Resilient Design

(short version)

CS350

Doo-Hwan Bae

School of Computing, KAIST

Resilence Design Patterns

Resources

- https://docs.microsoft.com/en-us/azure/architecture/patterns/category/resiliency
- http://microservices.io/patterns/monolithic.html
- https://conferences.oreilly.com/software-architecture/sa-eu-2017/public/schedule/detail/61746
- https://www.thoughtworks.com/de/insights/blog/scaling-microservices-event-stream

Contents

- What & Why?
- Resilient Patterns

“We will prepare for the armies of illogical users who do crazy, unpredictable things.” (by Michael Nygard)

What?

- Resilience:
  - Ability of a system to handle unexpected situations
    - Best case: without the user noticing it
    - Worst case: with a graceful degradation of service
- Part of design activity

Why? (1/2)

- Distributed systems are everywhere
- Fallacies of distributed systems (wrong perception/assumption)
  - Network is reliable, secure, homogeneous
  - Zero latency
  - Infinite bandwidth
  - No change on topology
  - One administrator
  - …
- Failures in distributed systems are not the exception
  - Normal, and even worse is ‘not predictable’
  - What do we do with such systems?
    - Option 1: Develop a fail-free system
    - Option 2: Embrace failures and increase availability of the system

Why? (2/2)

- It is getting worse and worse with recent IT evolution
  - Too complex to manage with traditional approaches
- Some of such system examples are
  - Cloud-based system
  - Microservices
  - Zero downtime (100% availability)
  - Mobile
  - IoT, CPS
  - Social Web
  - System of Systems

Resilience Approach

- Availability = MTTF / (MTTF + MTTR)

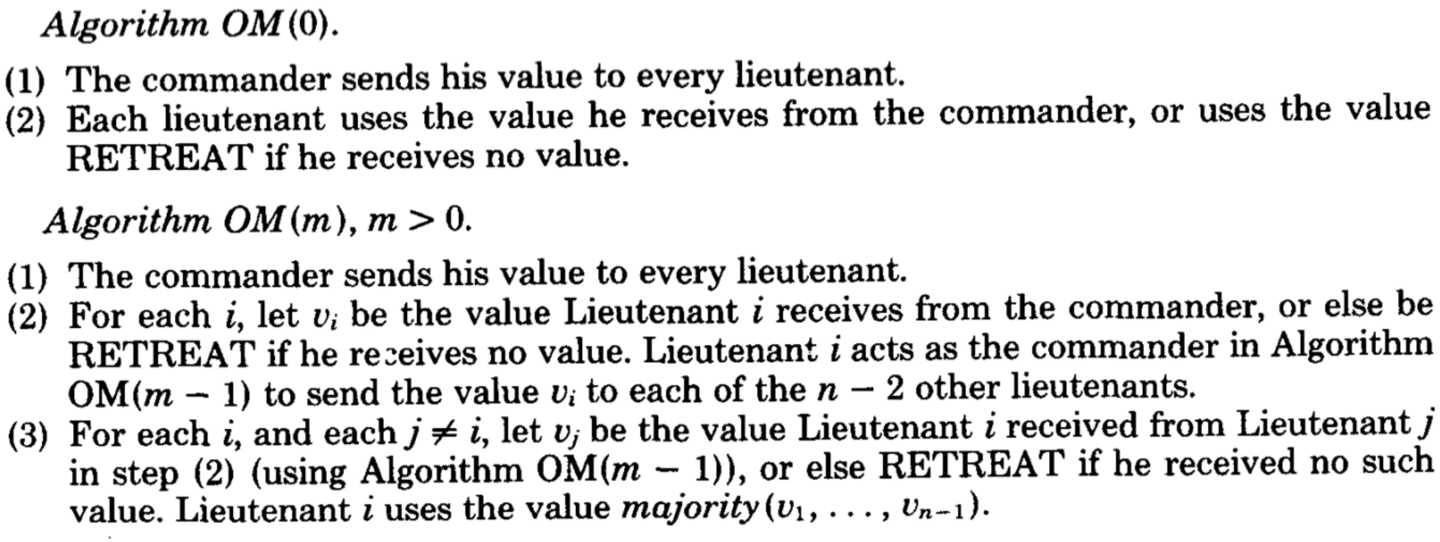
  - MTTF: Mean Time To Failure

  - MTTR: Mean Time to Repair

- How can we increase the availability of a (distributed) system?
  - Increase MTTF
  - Reduce MTTR
- Failure types: Crash failure, Omission failure, Timing failure, Response failure, Byzantine failure

‘Byzantine Generals Problem’

- Theorem: For any *m*, Algorithm *OM(m)* reaches consensus if there are more than *3m* generals and at most *m* traitors.
- This implies that the algorithm can reach consensus as long as 2/3 of the actors are honest. If the traitors are more than 1/3, consensus is not reached, the armies do not coordinate their attack and the enemy wins.
- Where to use?
- How to validate it?



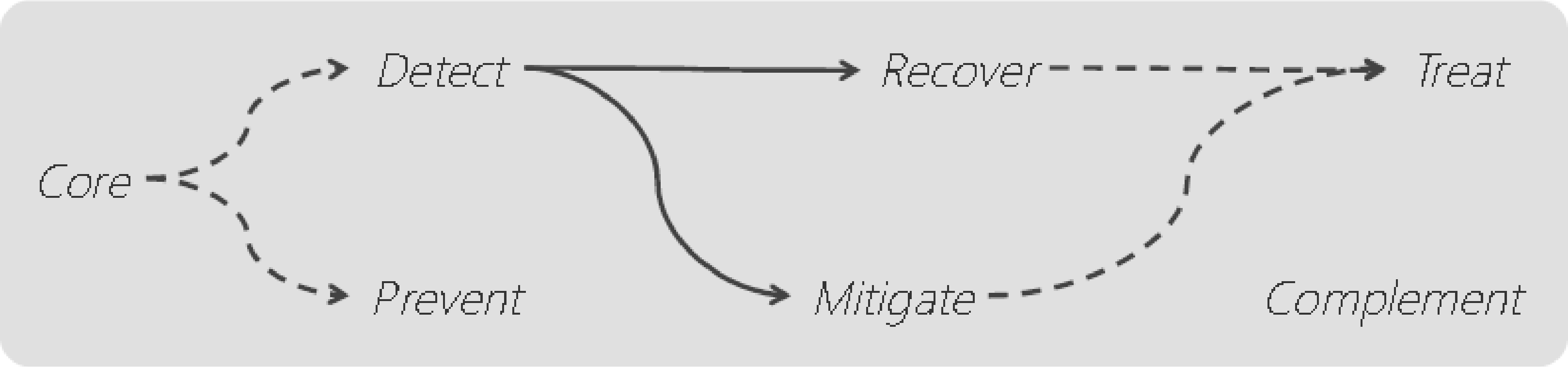
Algorithm OM(0).

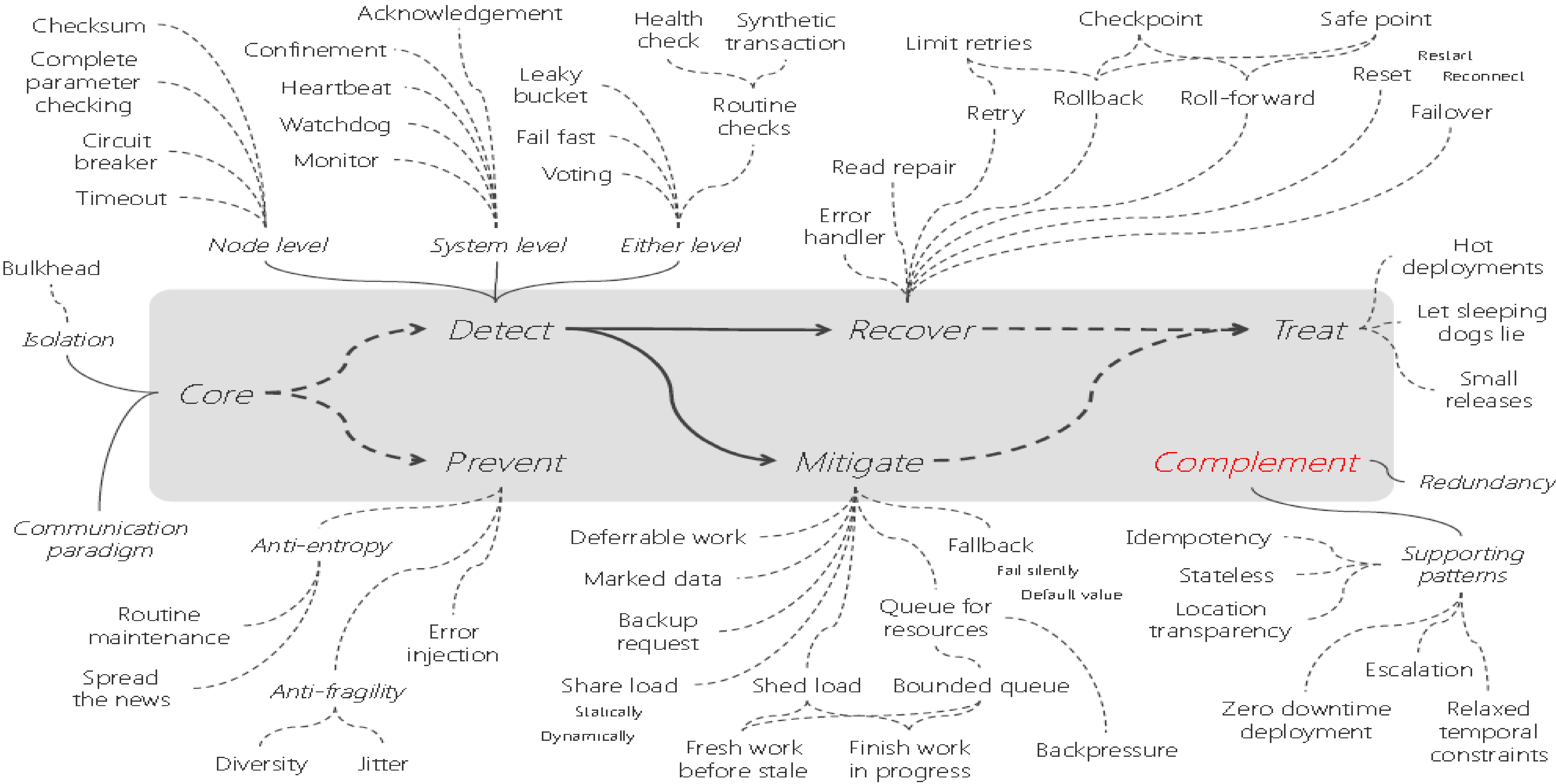
(1) The commander sends his value to every lieutenant.
(2) Each lieutenant uses the value he receives from the commander, or uses the value RETREAT if he receives no value.

Algorithm OM(m), $m > 0$.

(1) The commander sends his value to every lieutenant.
(2) For each i, let v_i be the value Lieutenant i receives from the commander, or else be RETREAT if he receives no value. Lieutenant i acts as the commander in Algorithm OM$(m - 1)$ to send the value v_i to each of the $n - 2$ other lieutenants.
(3) For each i, and each $j \neq i$, let v_j be the value Lieutenant i received from Lieutenant j in step (2) (using Algorithm OM$(m - 1)$), or else RETREAT if he received no such value. Lieutenant i uses the value *majority*$(v_1, \ldots, v_{n-1})$.

Whole Picture for Resilient Design





Core
Detect
Prevent
Recover
Mitigate
Treat
Complement

Isolation
Bulkhead
Communication paradigm

Node level
Checksum
Complete parameter checking
Circuit breaker
Timeout

System level
Acknowledgement
Confinement
Heartbeat
Watchdog
Monitor

Either level
Leaky bucket
Fail fast
Voting
Routine checks
Health check
Synthetic transaction

Error handler
Read repair
Retry
Limit retries
Rollback
Roll-forward
Checkpoint
Safe point
Reset
Restart
Reconnect
Failover

Hot deployments
Let sleeping dogs lie
Small releases

Anti-entropy
Routine maintenance
Spread the news
Error injection
Anti-fragility
Diversity
Jitter

Deferrable work
Marked data
Backup request
Share load
Statically
Dynamically
Shed load
Fresh work before stale
Finish work in progress
Bounded queue
Queue for resources
Fallback
Fail silently
Default value
Backpressure

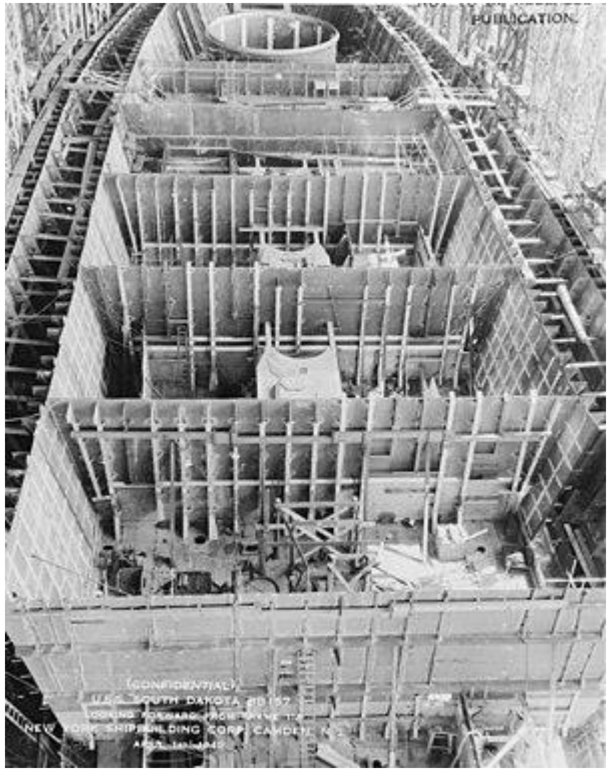
Redundancy
Supporting patterns
Idempotency
Stateless
Location transparency
Escalation
Zero downtime deployment
Relaxed temporal constraints

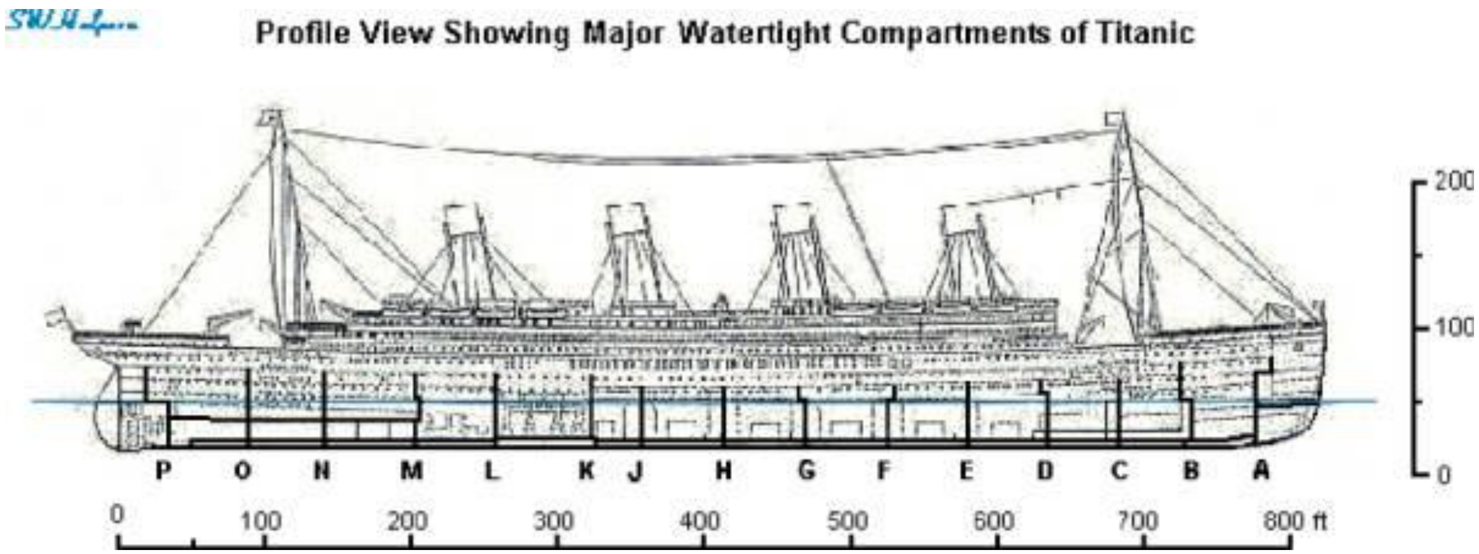
Isolation

- System must not fail as a whole
- Split system in parts and isolate parts against each other
- Avoid cascading failures
- Foundations of resilient software design
  - *High cohesion, low coupling*
  - *Separation of concerns*
- Isolation patterns
  - Bulkhead Design
  - Monolithic vs. Microarchitecture

Bulkhead Pattern

- Isolate elements of an application into pools so that if one fails, the others will continue to function.

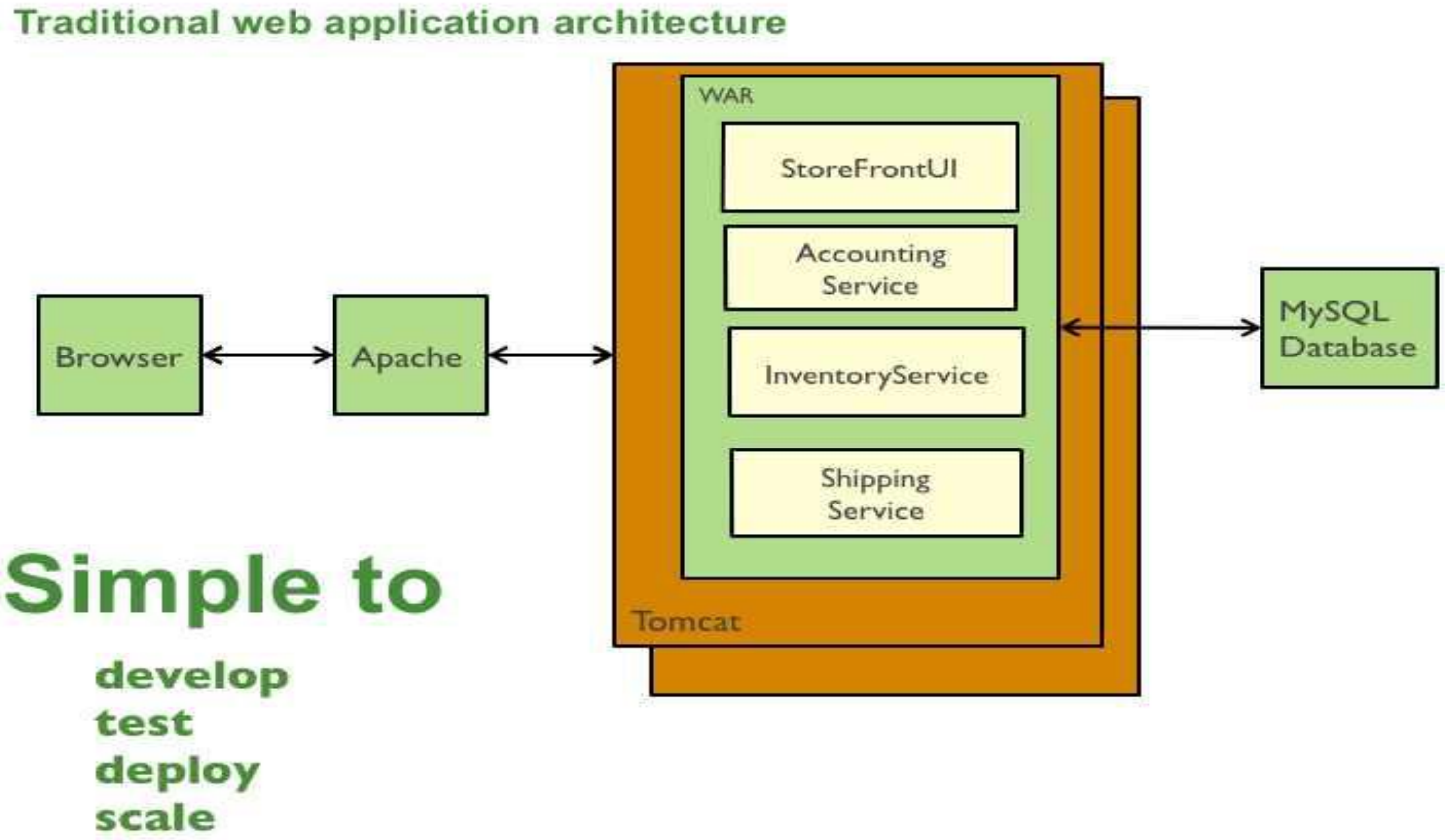



Profile View Showing Major Watertight Compartments of Titanic
P
O
N
M
L
K
J
H
G
F
E
D
C
B
A
0
100
200
300
400
500
600
700
800 ft
200
100
0

Bulkheads Pattern

- Core isolation pattern
- Diverse implementation choices available, such as microservice,
- Shaping good bulkheads is extremely hard
  - Software design issue
  - Needs understanding of SE principles, domain knowledge, and system behavior, future technology evolution, etc…

Monolithic Architecture pattern
(http://microservices.io/patterns/monolithic.html)



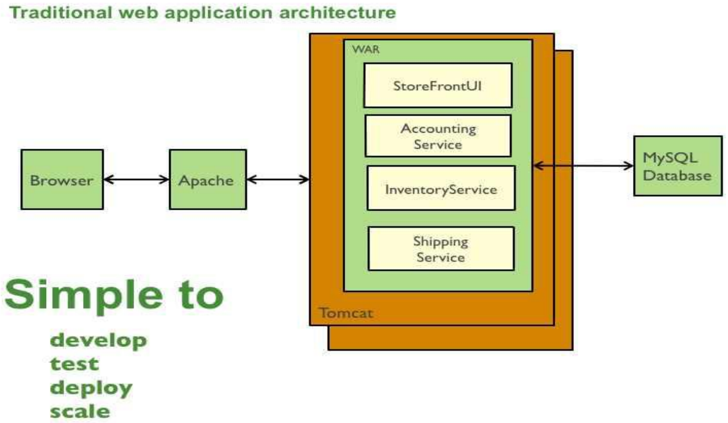
Traditional web application architecture

Simple to

develop
test
deploy
scale

Monolithic Architecture

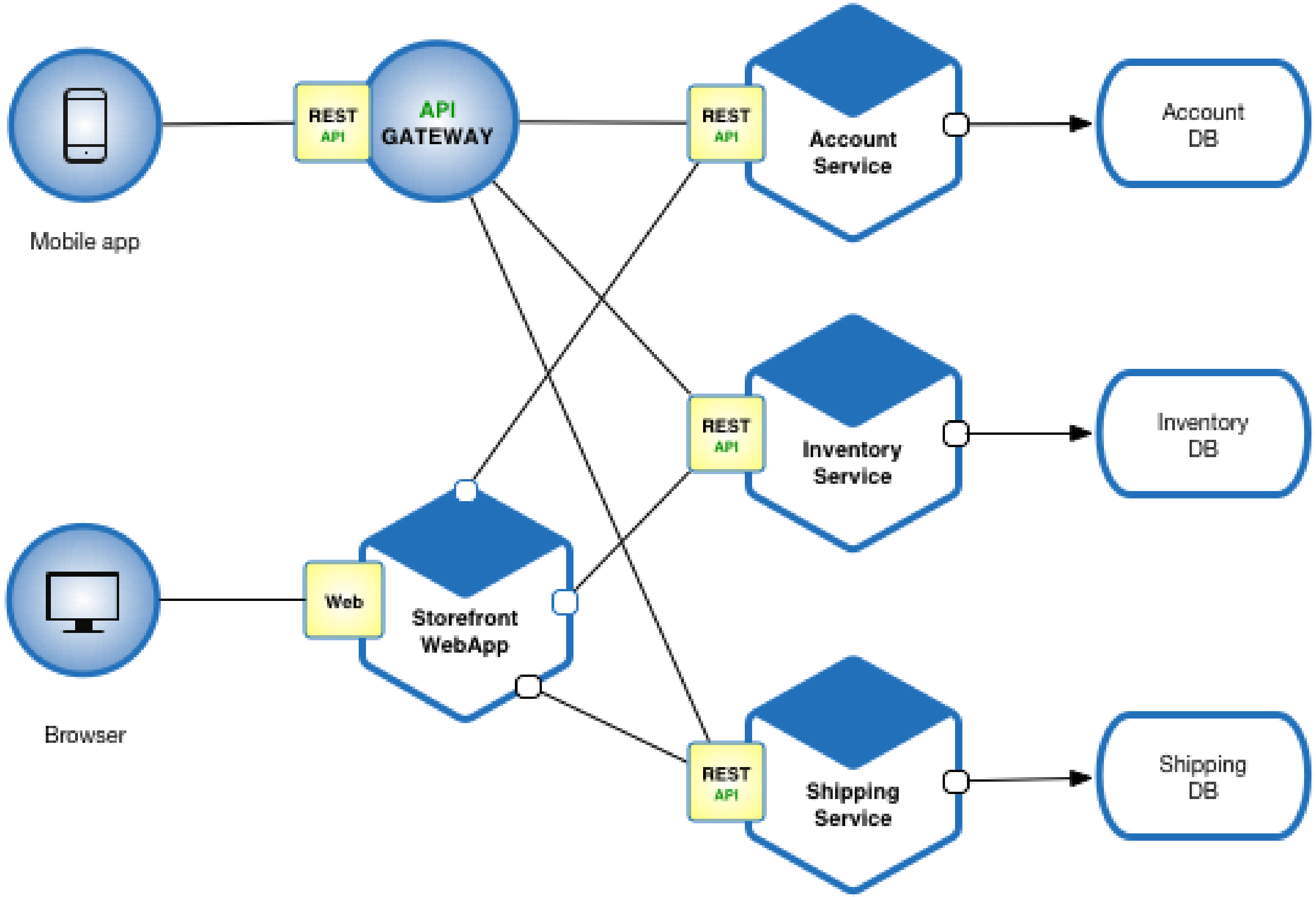
- Benefits
  - Simple to develop – most of current tools support
  - Simple to deploy – deploy WAR file
  - Simple to scale – by running multiple copies
- Drawbacks
  - Difficult to understand
  - Difficult to continuous deployments
  - Requires a long-term commitment to a technology


Traditional web application architecture

Simple to
develop
test
deploy
scale

Microservice Architecture (1/3)

- Partition a system into small manageable pieces, loosely coupled



Microservice Architecture (2/3)

- Benefits
  - Enables continuous delivery and deployment of large, complex applications
  - Organize the development effort with multiple, autonomous teams
  - Easier for a developer to understand
  - Application starts faster, more productive
- Drawbacks
  - Additional complexity of developing a distributed system
  - Difficult to test
  - Deployment complexity
  - Increased memory consumption
    - M (number of different services) times more JVM
  - Difficult to coordinate between teams, multiple services.

Microservice Architecture(3/3)

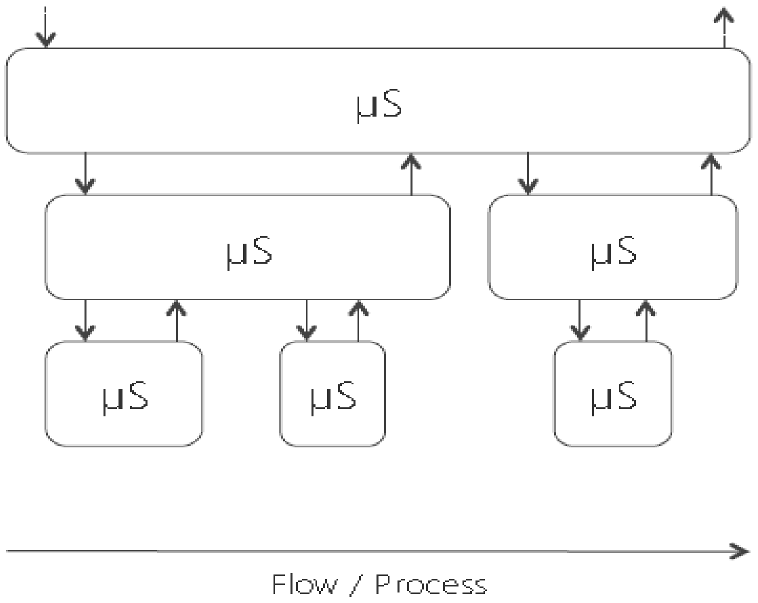
- When to use the microservice architecture
  - Startup?
  - Large-scale service provision?
- How to decompose the application into services
  - In short, it is an 'art'! (design is art!)
  - Some strategies
    - Decompose by business capability
    - Single Responsibility Principle (SRP),
    - Use case,
    - Functional cohesion
- How to maintain data consistency
  - In order to ensure loose coupling, each service has its own database. Then, how to guarantee data inconsistency?
    - Check 'Saga pattern', 'Event sourcing'

Communication Paradigm

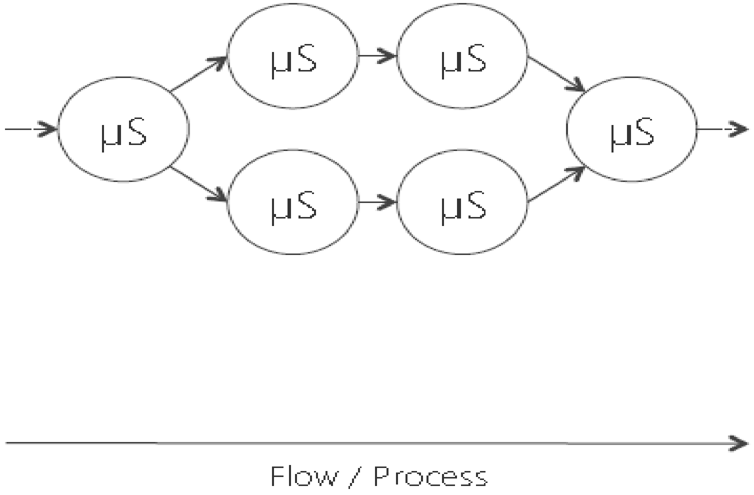
- Heavily influence resilient patterns to be used
- Request-Response vs. Event-Driven
  - Request-Response
  - Event-Driven

Request-Response vs. Event Driven (1/2)

Request/Response : Horizontal slicing

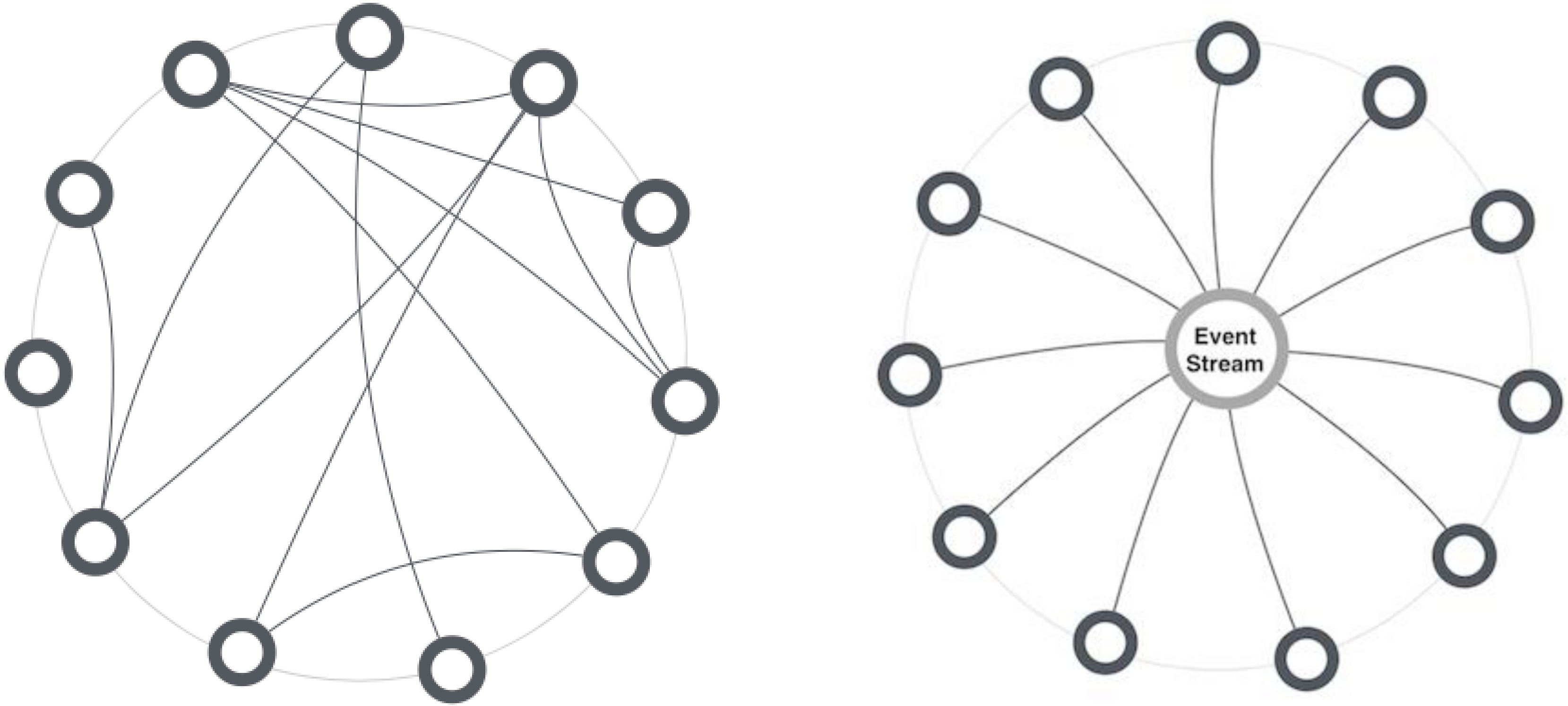


Event-driven : Vertical slicing

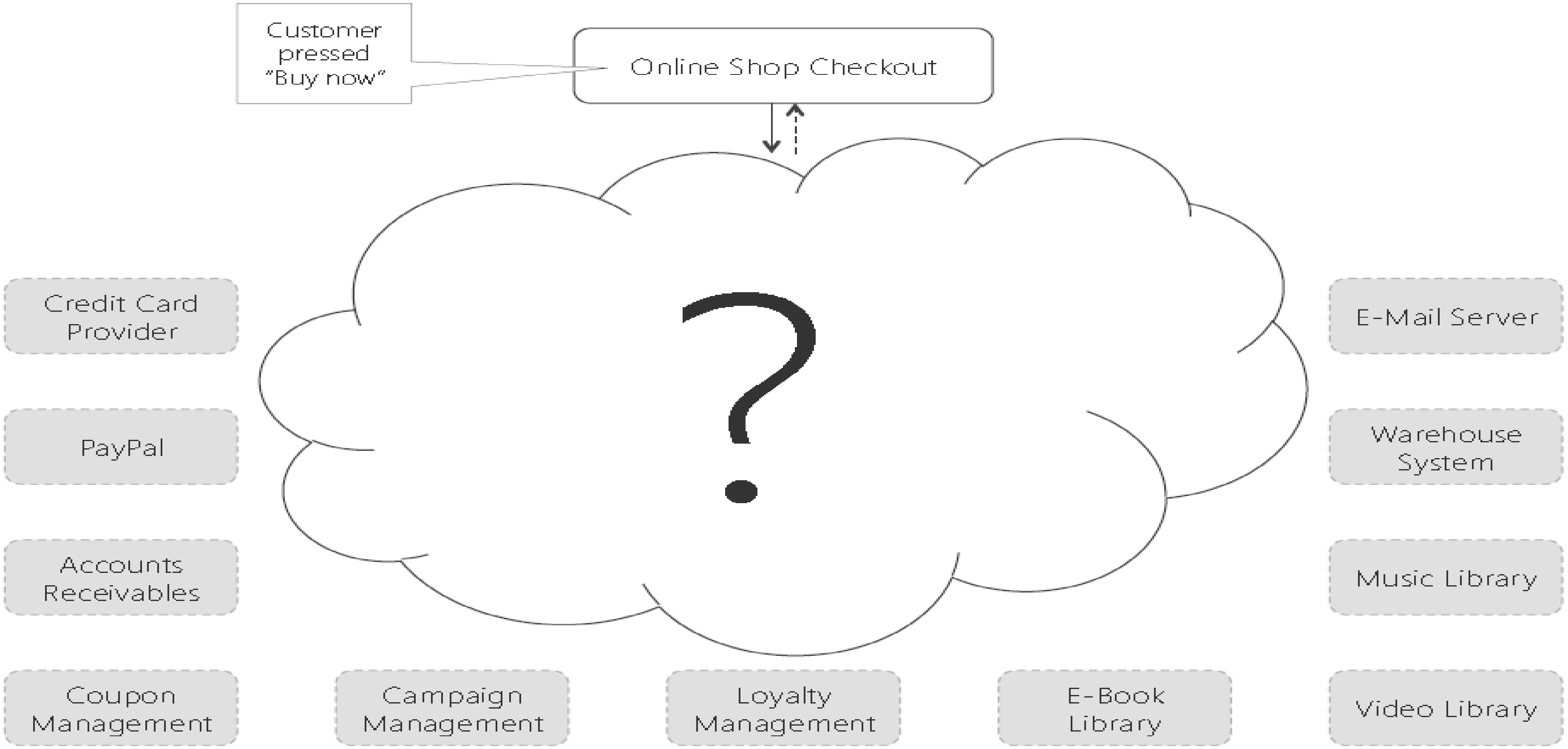


Request-Response vs. Event Driven (2/2) Orchestration vs. Choreography

- Which one looks better?
  - Why?

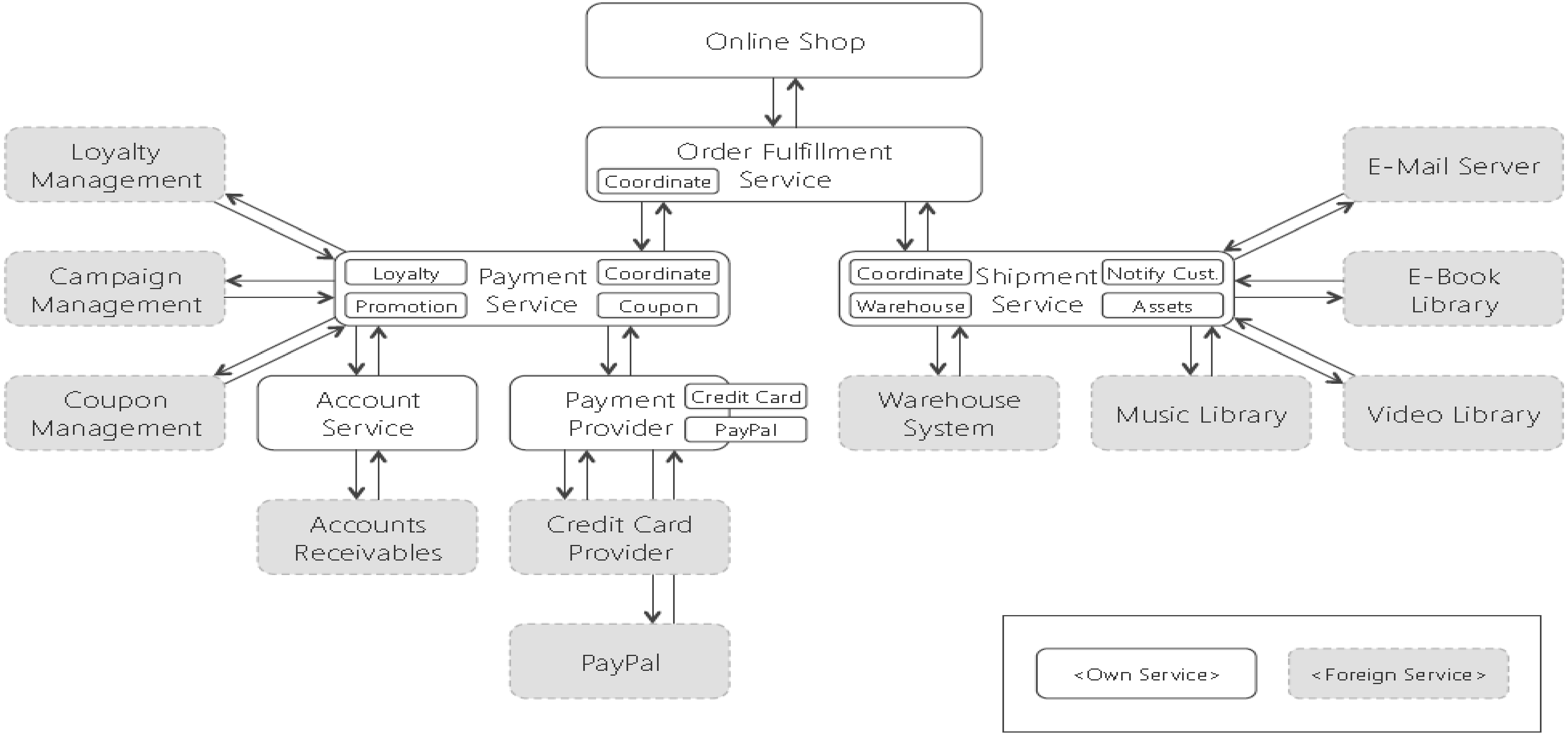


Online Shop Example (1/3)

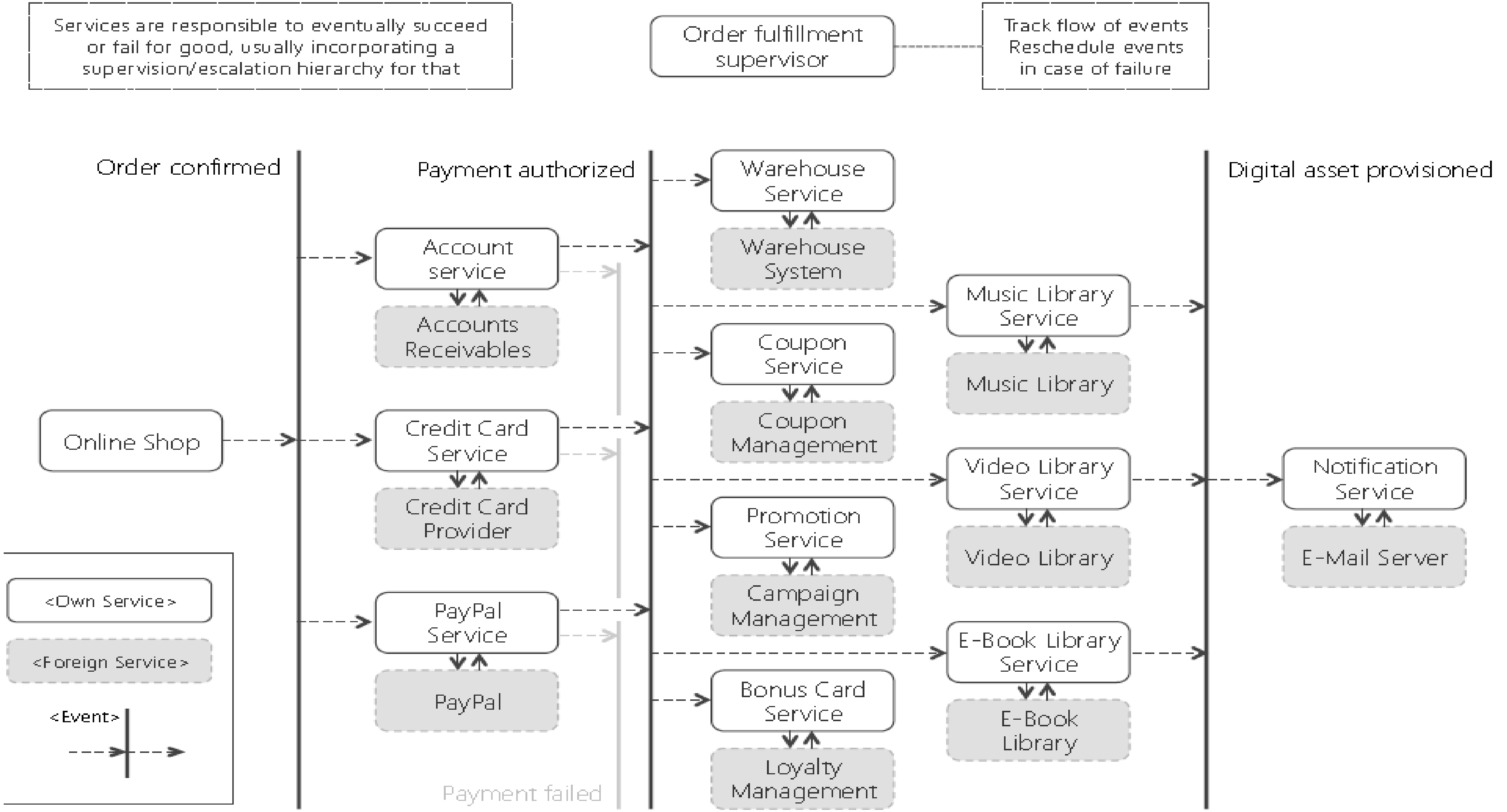


Online Shop Example (2/3)

-



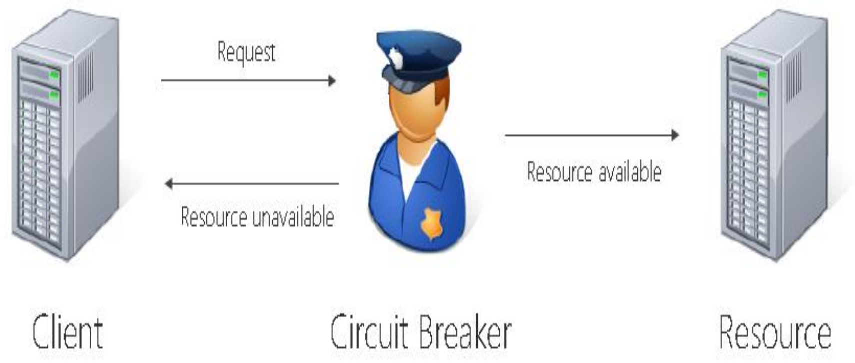
Online Shop Example (3/3)

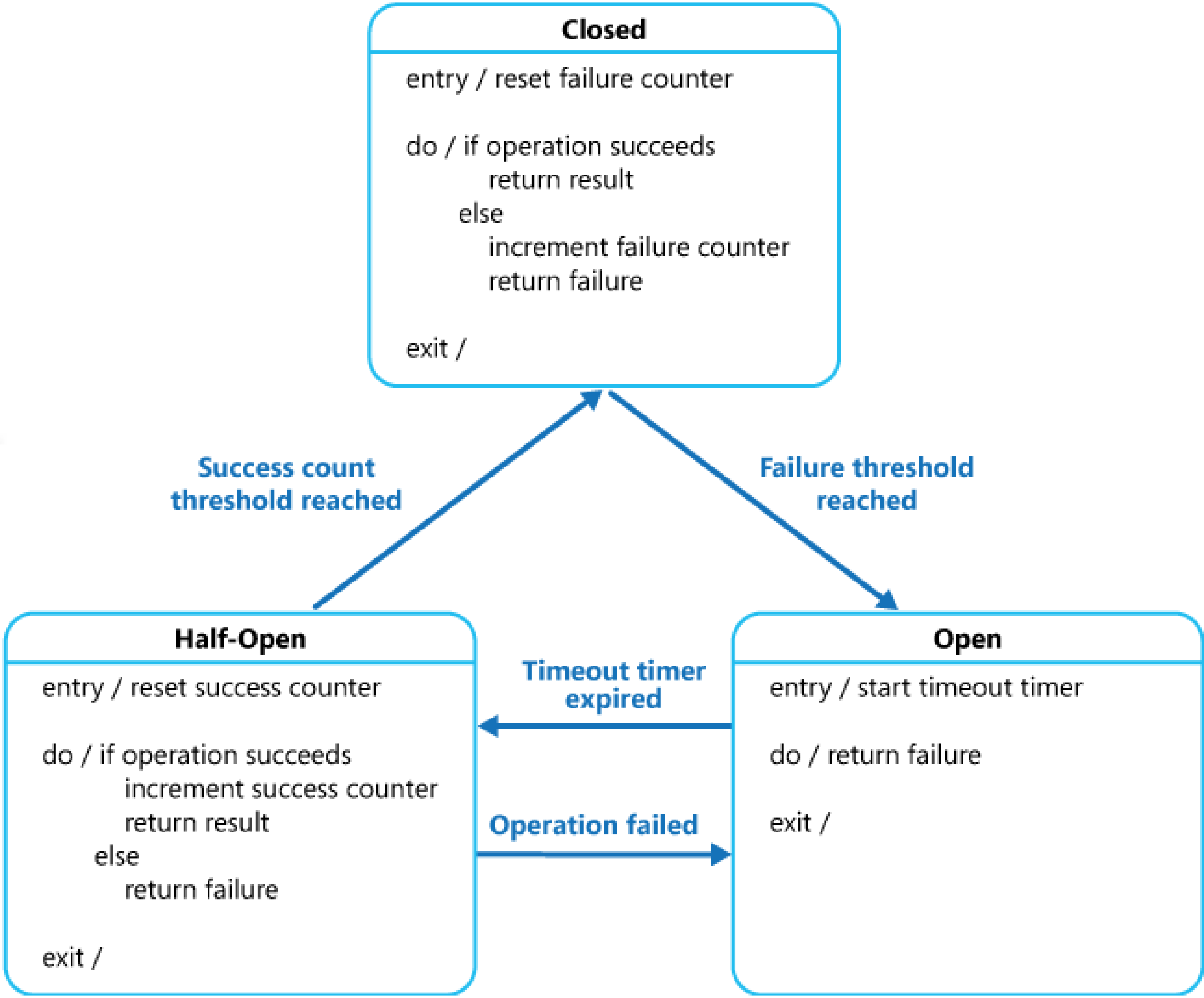


Detect: Circuit Breaker (1/2)

- Most often cited resilient pattern
- Takes downstream unit offline if calls fail multiple times.
- Circuit breaker detects failures and prevents the application from trying to perform the action that is doomed to fail (until it's safe to retry).
- Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource.

Detect: Circuit Breaker(2/2)





Closed

```
entry / reset failure counter

do / if operation succeeds
        return result
     else
        increment failure counter
        return failure

exit /
```

Failure threshold reached

Open

```
entry / start timeout timer

do / return failure

exit /
```

Timeout timer expired

Half-Open

```
entry / reset success counter

do / if operation succeeds
        increment success counter
        return result
     else
        return failure

exit /
```

Operation failed

Success count threshold reached

Prevent: Error Injection

- Inject errors at runtime and observe how the system reacts
  - Chaos engineering at Netflix
- Make sure to inject errors of all types

Complement: Redundancy

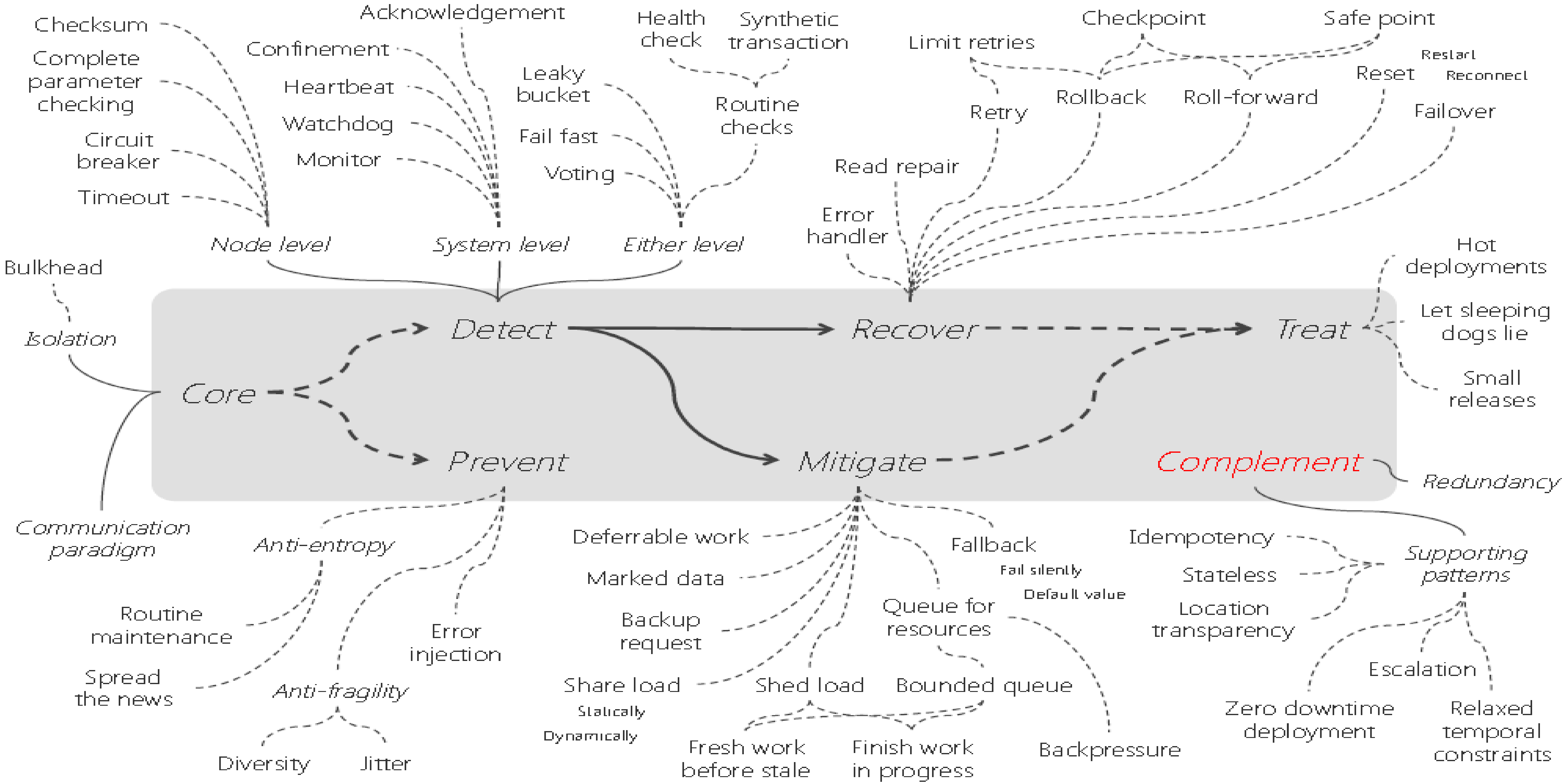
- Core resilient concept
- Basis for many recovery and mitigation patterns
- Often different variants implemented in a system
  - N-version program

Complement: Escalation

- Failed units may not have enough time or information to handle errors
- Escalation peer with more time and information needed
- Separate error handling flow from processing flow
- Often multi-level hierarchies

Treat: Hot deployment

- Hot-deployable services are those which can be added to or removed from the running server. It is the ability to change ON-THE-FLY what's currently deployed without redeploying it.
- Hot deployment is VERY hot for development. The time savings realized when your developers can simply run their build and have the new code auto-deploy instead of build, shutdown, startup is massive.
- Pros: business never stops
- Cons: may require large resources



Checksum

Complete parameter checking

Circuit breaker

Timeout

Acknowledgement

Confinement

Heartbeat

Watchdog

Monitor

Leaky bucket

Fail fast

Voting

Health check

Synthetic transaction

Routine checks

Node level

System level

Either level

Bulkhead

Isolation

Limit retries

Retry

Checkpoint

Safe point

Rollback

Roll-forward

Reset

Restart

Reconnect

Failover

Read repair

Error handler

Hot deployments

Let sleeping dogs lie

Small releases

Core

Detect

Prevent

Recover

Mitigate

Treat

Complement

Redundancy

Communication paradigm

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

Bounded queue

Queue for resources

Fallback

Fail silently

Default value

Backpressure

Idempotency

Stateless

Location transparency

Supporting patterns

Escalation

Zero downtime deployment

Relaxed temporal constraints

Wrap-Up

- Distributed systems are every corner of our society
- Attempts rather to have a fail-free system, better to have a resilient system.
- Resilient SW design patterns (or approaches) need to be mastered for distributed software development (design)
- Try to use of existing ones,
- Even better, “create your own patterns!”