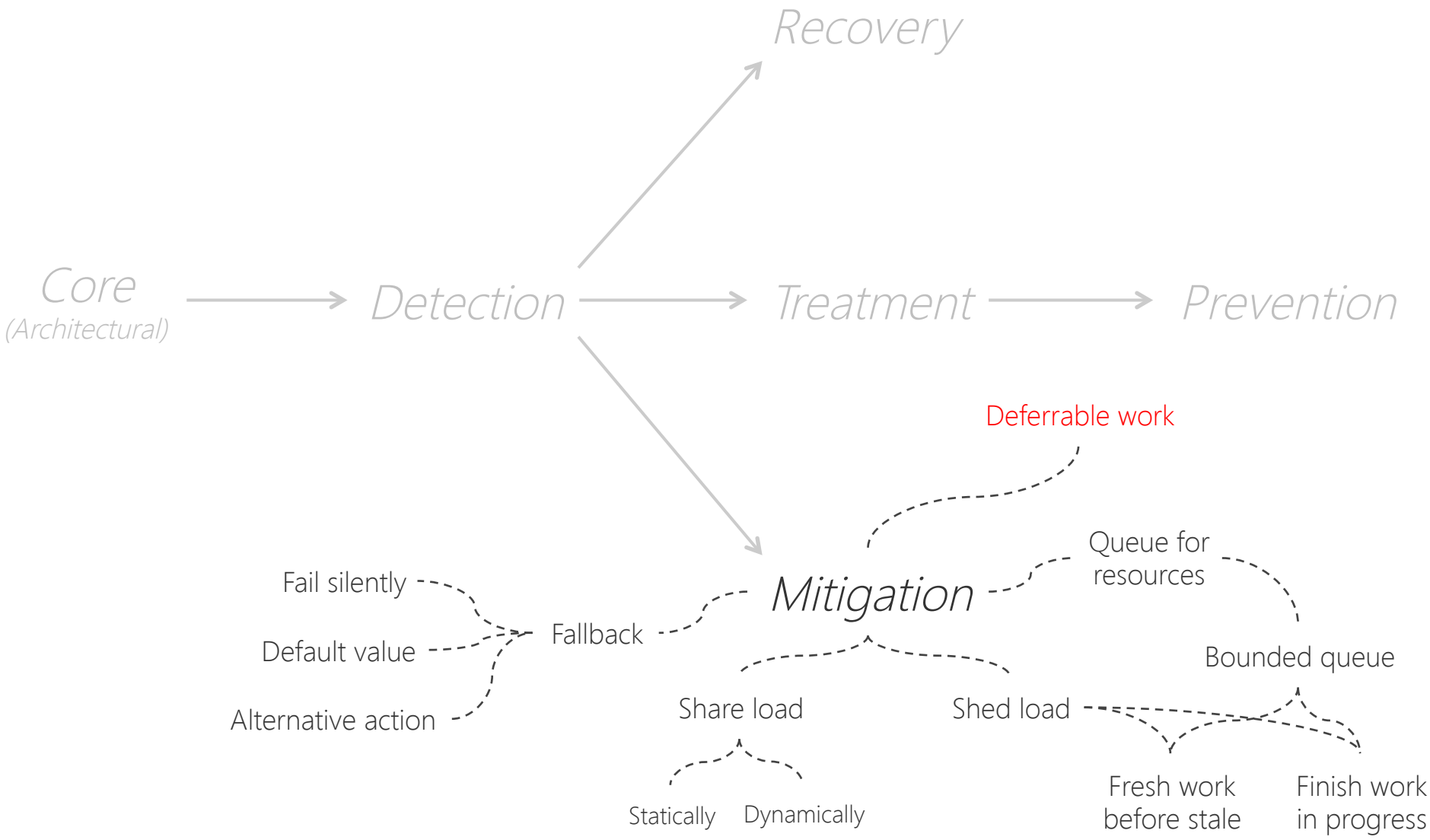
Shed Load

- Use if overload will lead to unacceptable throughput of resource
- Shed requests in order to keep throughput of resource acceptable
- Shed load at periphery – Minimize impact on resource itself
- Usually combined with *monitor* to watch load of resource


Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Fail silently
Default value
Alternative action
Fallback
Mitigation
Queue for resources
Bounded queue
Share load
Shed load
Statically
Dynamically
Fresh work before stale
Finish work in progress

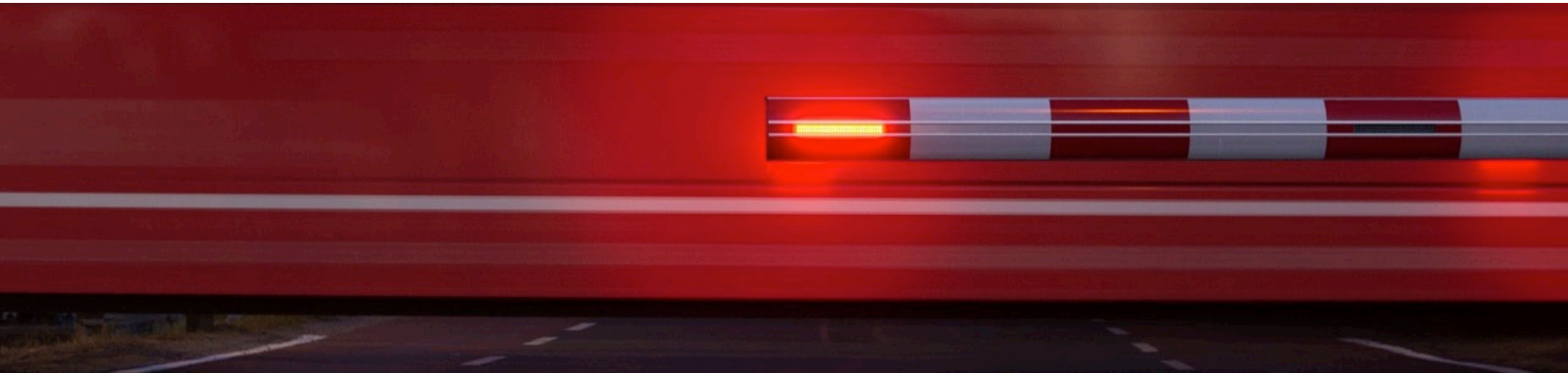
Share Load

- Use if overload will lead to unacceptable throughput of resource
- Share load between (added) resources to keep throughput good
- Minimize amount of synchronization needed between resources
- Usually combined with *monitor* to watch load of resource(s)


Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Deferrable work
Fail silently
Default value
Alternative action
Fallback
Mitigation
Queue for resources
Bounded queue
Share load
Shed load
Statically
Dynamically
Fresh work before stale
Finish work in progress

Deferrable work

- Maximize resources for online request processing under high load
- Pause or slow down routine and batch jobs
- Provide a means to pause routine and batch jobs from outside
- Alternatively use a scheduler with dynamic resource allocation

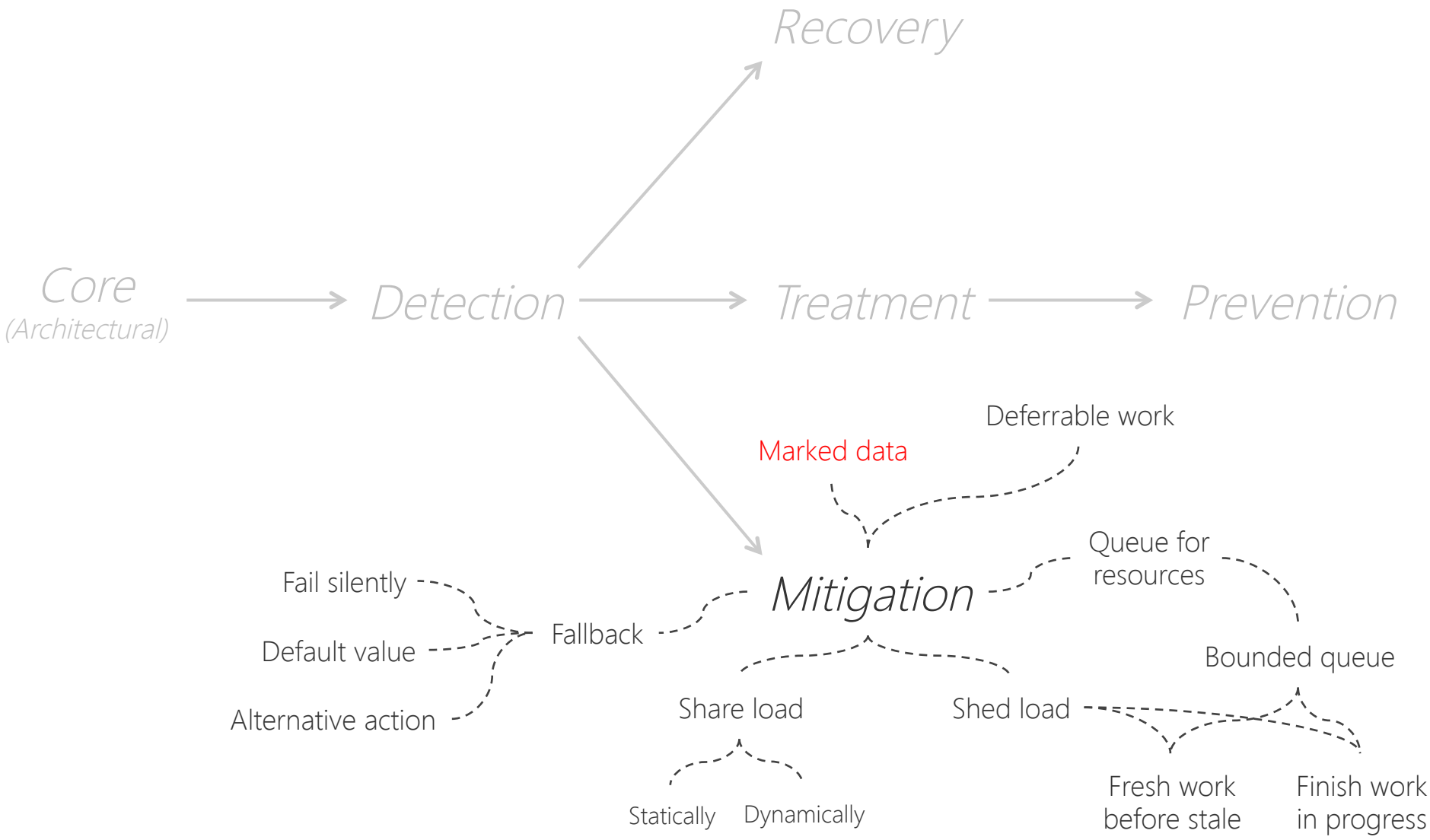


Deferrable work example 1/2

```
// Do or wait variant
<init batch>
while(<more to process>) {
    int load = getLoad();
    if (load > THRESHOLD) {
        waitFixedDuration();
    } else {
        <process next batch of work>
    }
}

void waitFixedDuration() {
    Thread.sleep(DELAY);  // try-catch left out for better readability
}
```

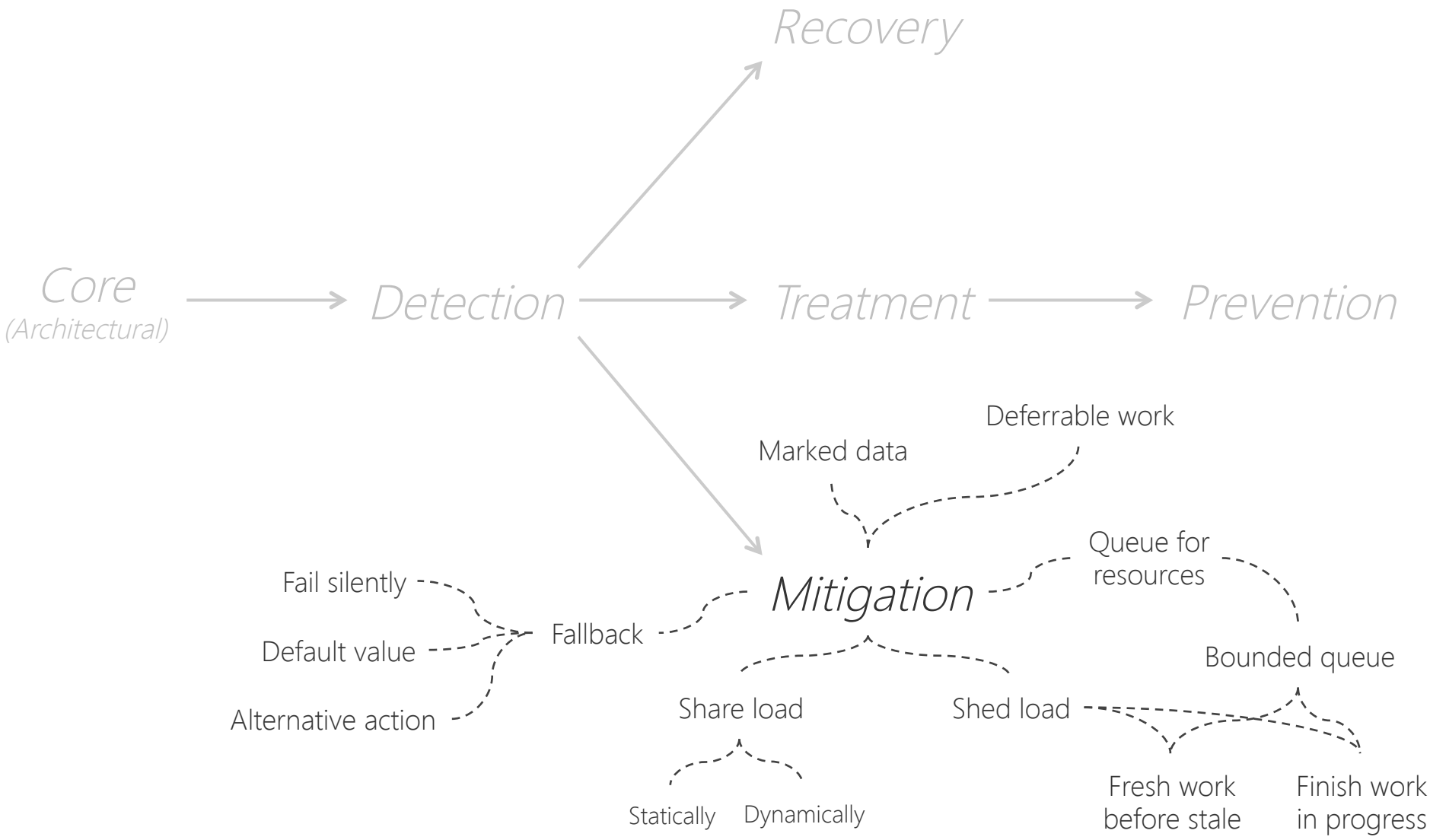
Deferrable work example 2/2

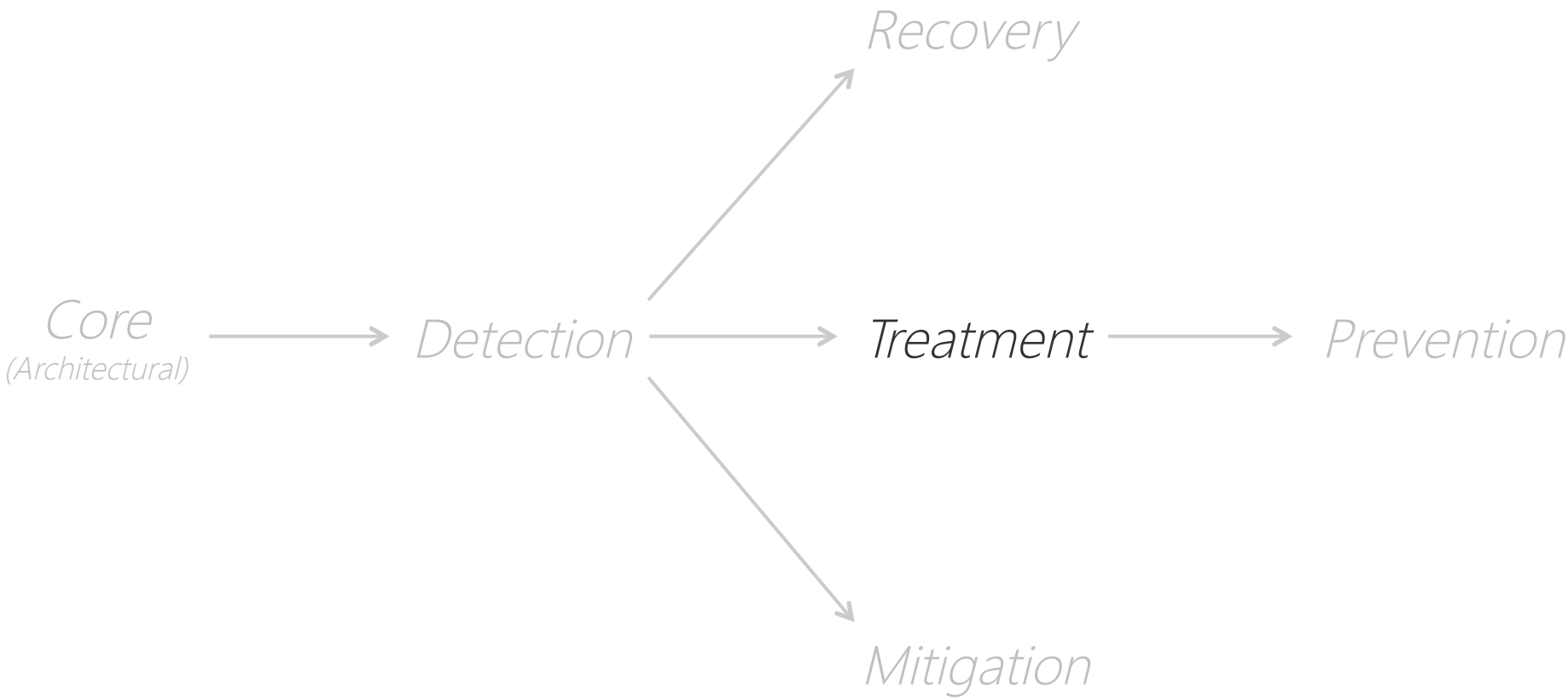
```
// Adaptive load variant
<init batch>
while(<more to process>) {
    waitLoadBased();
    <process next batch of work>
}

void waitLoadBased() {
    int load = getLoad();
    long delay = calcDelay(load);
    Thread.sleep(delay);  // try-catch left out for better readability
}

long calcDelay(int load) {  // Simple example implementation
    if (load < THRESHOLD) {
        return 0L;
    }
    return (load - THRESHOLD) * DELAY_FACTOR;
}
```

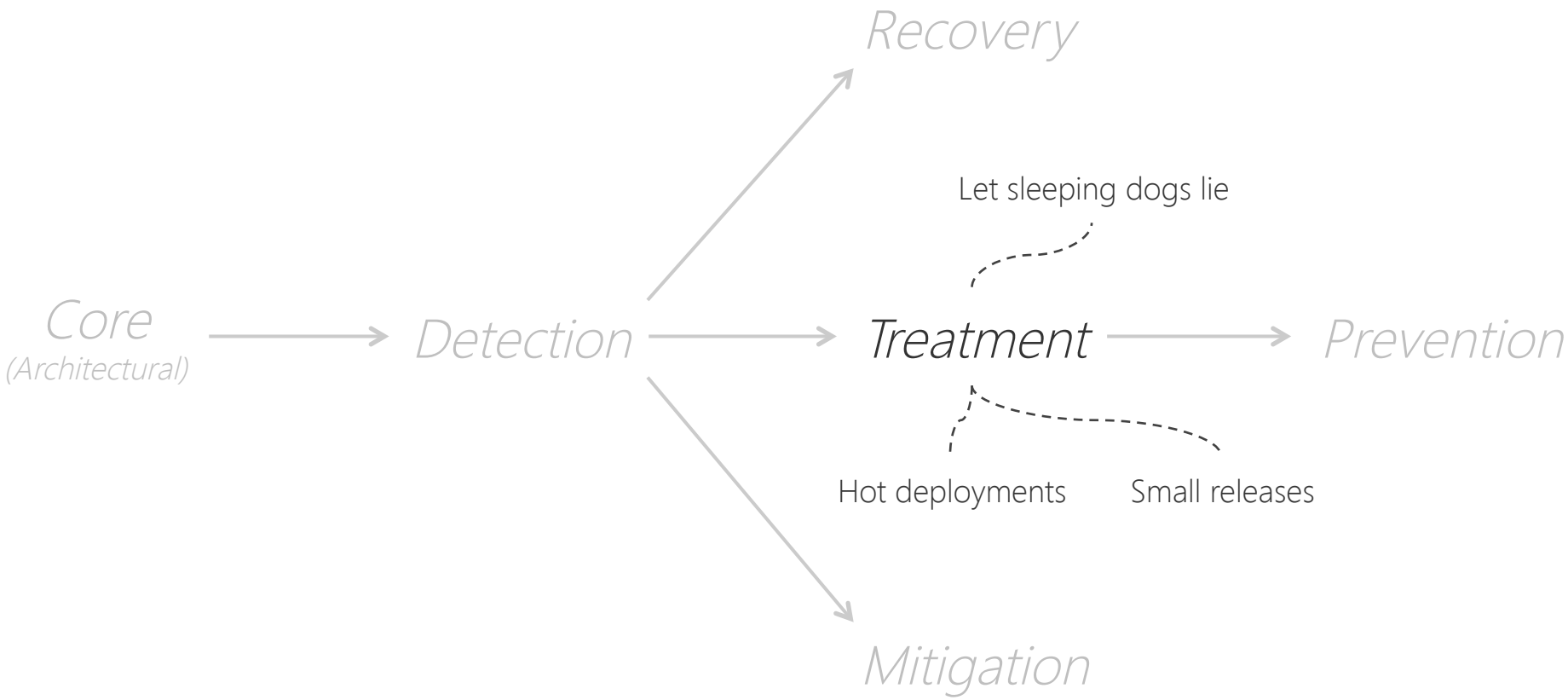

Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Deferrable work
Marked data
Queue for
resources
Fail silently
Default value
Alternative action
Fallback
Mitigation
Bounded queue
Share load
Shed load
Statically
Dynamically
Fresh work
before stale
Finish work
in progress

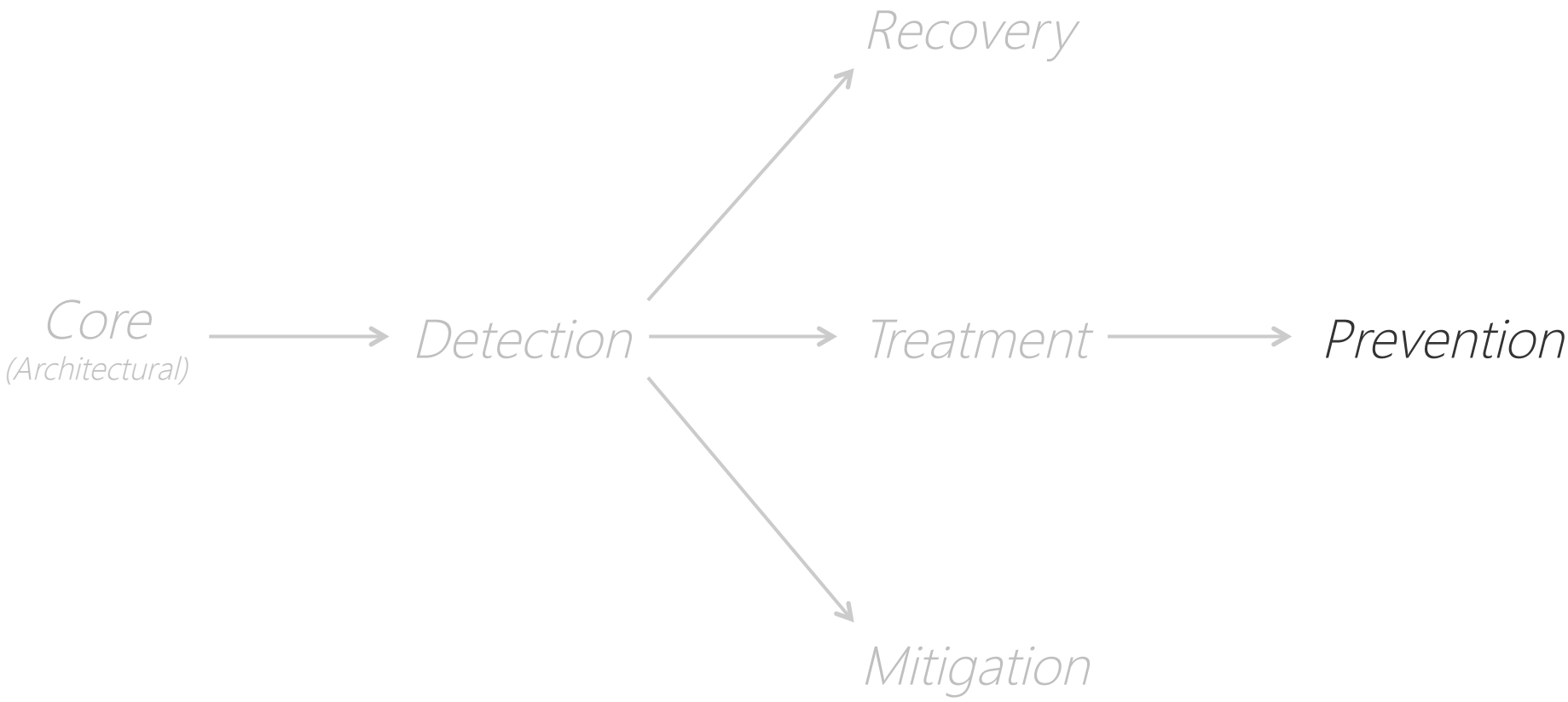
Marked data

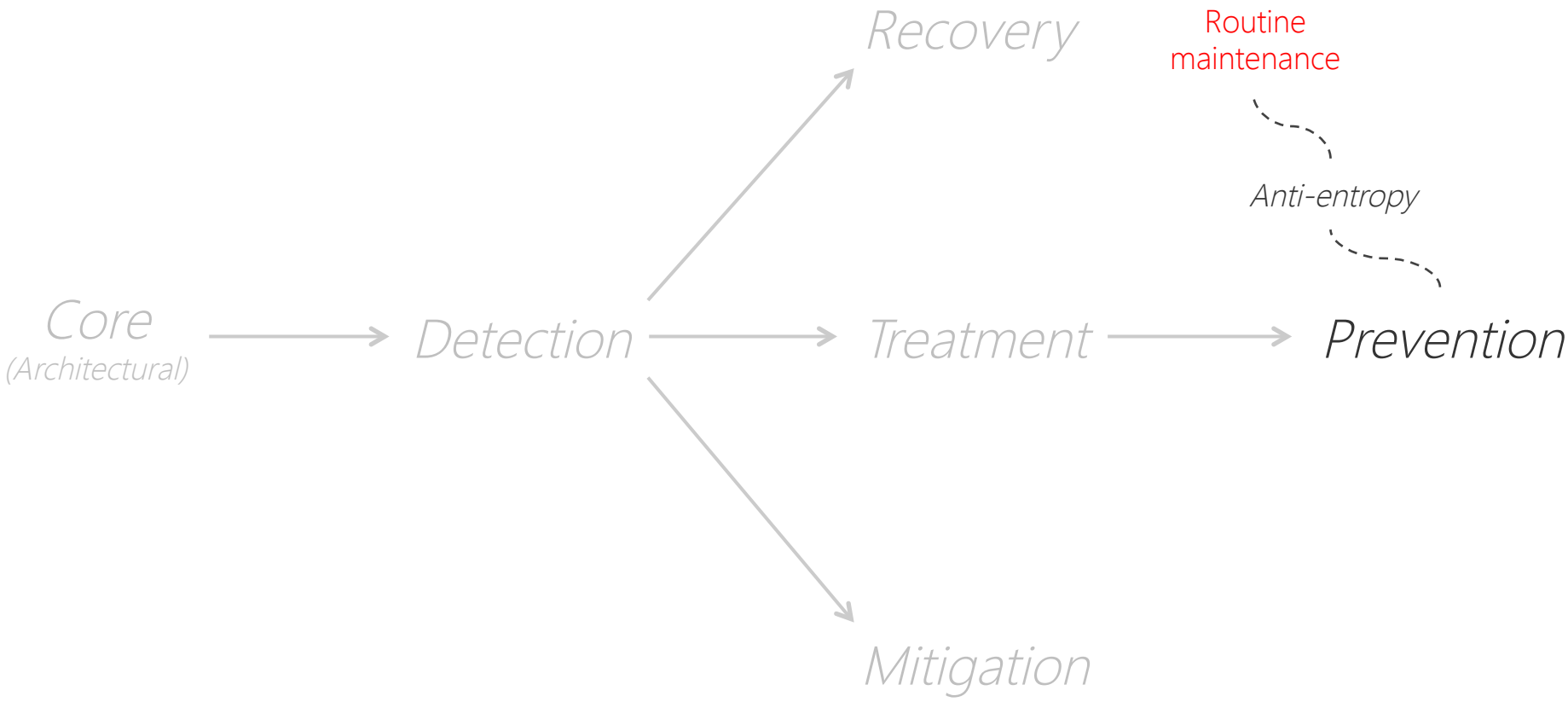
- Avoid repeated and/or spreading errors due to erroneous data
- Use if time or information to correct data immediately is missing
- Mark data as being erroneous – check flag before processing data
- Use routine maintenance job to correct data


Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Deferrable work
Marked data
Queue for
resources
Fail silently
Mitigation
Default value
Fallback
Bounded queue
Alternative action
Share load
Shed load
Statically
Dynamically
Fresh work
before stale
Finish work
in progress


Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation

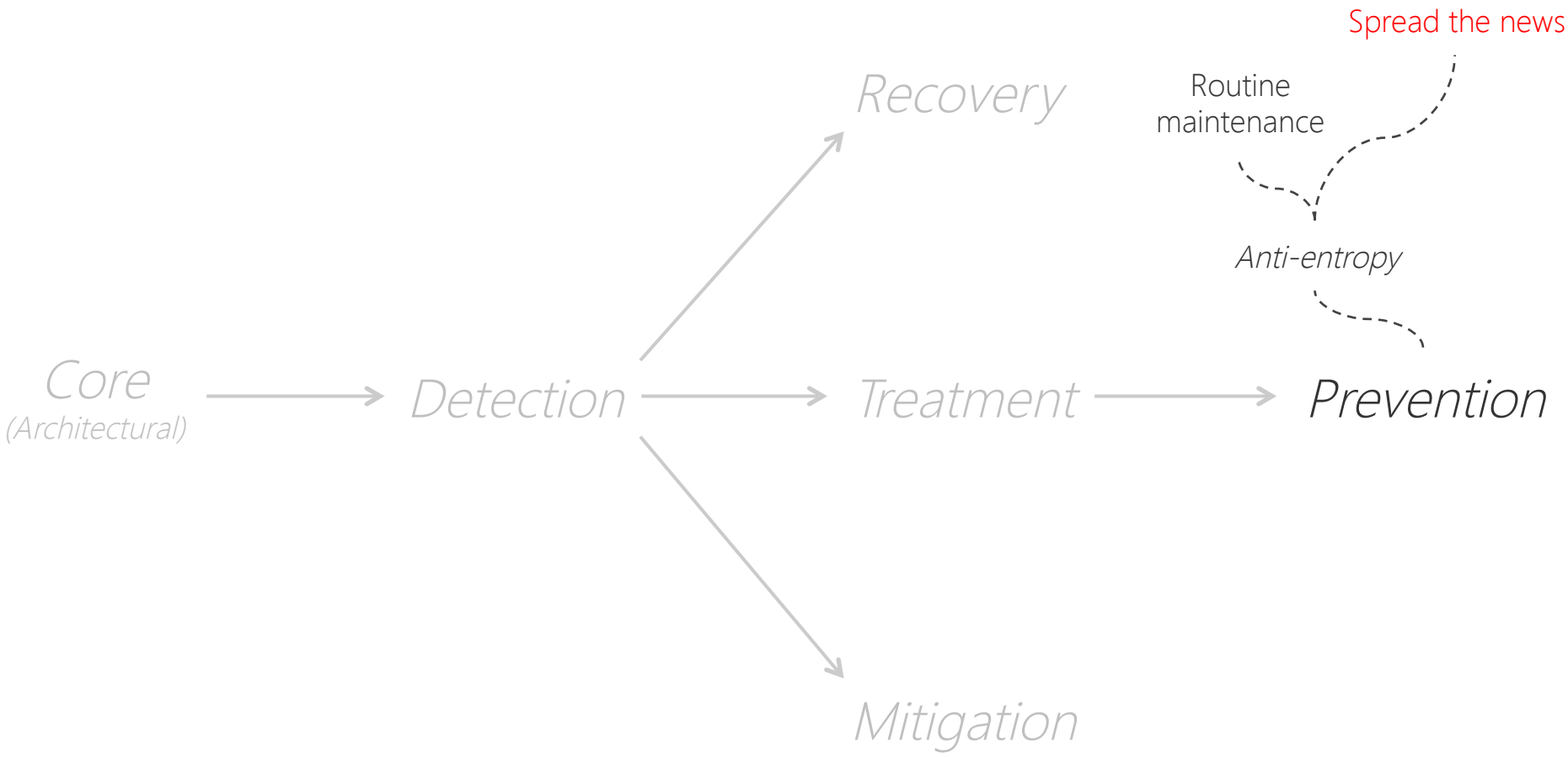

Recovery
Let sleeping dogs lie
Core
(Architectural)
Detection
Treatment
Prevention
Hot deployments
Small releases
Mitigation


Recovery
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation


Recovery
Routine
maintenance
Anti-entropy
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation

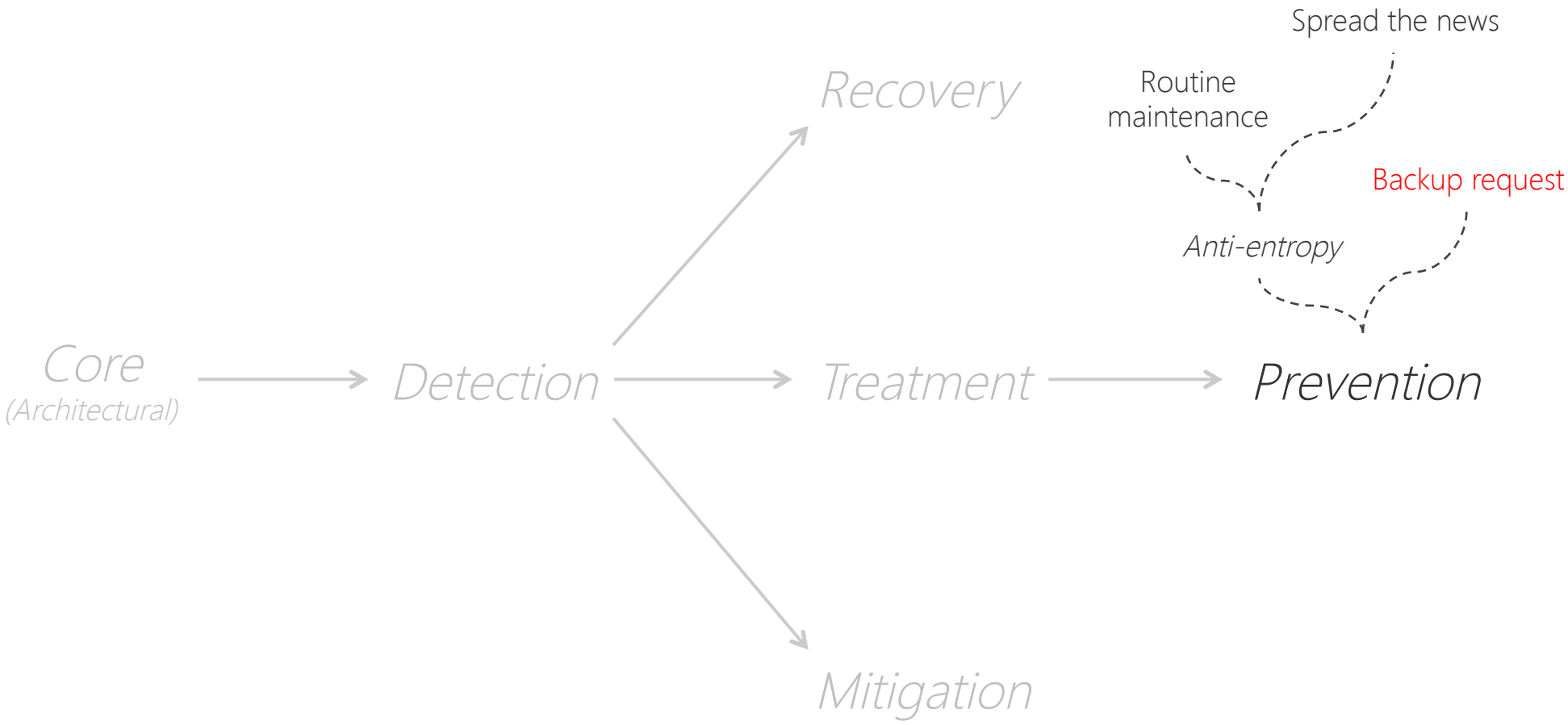
Routine maintenance

- Reduce system entropy – keep preventable errors from occurring
- Especially important if errors were only mitigated, not corrected
- Check system periodically and fix detected faults and errors
- Balance benefits, costs and additional system load


Spread the news
Recovery
Routine
maintenance
Anti-entropy
Core
(Architectural)
Detection
Treatment
Prevention
Mitigation

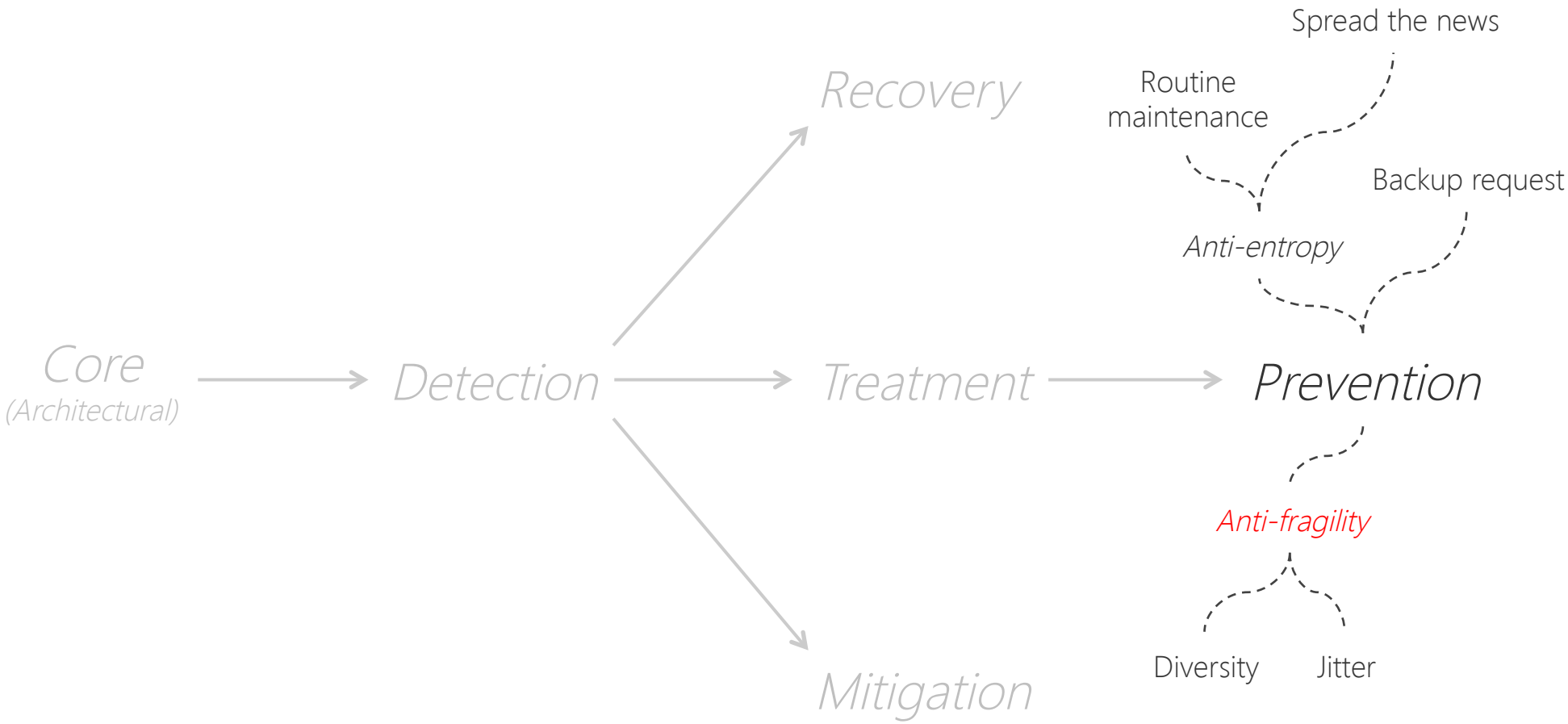
Spread the news

- Pro-actively spread information about changes in system state
- Use a gossip or epidemic protocol for robustness and efficiency
- Can also be used for data reconciliation
- Balance benefits, costs and additional network load


Core
(Architectural)
Detection
Recovery
Treatment
Mitigation
Prevention
Spread the news
Routine
maintenance
Backup request
Anti-entropy

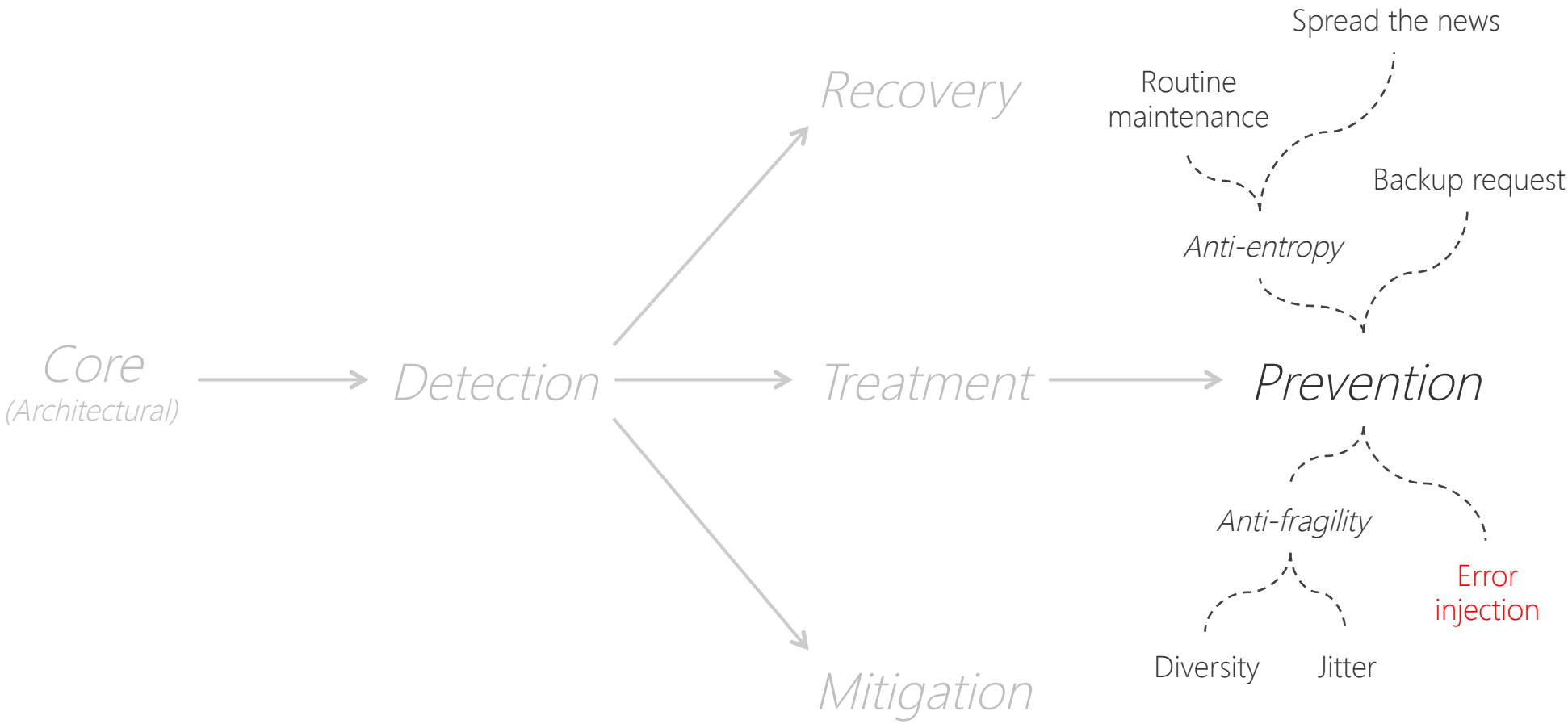
Backup request

- Send request to multiple workers (optionally a bit offset)
- Use quickest reply and discard all other responses
- Prevents latent responses (or at least reduces probability)
- Requires redundancy – trades resources for availability


Core
(Architectural)
Detection
Recovery
Treatment
Mitigation
Prevention
Spread the news
Routine maintenance
Backup request
Anti-entropy
Anti-fragility
Diversity
Jitter

Anti-fragility

- Avoid fragility caused by homogenization and standardization
- Protect against disastrous failures by using diverse solutions
- Protect against cumulating effects by introducing jitter
- Balance risks, benefits and added costs and efforts carefully


Core
(Architectural)
Detection
Recovery
Treatment
Mitigation
Prevention
Spread the news
Routine maintenance
Backup request
Anti-entropy
Anti-fragility
Error injection
Diversity
Jitter

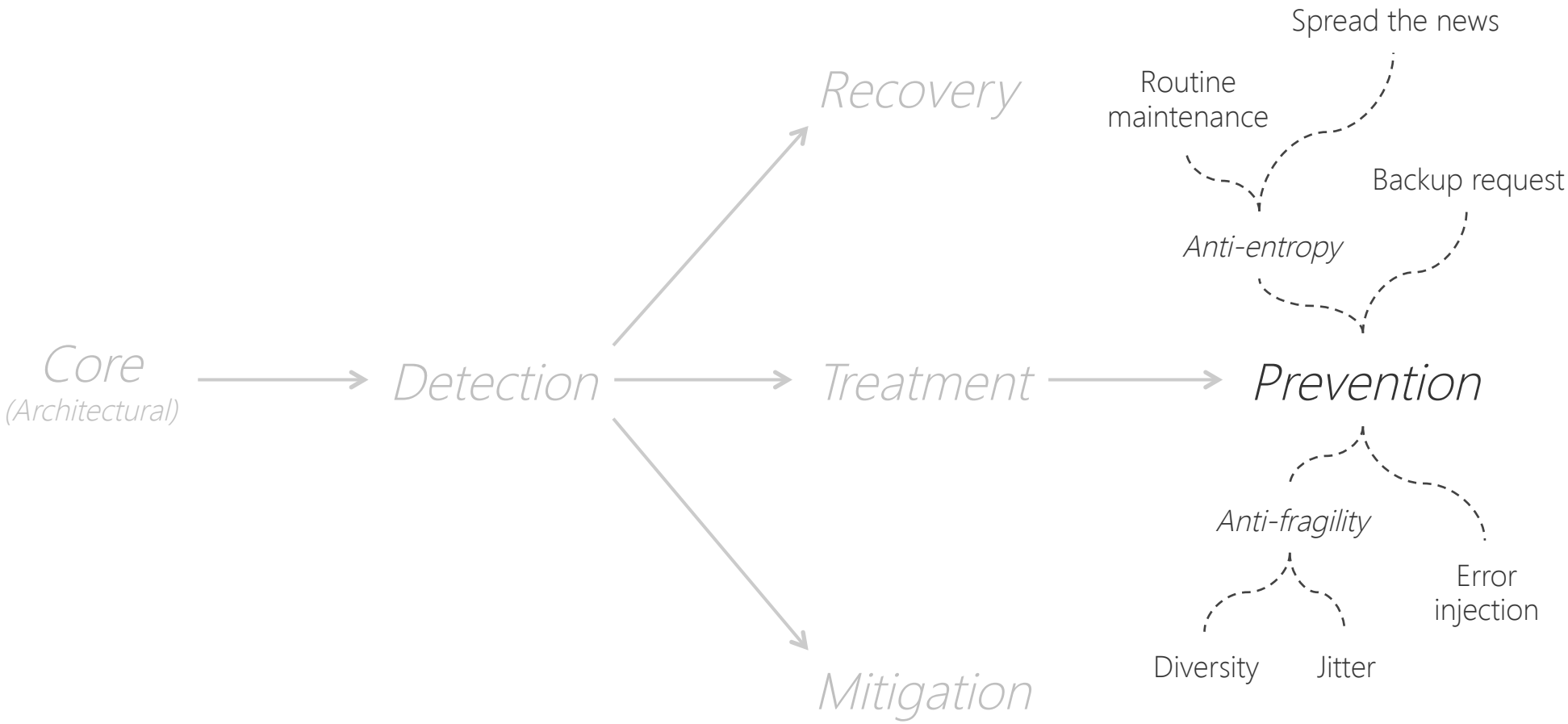
Error injection

- Make resilient software design sustainable
- Inject errors at runtime and observe how the system reacts
- Can also be used to detect yet unknown failure modes
- Make sure to inject errors of all types

NETFLIX

- Chaos Monkey
- Chaos Gorilla
- Chaos Kong
- Latency Monkey
- Compliance Monkey
- Security Monkey
- Janitor Monkey
- Doctor Monkey

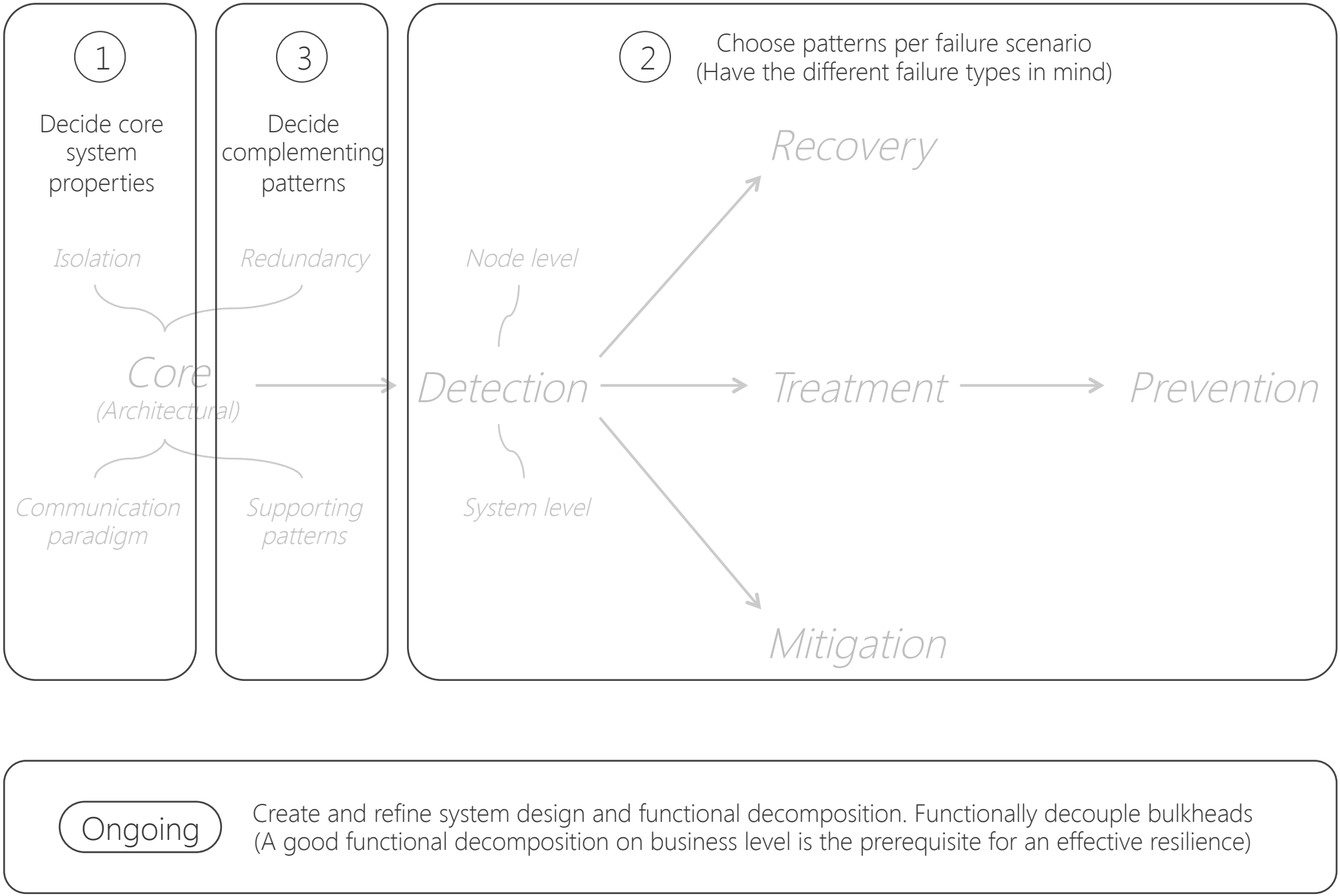
https://github.com/Netflix/SimianArmy


Core
(Architectural)
Detection
Recovery
Treatment
Mitigation
Prevention
Spread the news
Routine maintenance
Backup request
Anti-entropy
Anti-fragility
Error injection
Diversity
Jitter

Towards a pattern language …

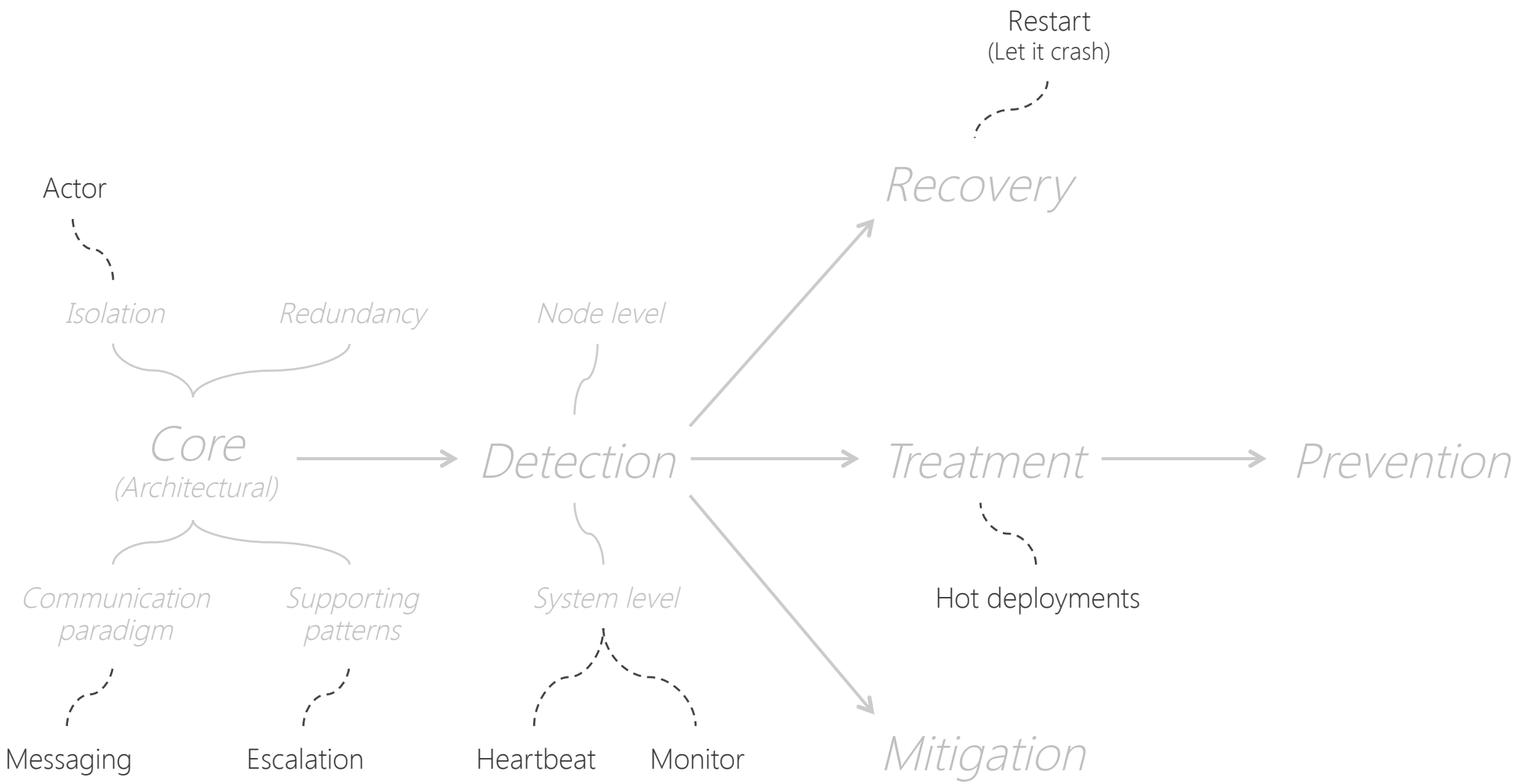
Decisions to make

- 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

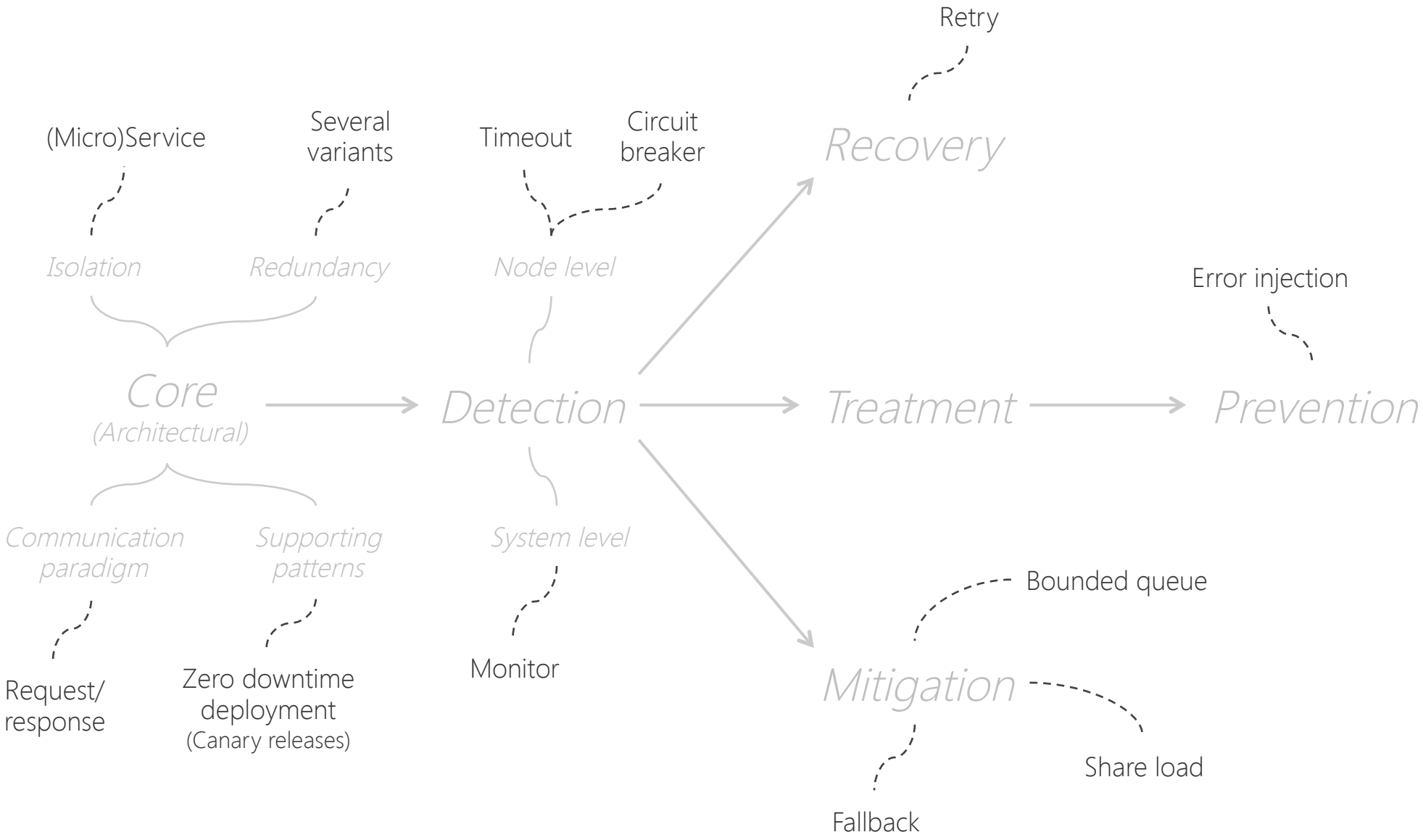


Ongoing

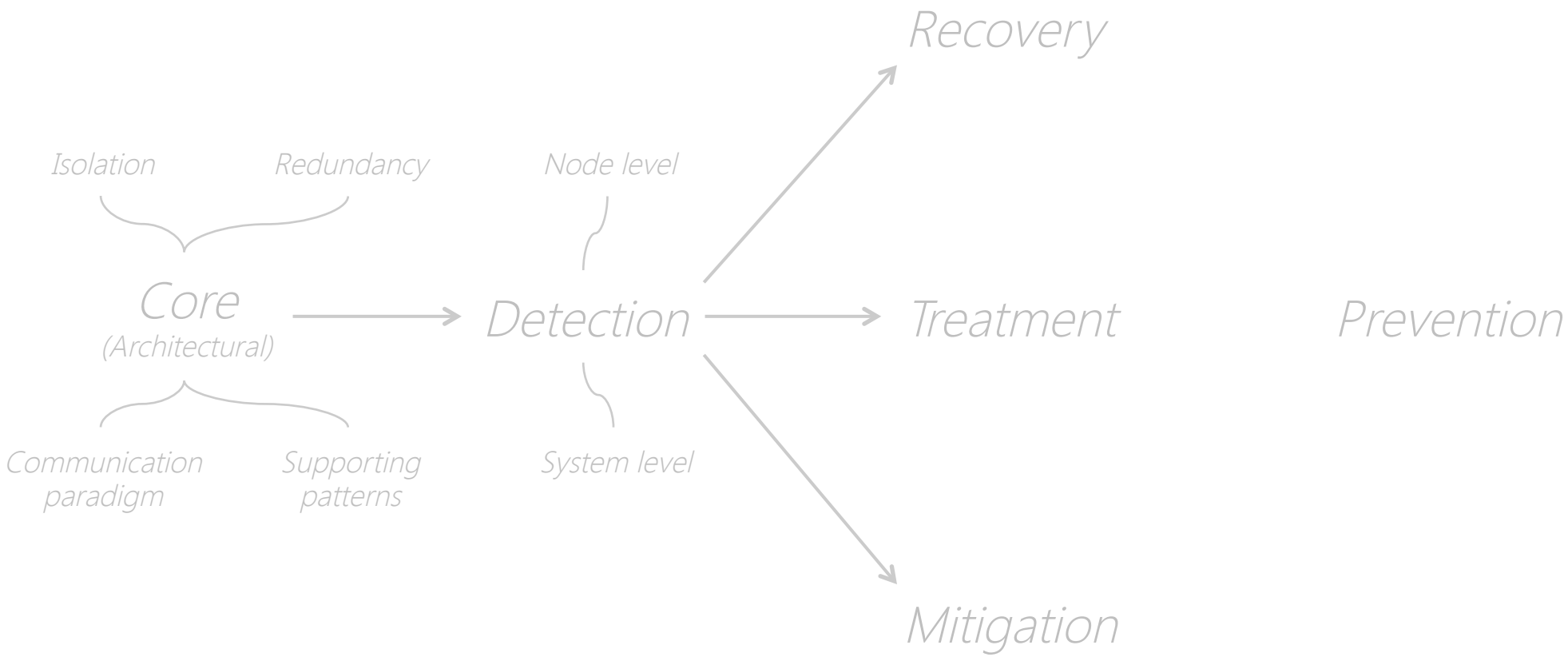
Create and refine system design and functional decomposition. Functionally decouple bulkheads
(A good functional decomposition on business level is the prerequisite for an effective resilience)



Example: Erlang (Akka)



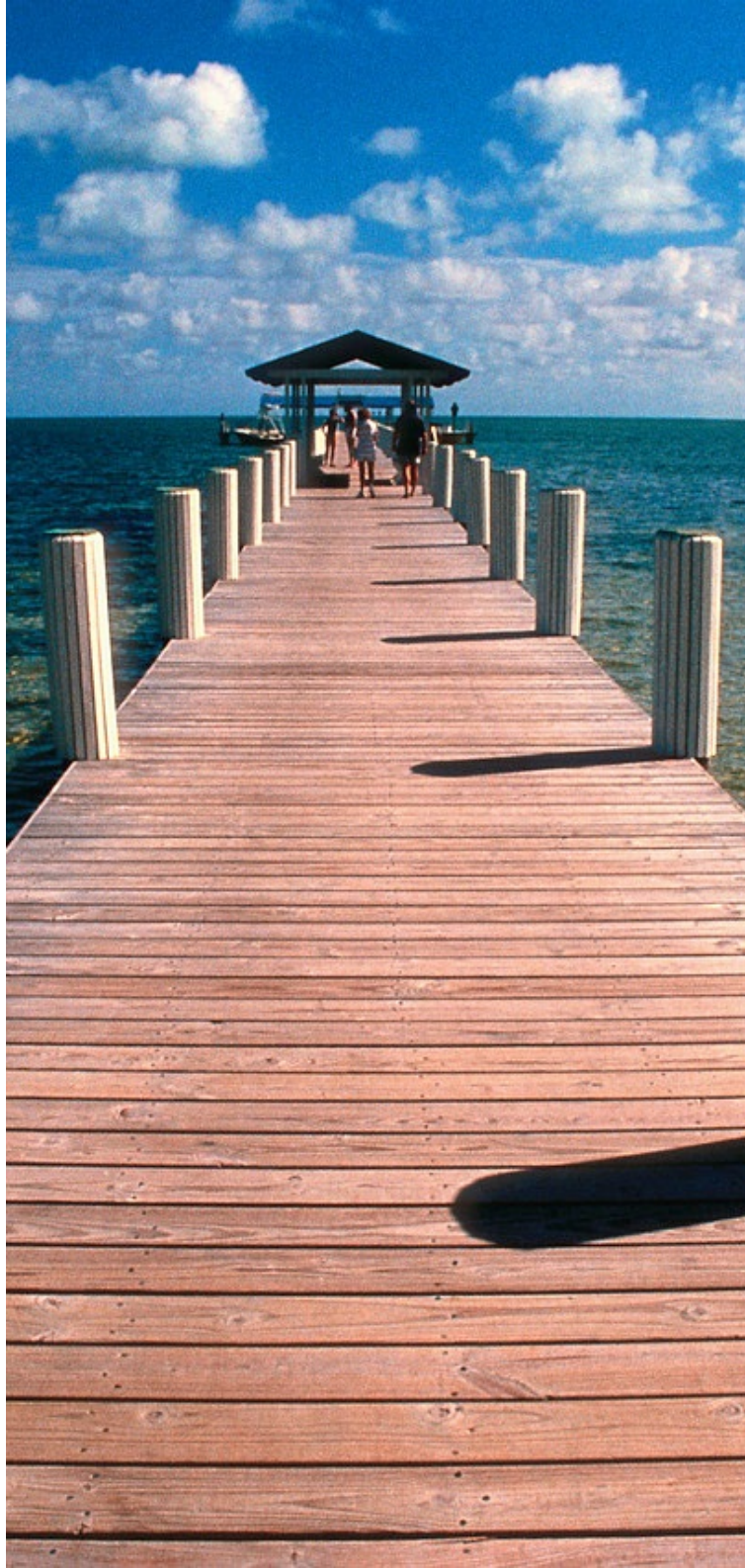
Example: Netflix



What is your pattern language?

Wrap-up

- Today's systems are distributed
- Failures are not avoidable
- Failures are not predictable
- Resilient software design needed
- Rich pattern language
- Start with core system properties
- Choose patterns based on failure scenarios
- Complement with careful functional design



Further reading

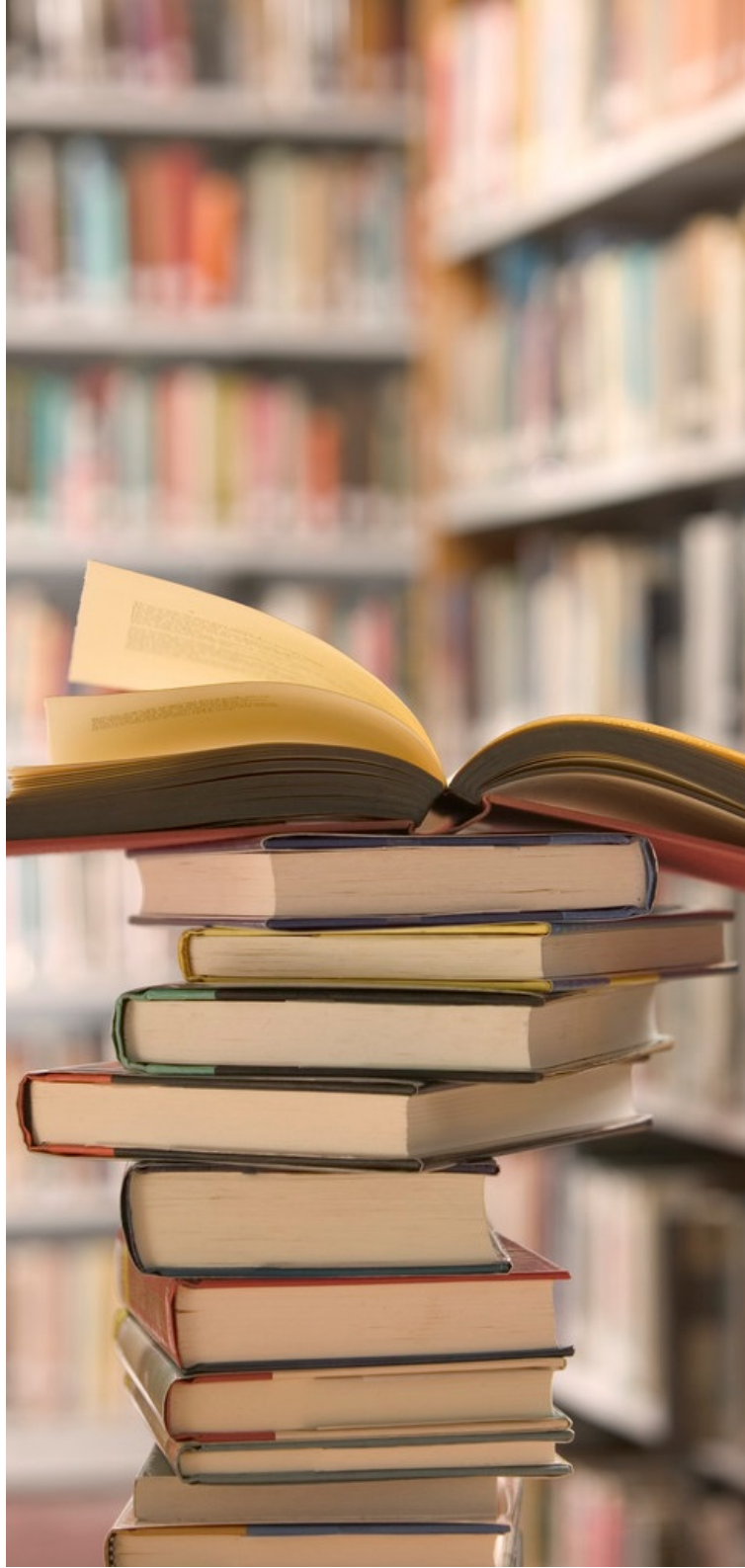
1. Michael T. Nygard, Release It!,
   Pragmatic Bookshelf, 2007

2. Robert S. Hanmer,
   Patterns for Fault Tolerant Software, Wiley, 2007

3. Andrew Tanenbaum, Marten van Steen,
   Distributed Systems – Principles and Paradigms,
   Prentice Hall, 2nd Edition, 2006

4. Hystrix Wiki,
   https://github.com/Netflix/Hystrix/wiki

5. Uwe Friedrichsen, Patterns of resilience,
   http://de.slideshare.net/ufried/patterns-of-resilience



Do not avoid failures. Embrace them!

@ufried

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