# 10 patterns for more resilient applications

## A gentle start into resilient software design

Uwe Friedrichsen (codecentric AG) – Java Forum Stuttgart – Stuttgart, 7. July 2022

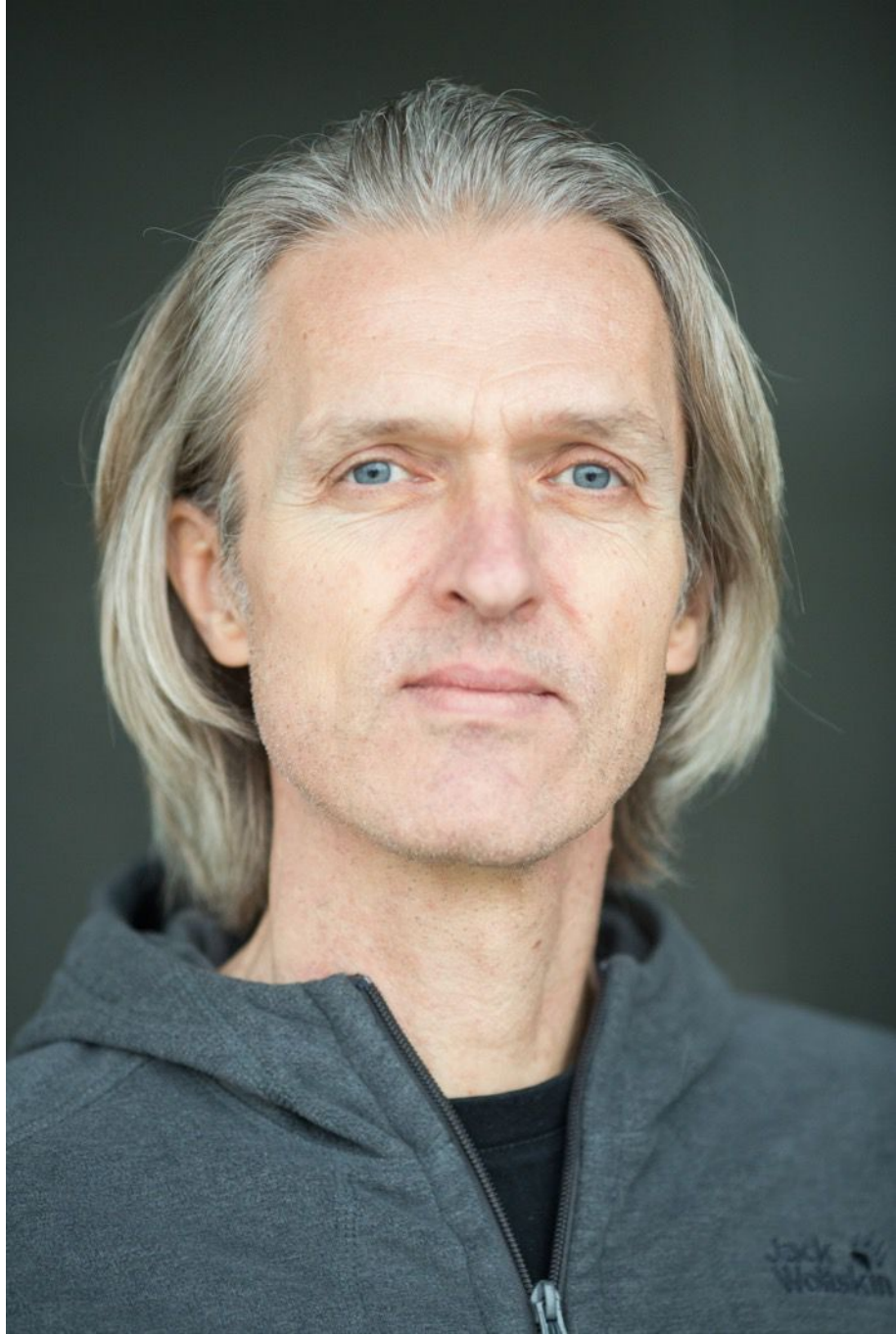
# Uwe Friedrichsen

Works @ codecentric

https://twitter.com/ufried

https://www.speakerdeck.com/ufried

https://ufried.com/




What is that “resilience” thing?

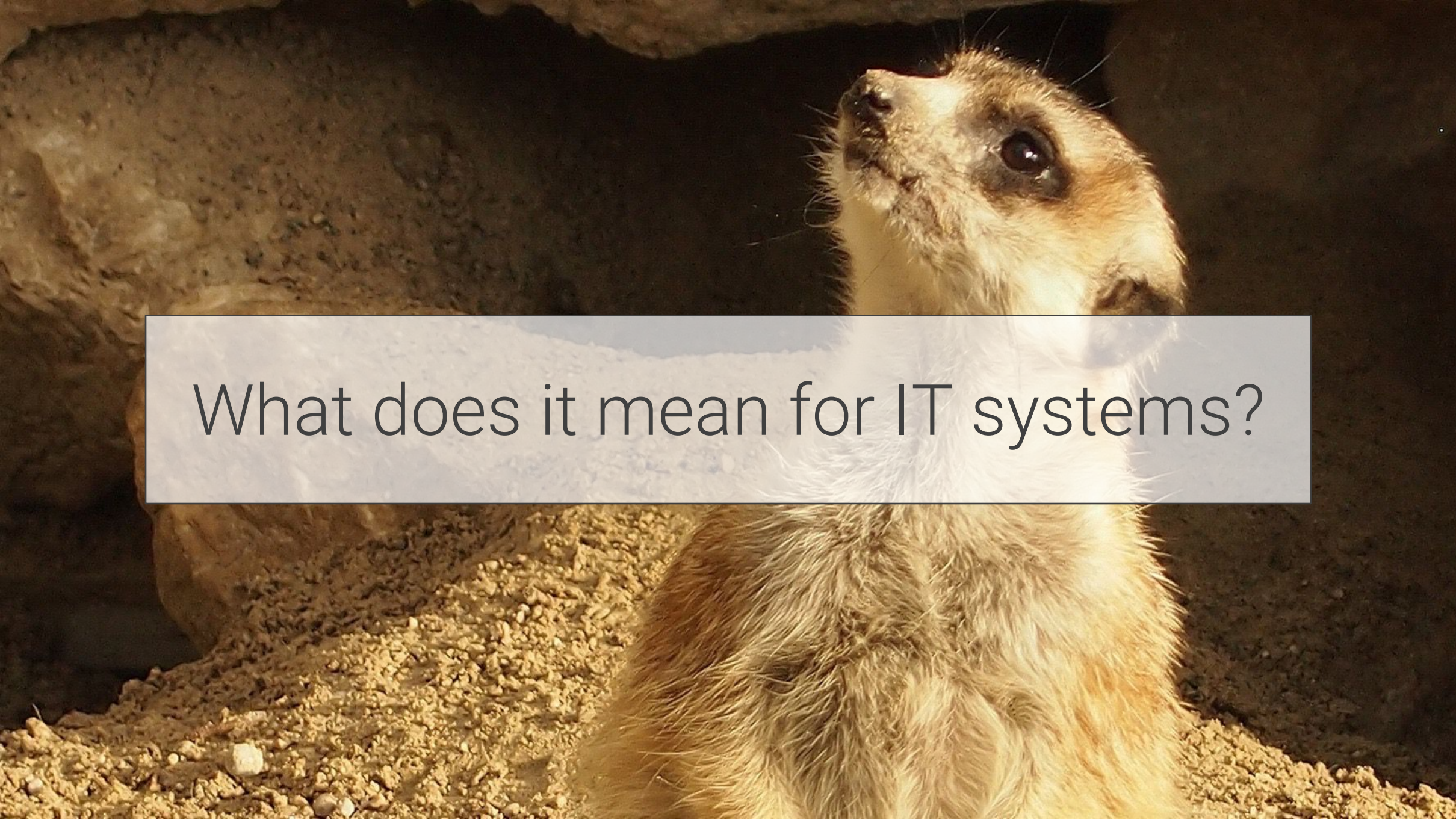
**re·sil·ience** (rĭ-zĭl'yəns)

*n.*

1. The ability to recover quickly from illness, change, or misfortune; buoyancy.
2. The property of a material that enables it to resume its original shape or position after being bent, stretched, or compressed; elasticity.



https://www.thefreedictionary.com/resilience


What does it mean for IT systems?

# **re·sil·ience** (of IT systems)

*n.*

The ability of a system to handle unexpected situations

- without the user noticing it (ideal case)
- with a graceful degradation of service (non-ideal case)

The cautious attempt to provide a useful definition for resilience in the context of software systems.


Can’t we just leave it to ops
as we did it in the past?

What is the problem?

Let ops run our software on some
**HA infrastructure or alike**
and everything will be fine.

Sorry, not that easy anymore


But why?

For a single, monolithic, isolated system
this might indeed work, but …

(Almost) every system is a distributed system.

-- Chas Emerick

http://www.infoq.com/presentations/problems-distributed-systems

The software you develop and maintain is most likely part of a (big) distributed system landscape

# Distributed systems in a nutshell

Everything fails, all the time.

-- Werner Vogels

# Failures in distributed systems ...

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

# … lead to a variety of effects …

- Lost messages
- Incomplete messages
- Duplicate messages
- Distorted messages
- Delayed messages
- Out-of-order message arrival
- Partial, out-of-sync local memory
- …

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

Time & Ordering

Leslie Lamport

*"Time, clocks, and the ordering of events in distributed systems"*

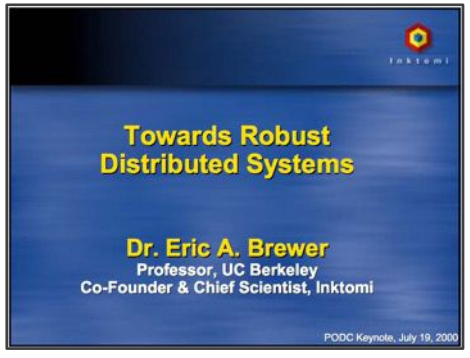
The Part-Time Parliament

Leslie Lamport

Consensus

Leslie Lamport

*"The part-time parliament" (Paxos)*




CAP

Eric A. Brewer

*"Towards robust distributed systems"*

The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

Faulty processes

Leslie Lamport, Robert Shostak, Marshall Pease

*"The Byzantine generals problem"*

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

NANCY A. LYNCH

MICHAEL S. PATERSON

Consensus

Michael J. Fischer, Nancy A. Lynch, Michael S. Paterson

*"Impossibility of distributed consensus with one faulty process" (FLP)*

A Hundred Impossibility Proofs for Distributed Computing

Nancy A. Lynch
Lab for Computer Science
MIT, Cambridge, MA 02139
lynch@theta.lcs.mit.edu

Impossibility

Nancy A. Lynch

*"A hundred impossibility proofs for distributed computing"*

... turning seemingly simple issues into very hard ones

# Embracing distributed systems

- **Distributed systems introduce non-determinism regarding**
  - **Execution completeness**
  - **Message ordering**
  - **Communication timing**

- You will be affected by this at the application level
  - Don’t expect your infrastructure to hide all effects from you
  - Better have a plan to detect and recover from inconsistencies


Okay, I buy it. But how do I start?

Let us start simple … *

* which often improves the situation amazingly much

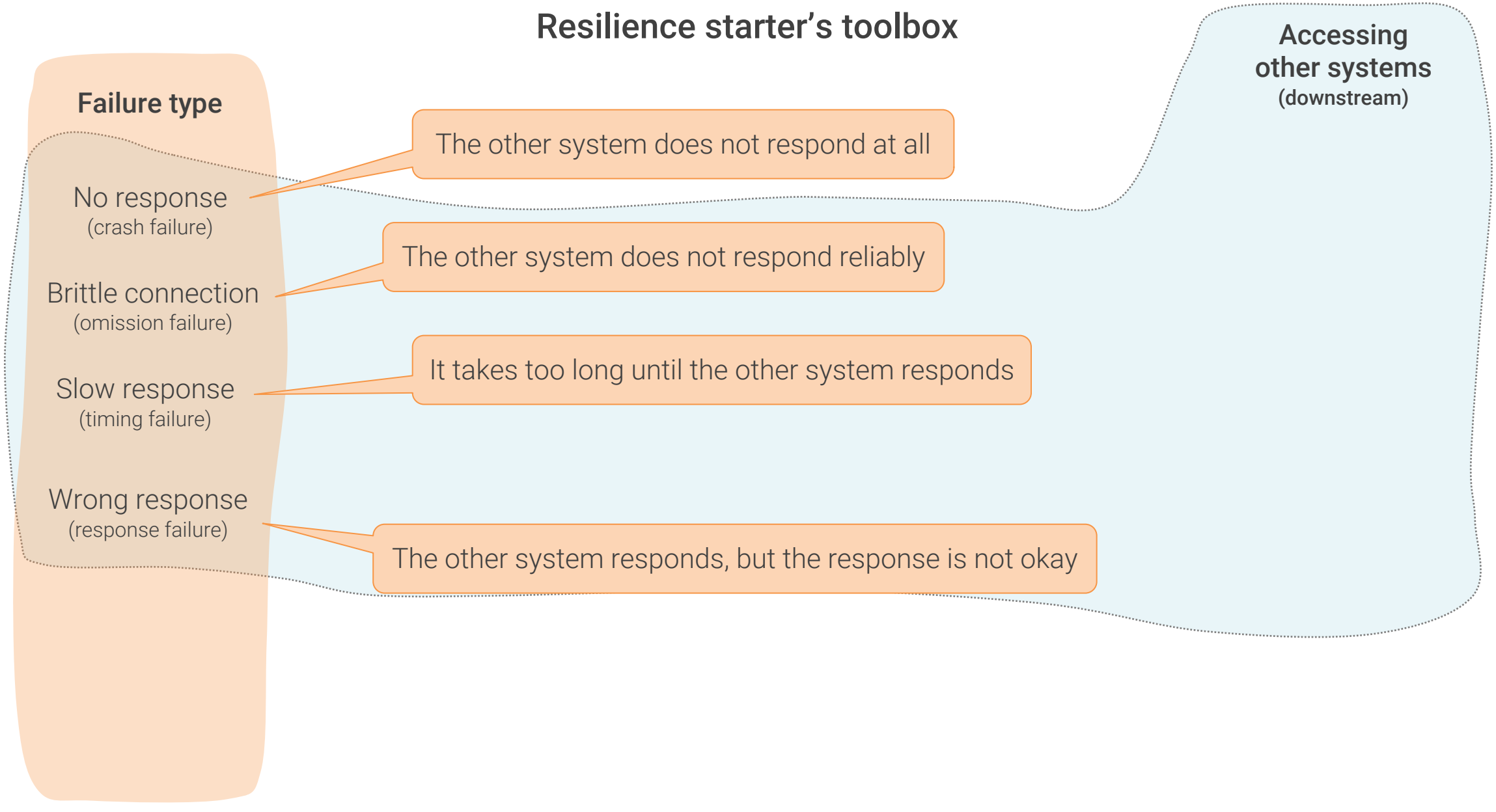

Let us create our starter’s toolbox

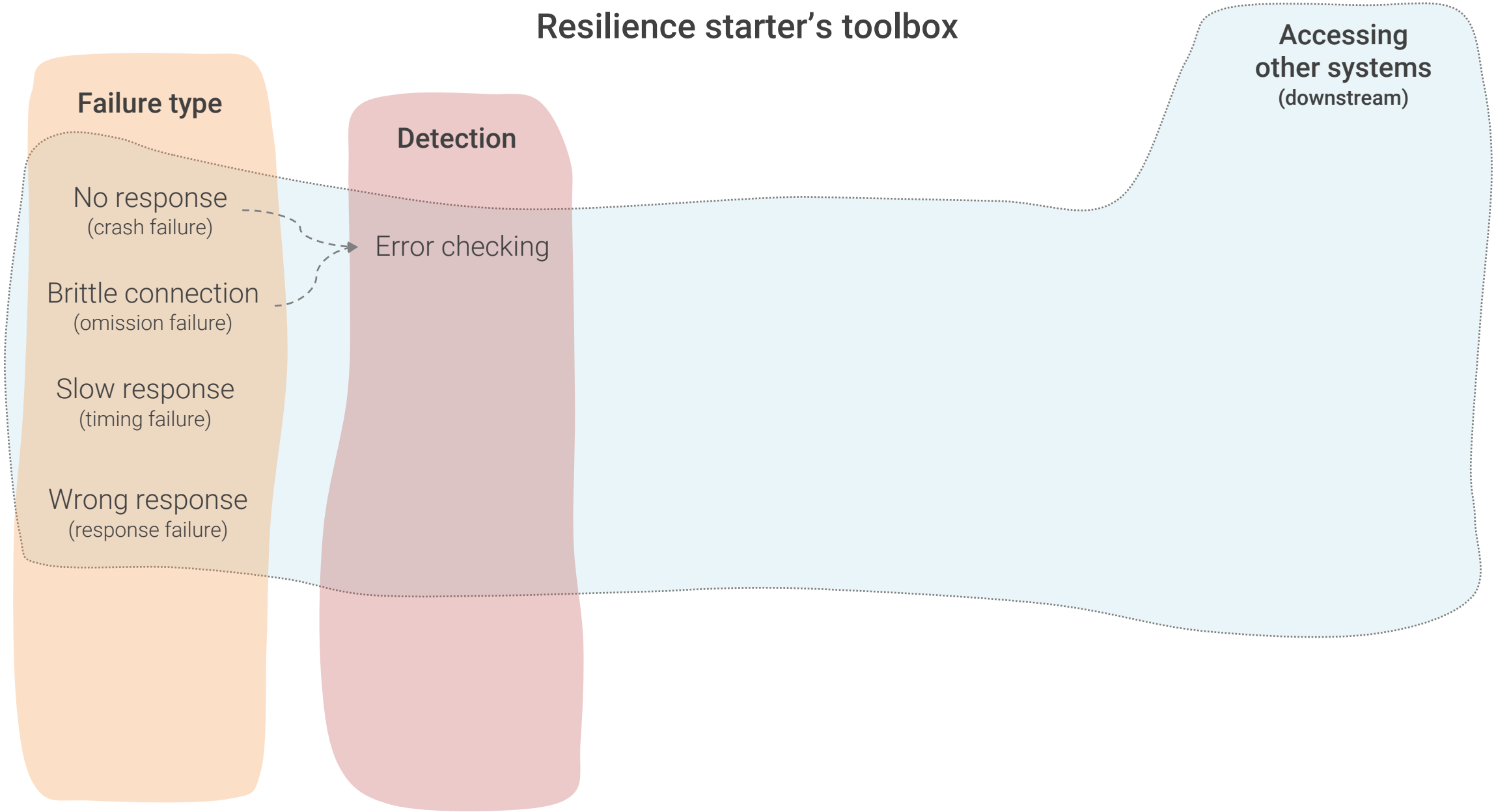
# Resilience starter’s toolbox

# Accessing other systems

```
from urllib3 import PoolManager

URL = <…>

http = PoolManager()
r = http.request('GET’, URL)
```

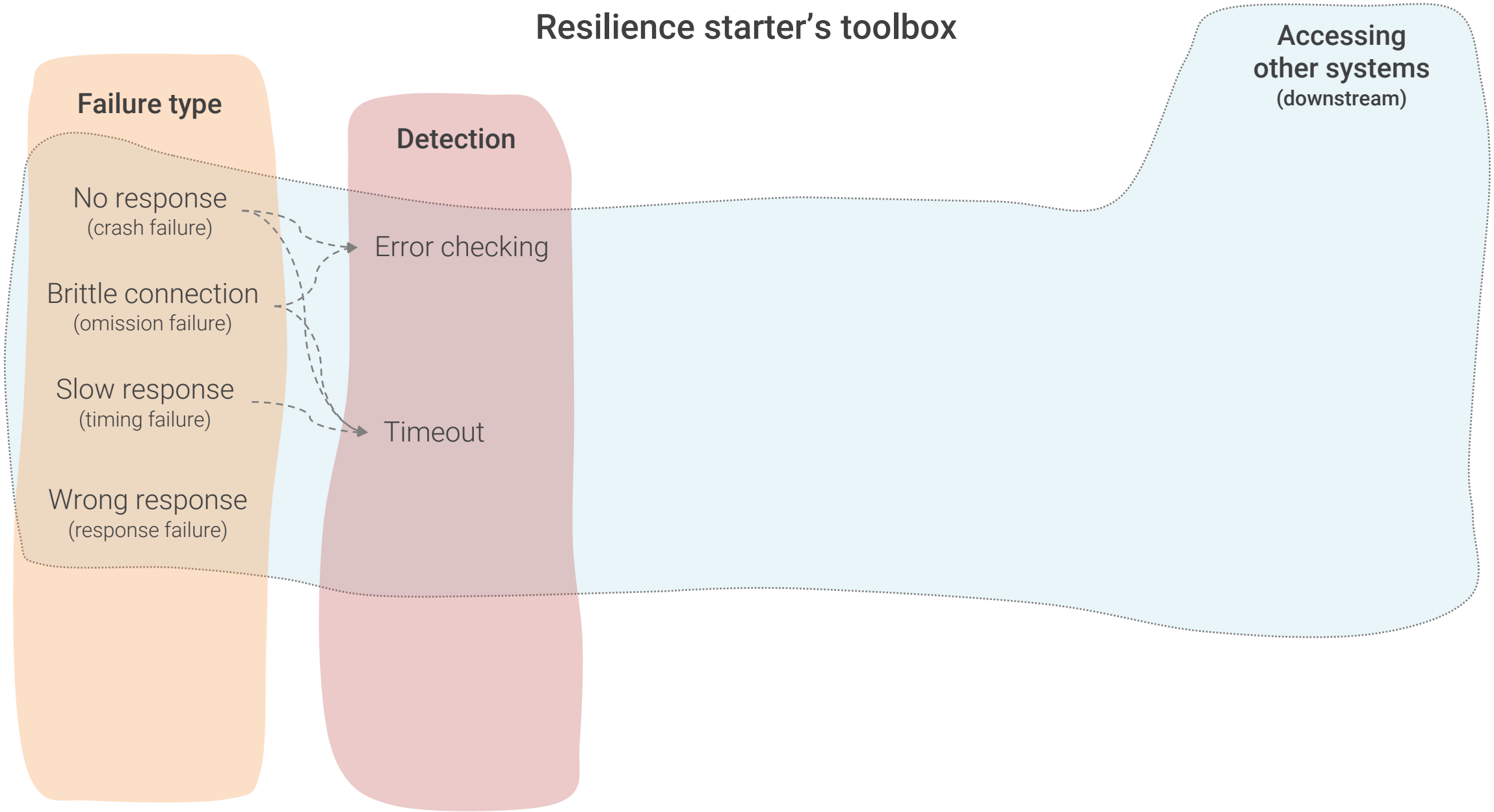
https://github.com/urllib3/urllib3


Resilience starter's toolbox
Accessing other systems (downstream)
Failure type
No response (crash failure)
Brittle connection (omission failure)
Slow response (timing failure)
Wrong response (response failure)
The other system does not respond at all
The other system does not respond reliably
It takes too long until the other system responds
The other system responds, but the response is not okay


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Detection
Error checking
Accessing
other systems
(downstream)

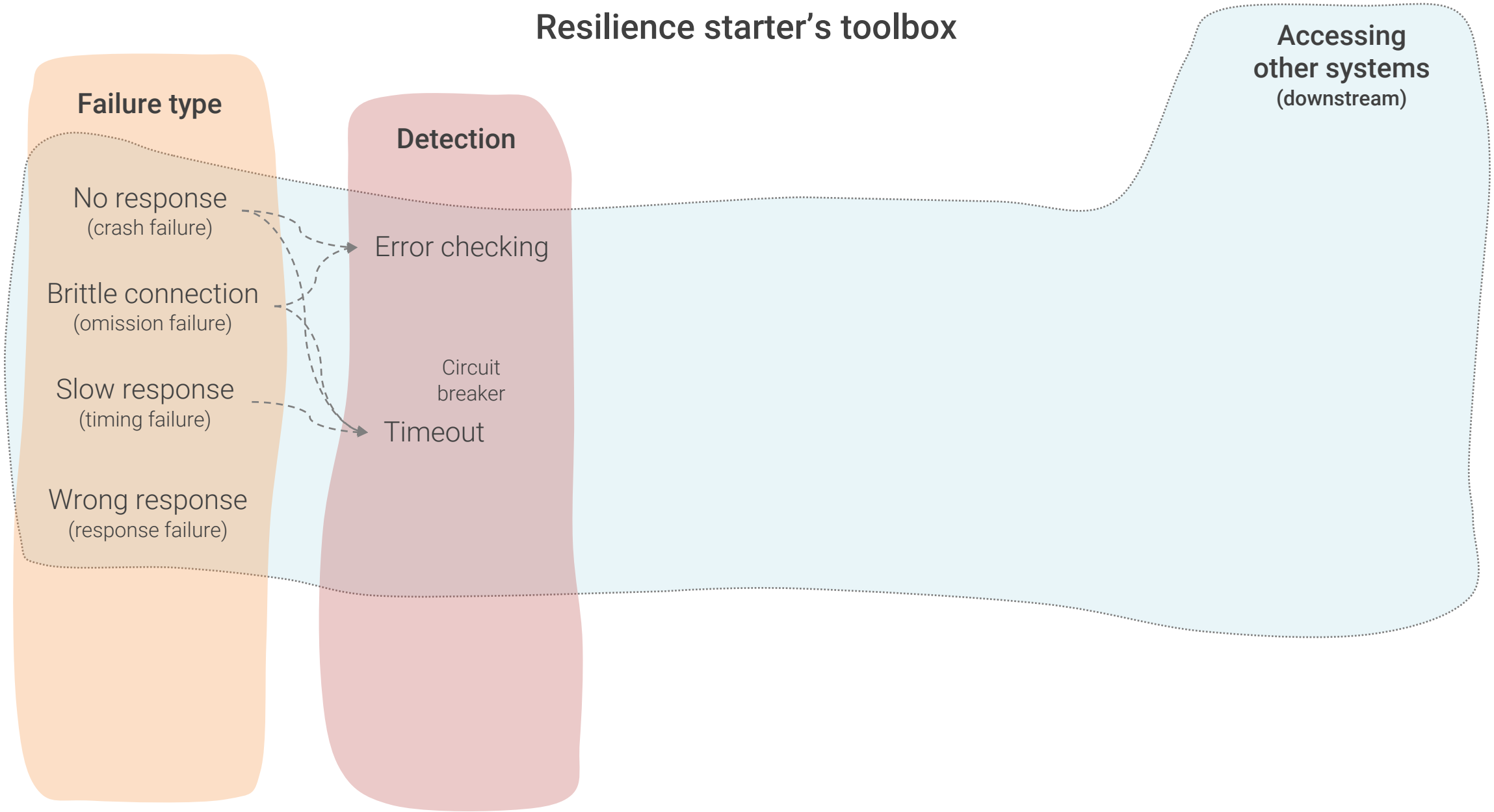
# Error checking

- Most basic error detection pattern
- Yet too often neglected
- Multiple implementation variants
  - Exception handling (Java, C++, …)
  - Return code checking (C, …)
  - Extra error return value (Go, …)
- Thorough error checking tends to make code harder to read


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Detection
Error checking
Timeout
Accessing
other systems
(downstream)

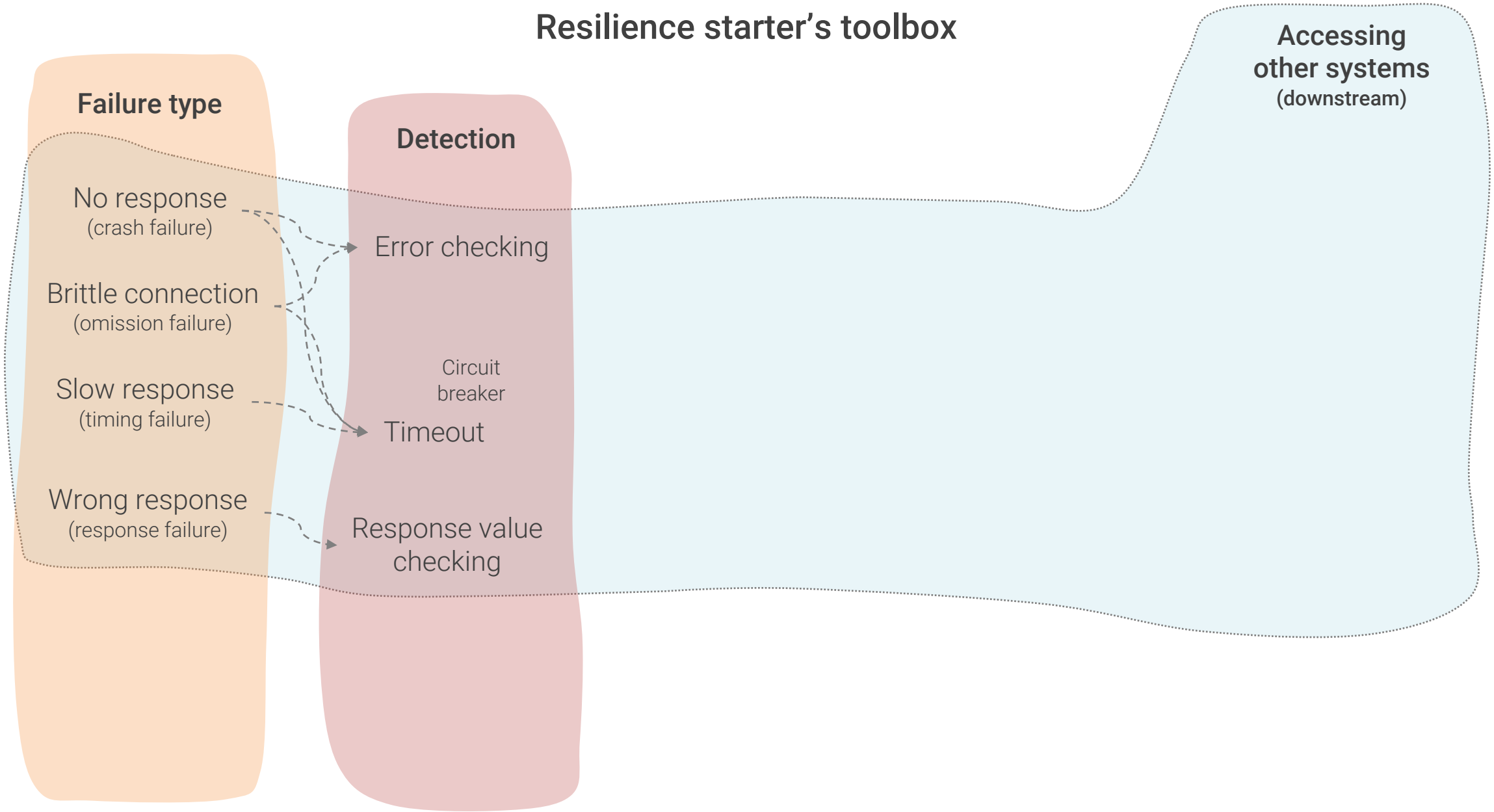
# Timeout

- Preserve responsiveness independent of downstream latency
- Essential error detection pattern
- Crucial if using synchronous communication
- Also needed if using asynchronous request/response style
- Good library support in most programming languages


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Detection
Error checking
Circuit
breaker
Timeout
Accessing
other systems
(downstream)

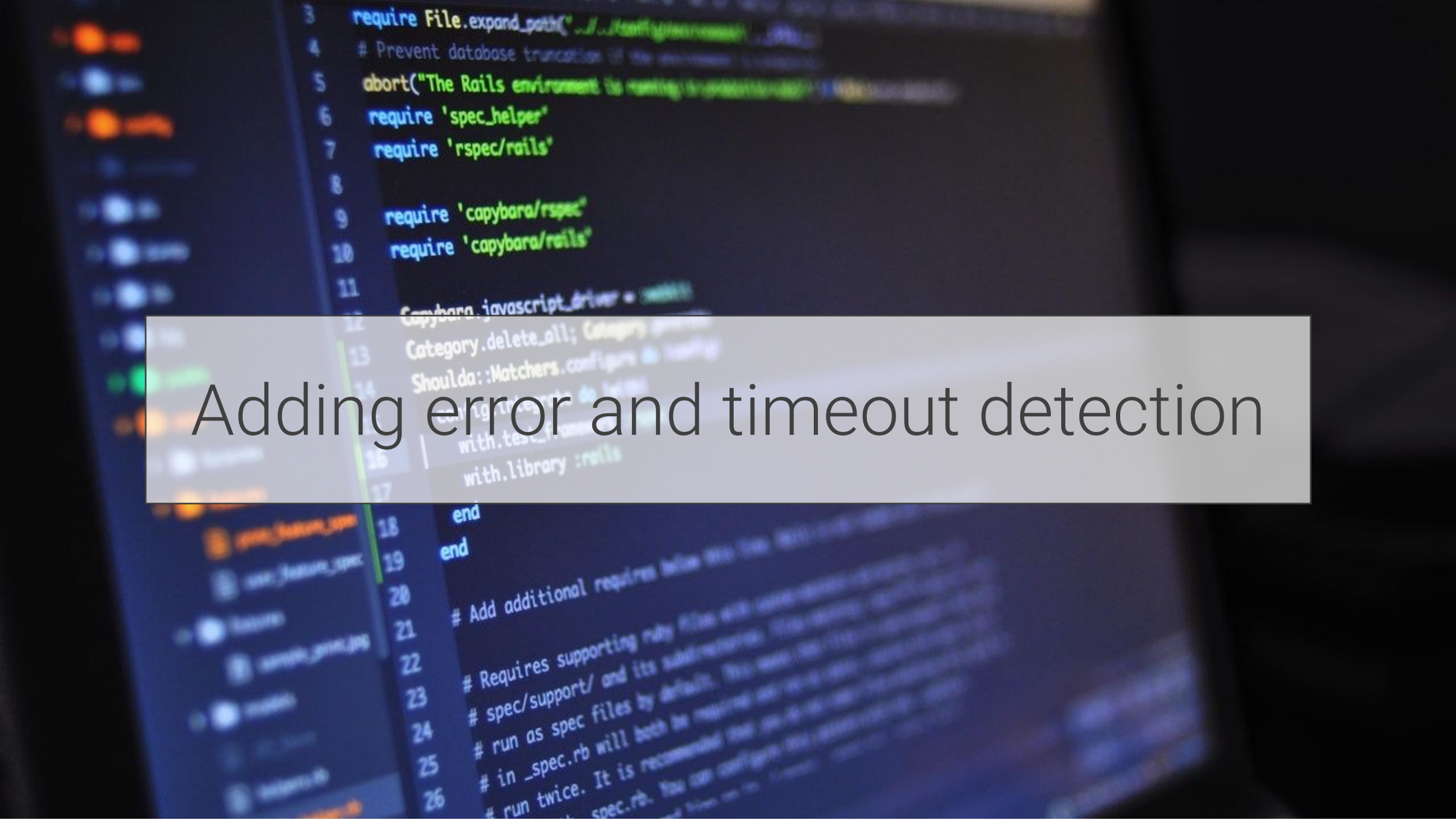
# Circuit breaker

- Probably most often cited resilience pattern
- Extension of the timeout pattern
- Takes downstream unit offline if calls fail multiple times
- Can be used for most failure types
    - Crash failures, omission failure, timing failures
- Many implementations available


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Detection
Error checking
Circuit
breaker
Timeout
Response value
checking
Accessing
other systems
(downstream)

# Response value checking

- As obvious as it sounds, yet often neglected
- Protection from broken/malicious return values
  - Especially do not forget to check for Null values
- Quite good library support
  - But often do not cover all checks needed
- Consider specific data types



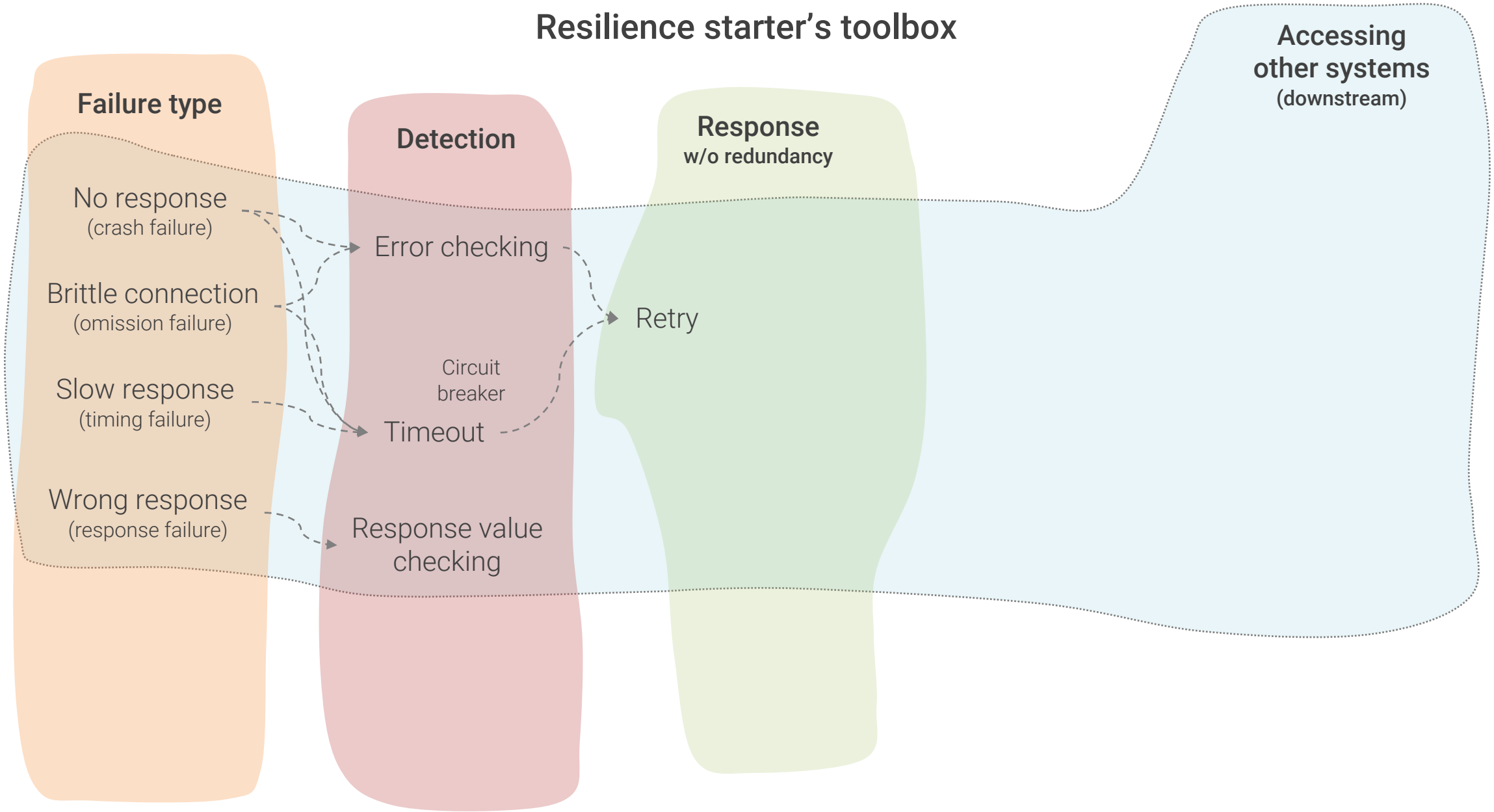

# Adding error and timeout detection

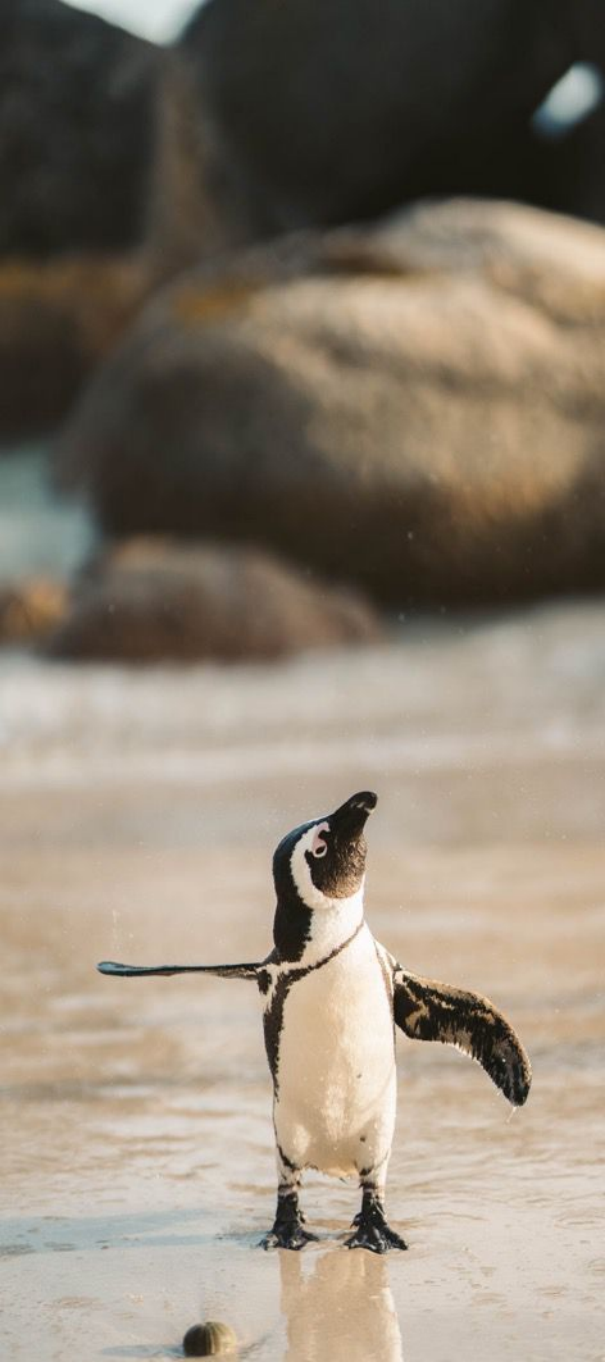
```
from urllib3 import PoolManager

URL = <…>

http = PoolManager()
r = http.request('GET’, URL)
```

https://github.com/urllib3/urllib3

```
from concurrent.futures import ThreadPoolExecutor, TimeoutError
from urllib3 import PoolManager
from urllib3.exceptions import HTTPError

URL = 'http://httpbin.org/delay/2'

def get_url(http, url):
    return http.request('GET', url)

http = PoolManager()

with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(get_url, http, URL)
    try:
        r = future.result(timeout=0.5)
    except TimeoutError:
        print('Request timed out')
        future.cancel()
    except HTTPError:
        print('An error occurred')
    else:
        print('Received:', r.data)
```

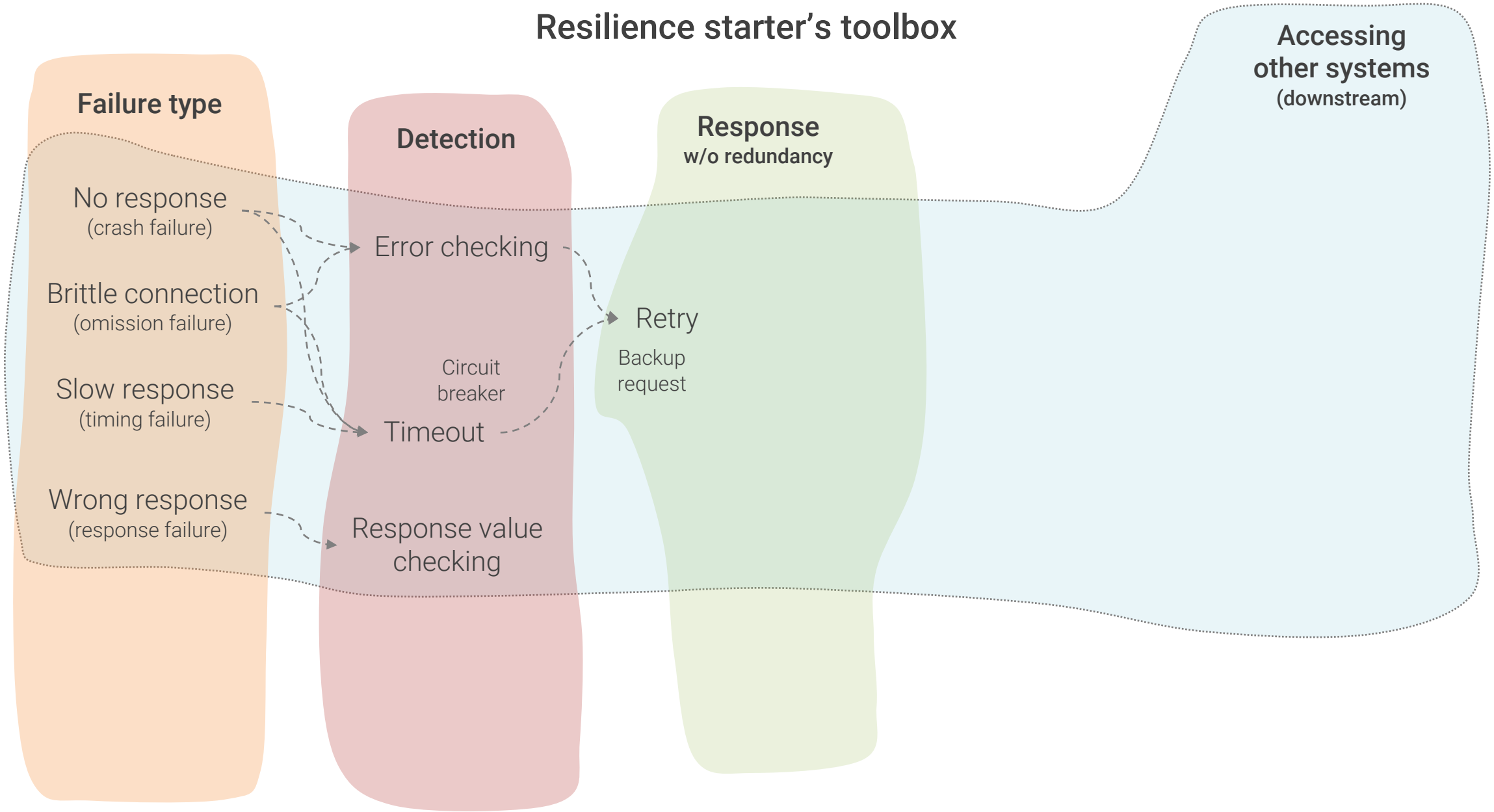
```
from urllib3 import PoolManager
from urllib3.exceptions import HTTPError

URL = 'http://httpbin.org/delay/2'

http = PoolManager()

try:
    r = http.request('GET', URL, timeout=0.5)
except HTTPError:
    print('An error occurred or request timed out')
else:
    print('Received:', r.data)
```


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Detection
Error checking
Circuit
breaker
Timeout
Response value
checking
Response
w/o redundancy
Retry
Accessing
other systems
(downstream)



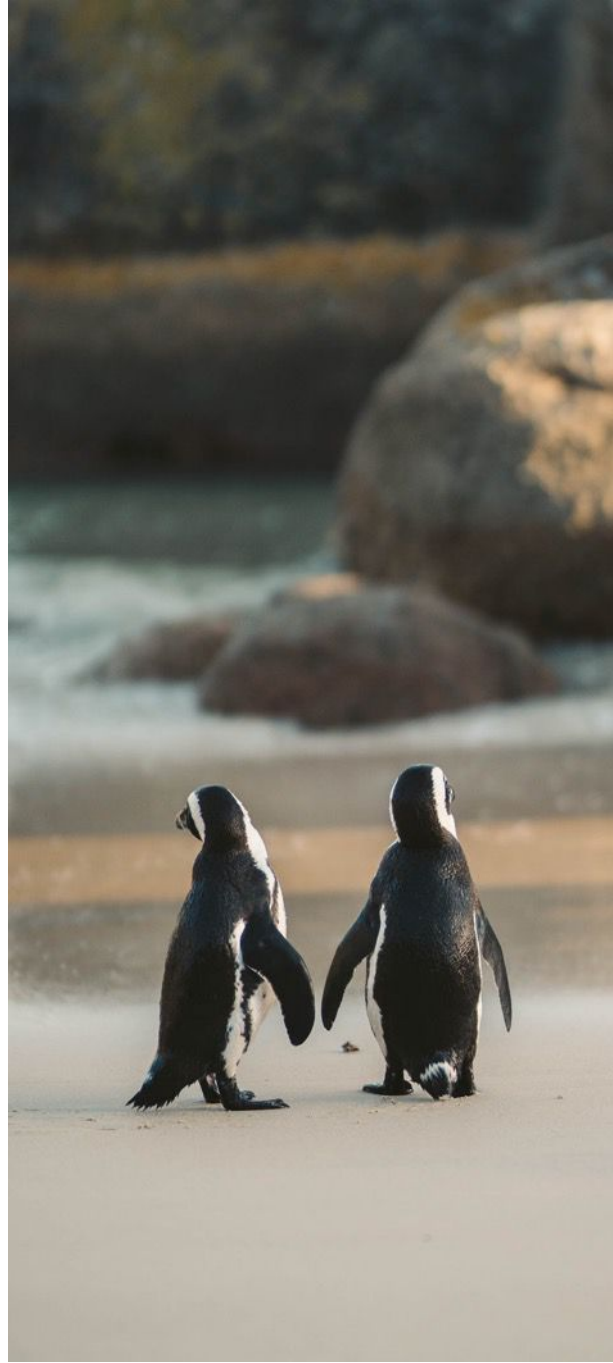
# Retry

- Basic recovery pattern for downstream calls
- Recover from omission or other transient errors
- Limit retries to minimize extra load on an overloaded resource
- Limit retries to avoid recurring errors
- Some library support available

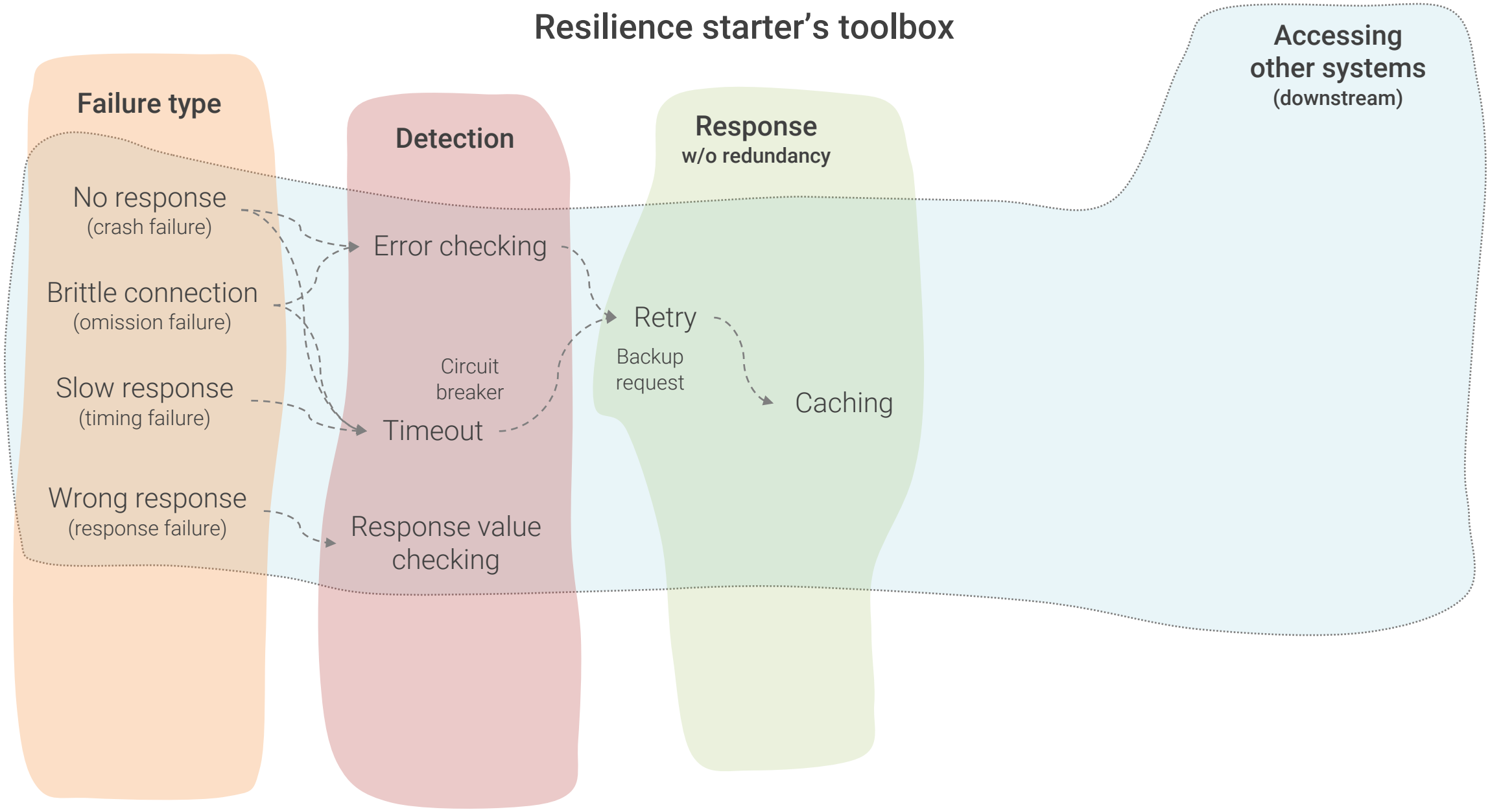

Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Detection
Error checking
Circuit
breaker
Timeout
Response value
checking
Response
w/o redundancy
Retry
Backup
request
Accessing
other systems
(downstream)

# Backup request

- Send request to multiple workers (usually with some delay)
- Use quickest reply and discard all other responses
- Prevents latent responses (or at least reduces probability)
- Requires redundancy – trades resources for availability

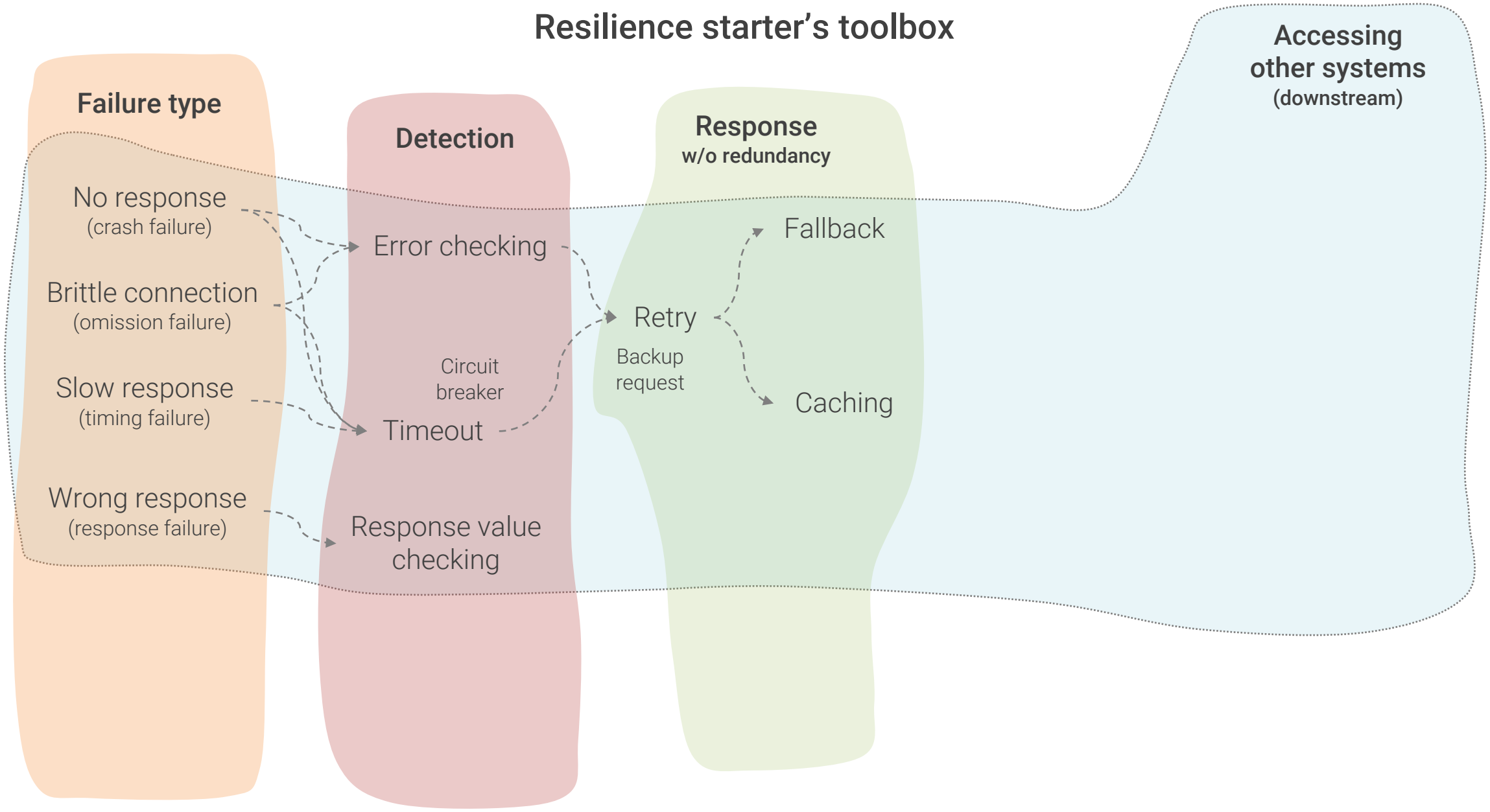


also see: J. Dean, L. A. Barroso, “The tail at scale”, Communications of the ACM, Vol. 56 No. 2


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Detection
Error checking
Circuit
breaker
Timeout
Response value
checking
Response
w/o redundancy
Retry
Backup
request
Caching
Accessing
other systems
(downstream)

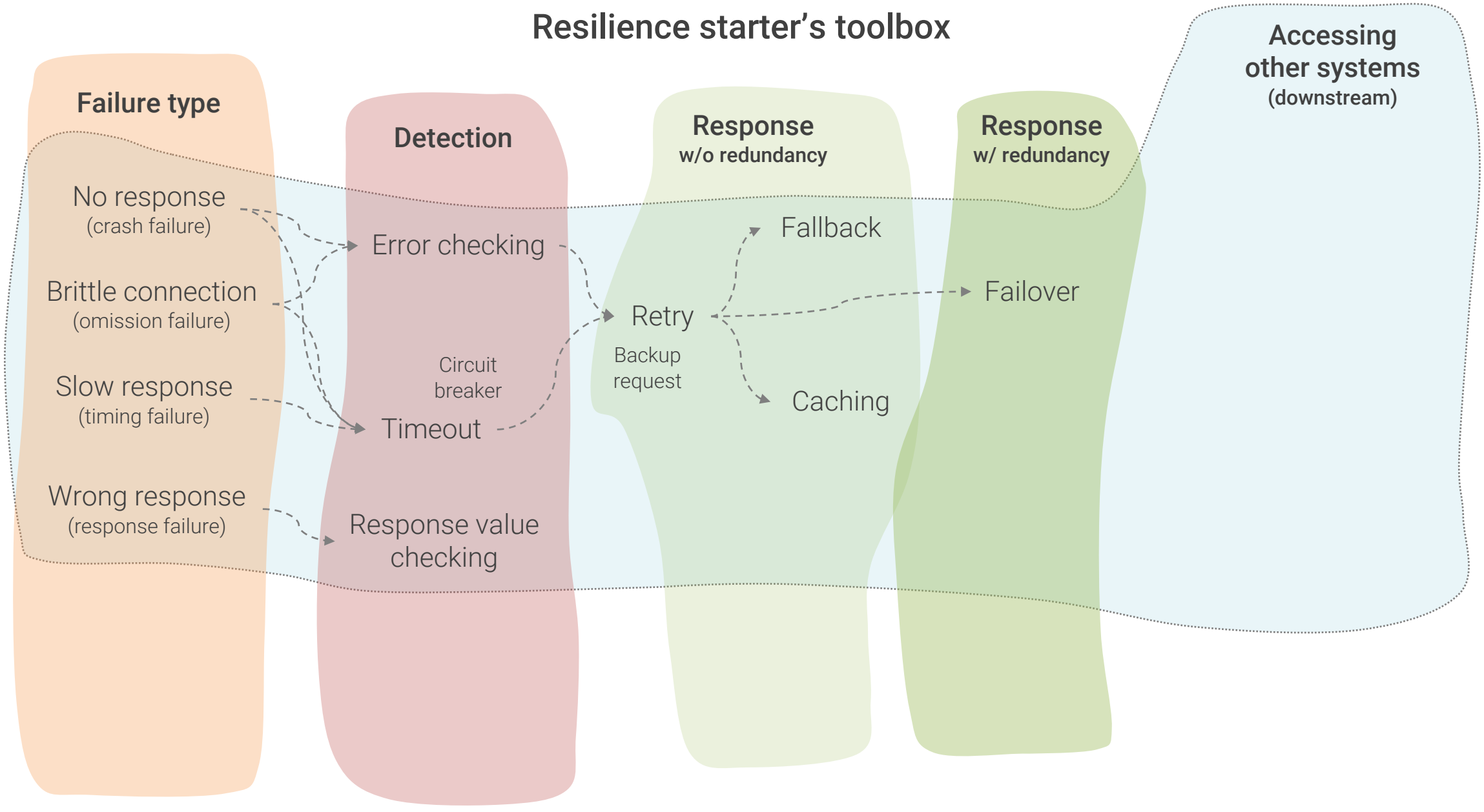
# Caching

- Re-use responses from prior calls to downstream resources
- Can bridge temporary unavailability of resources
- Use with caution
  - Requires extra resources to store cached data
  - Leaves you with potentially stale data
    and all consistency issues associated with it
- Good tool and library support


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Detection
Error checking
Circuit
breaker
Timeout
Response value
checking
Response
w/o redundancy
Retry
Backup
request
Fallback
Caching
Accessing
other systems
(downstream)

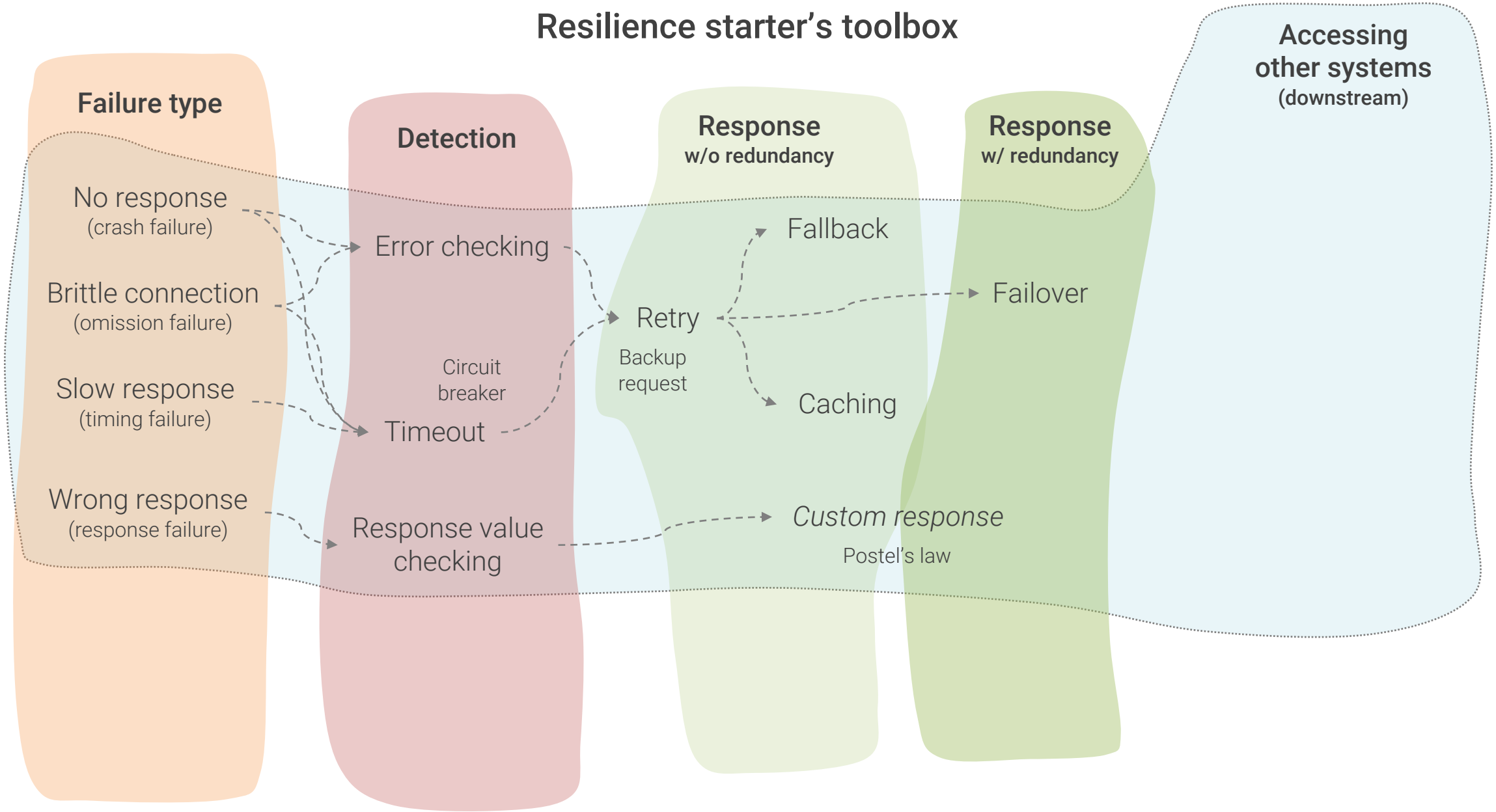
# Fallback

- Execute an alternative action if the original action fails
- Basis for most mitigation patterns
- Widespread simple variants
    - *Fail silently*: silently ignore error and continue processing
    - *Default value*: return predefined default value if error occurs
- Note that fallback action is a business decision


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Detection
Error checking
Circuit
breaker
Timeout
Response value
checking
Response
w/o redundancy
Retry
Backup
request
Fallback
Caching
Response
w/ redundancy
Failover
Accessing
other systems
(downstream)

# Failover

- Used if simpler recovery measures fail or take too long
- Many implementation variants available
- Good support on the infrastructure level
  - Recovery and state replication usually not covered
- Mind the business case
  - Requires redundancy – trades resources for availability
  - Added costs need to justify added value


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Detection
Error checking
Circuit
breaker
Timeout
Response value
checking
Response
w/o redundancy
Retry
Backup
request
Fallback
Caching
Response
w/ redundancy
Failover
Custom response
Postel's law
Accessing
other systems
(downstream)

# Remember Postel's law

"Be conservative in what you do,
be liberal in what you accept from others"

(Often reworded as: "Be conservative in what you send, be liberal in what you accept")

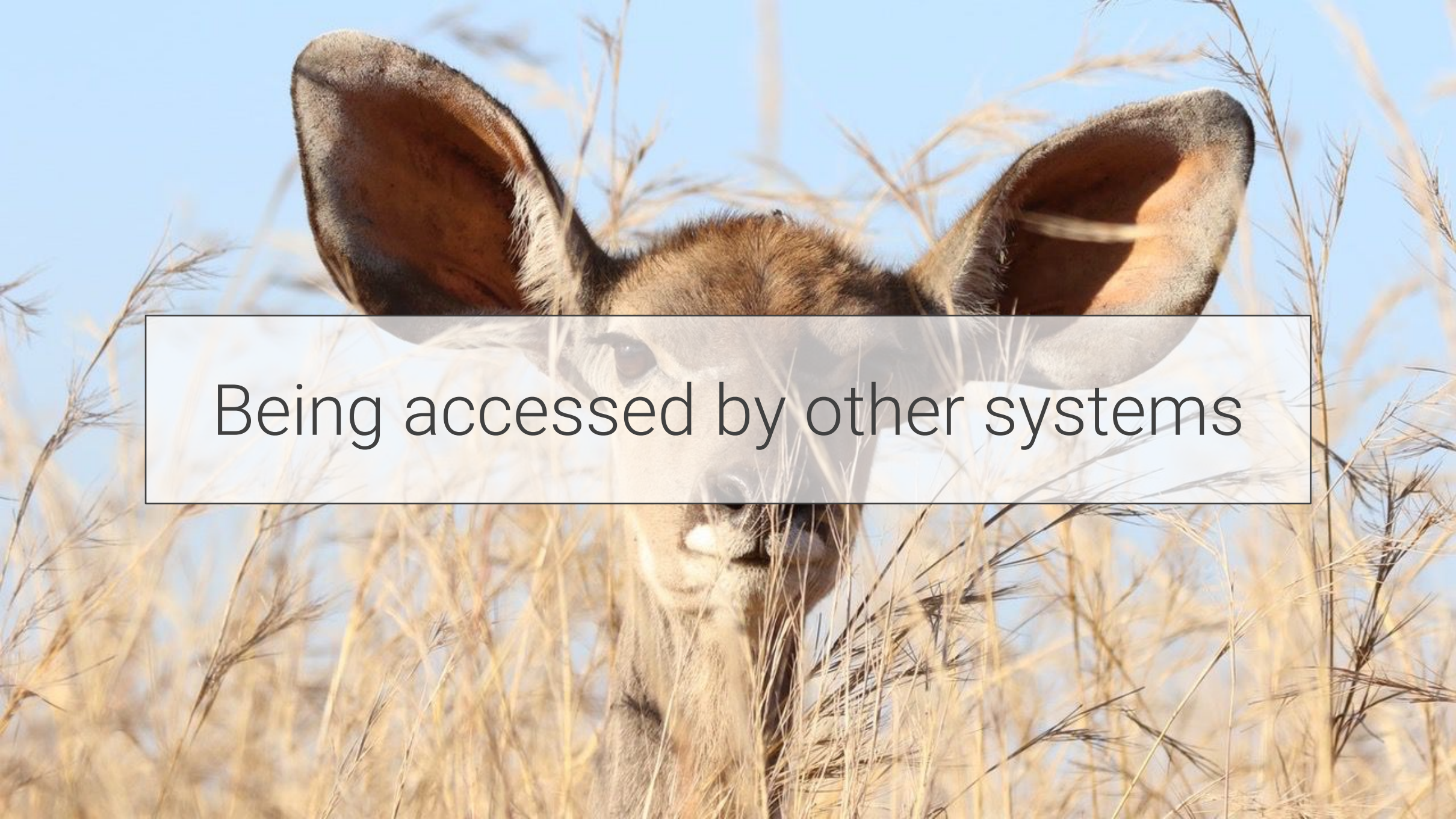
see also: https://en.wikipedia.org/wiki/Robustness_principle


Adding retry and fallback

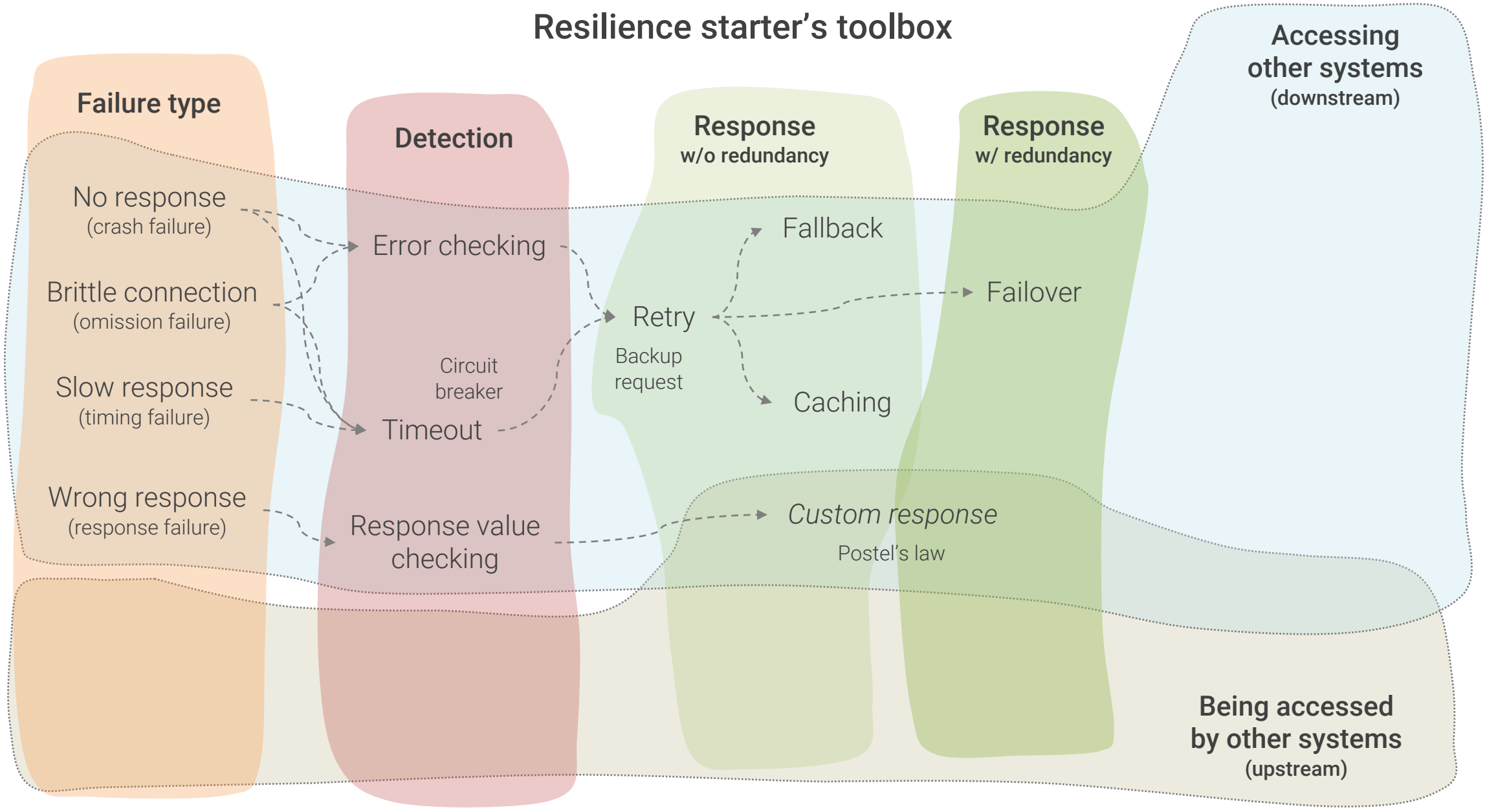
```
from urllib3 import PoolManager
from urllib3.exceptions import HTTPError

URL = 'http://httpbin.org/delay/2'

http = PoolManager()

try:
    r = http.request('GET', URL, timeout=0.5)
except HTTPError:
    print('An error occurred or request timed out')
else:
    print('Received:', r.data)
```

```
from urllib3 import PoolManager
from urllib3.exceptions import HTTPError

URL = 'http://httpbin.org/delay/2'

http = PoolManager()

def get_url(http, url):
    try:
        r = http.request('GET', url, timeout=0.5)
    except HTTPError:
        return None  # None means something went wrong
    else:
        return r.data

d = get_url(http, URL)
if d is None:
    d = get_url(http, URL)  # Retry once
if d is None:
    d = 42  # Execute fallback
print('Received:', d)
```

```
from urllib3 import PoolManager
from urllib3.exceptions import HTTPError

URL = 'http://httpbin.org/delay/2'

http = PoolManager()

try:
    r = http.request('GET', URL, timeout=0.5, retries=1)
except HTTPError:
    d = 42  # Execute fallback
else:
    d = r.data
print('Received:', d)
```

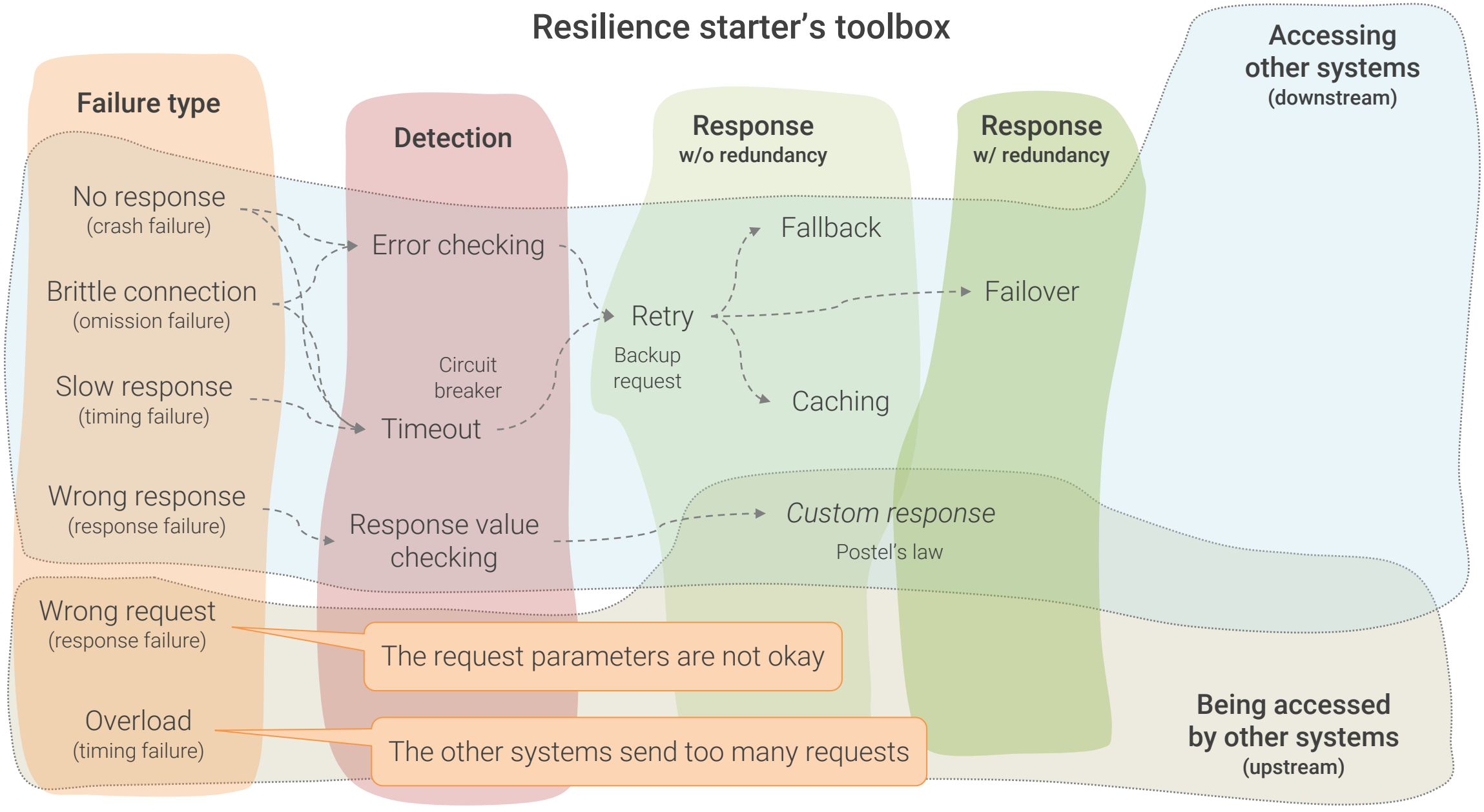


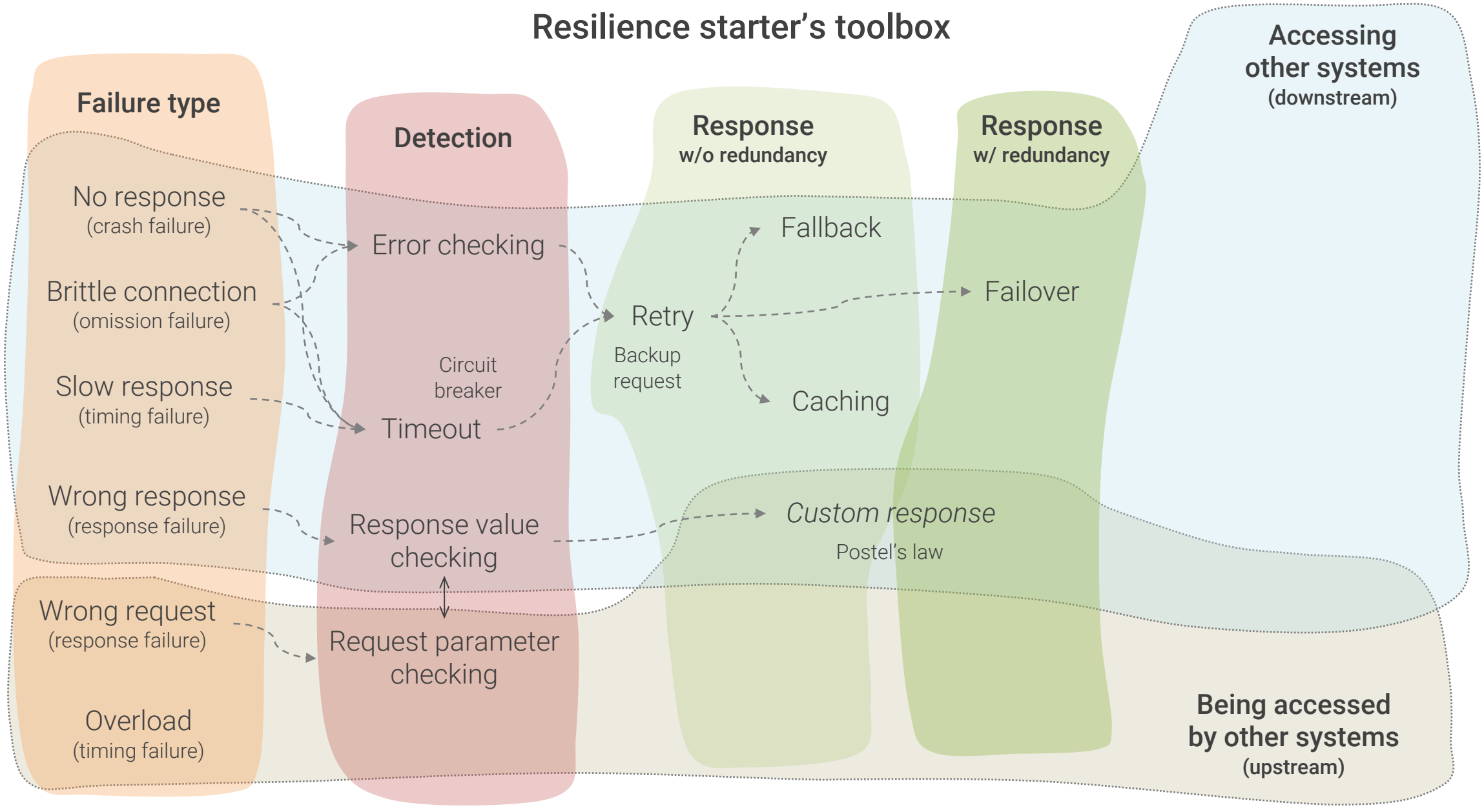
# Being accessed by other systems


Resilience starter's toolbox
Failure type
Detection
Response
w/o redundancy
Response
w/ redundancy
Accessing
other systems
(downstream)
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Error checking
Circuit
breaker
Timeout
Response value
checking
Retry
Backup
request
Fallback
Caching
Failover
Custom response
Postel's law
Being accessed
by other systems
(upstream)

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/square/{number}")
def read_root(number):
    n = int(number)
    return {"result": n*n}
```

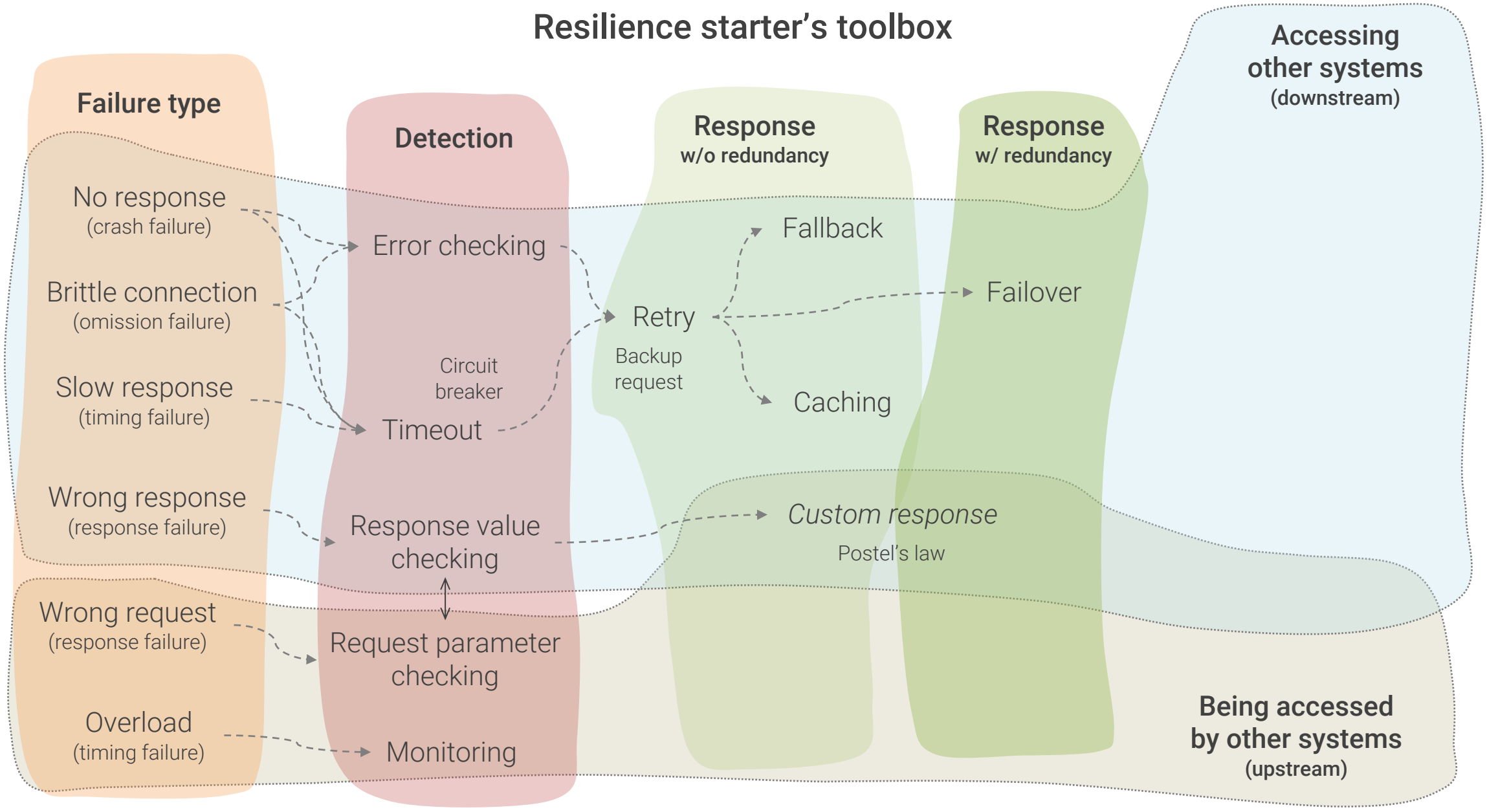
https://fastapi.tiangolo.com/


Resilience starter's toolbox
Failure type
Detection
Response
w/o redundancy
Response
w/ redundancy
Accessing
other systems
(downstream)
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Wrong request
(response failure)
Overload
(timing failure)
Error checking
Circuit
breaker
Timeout
Response value
checking
Retry
Backup
request
Fallback
Failover
Caching
Custom response
Postel's law
The request parameters are not okay
The other systems send too many requests
Being accessed
by other systems
(upstream)


Resilience starter's toolbox
Failure type
Detection
Response
w/o redundancy
Response
w/ redundancy
Accessing
other systems
(downstream)
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Wrong request
(response failure)
Overload
(timing failure)
Error checking
Circuit
breaker
Timeout
Response value
checking
Request parameter
checking
Retry
Backup
request
Fallback
Failover
Caching
Custom response
Postel's law
Being accessed
by other systems
(upstream)

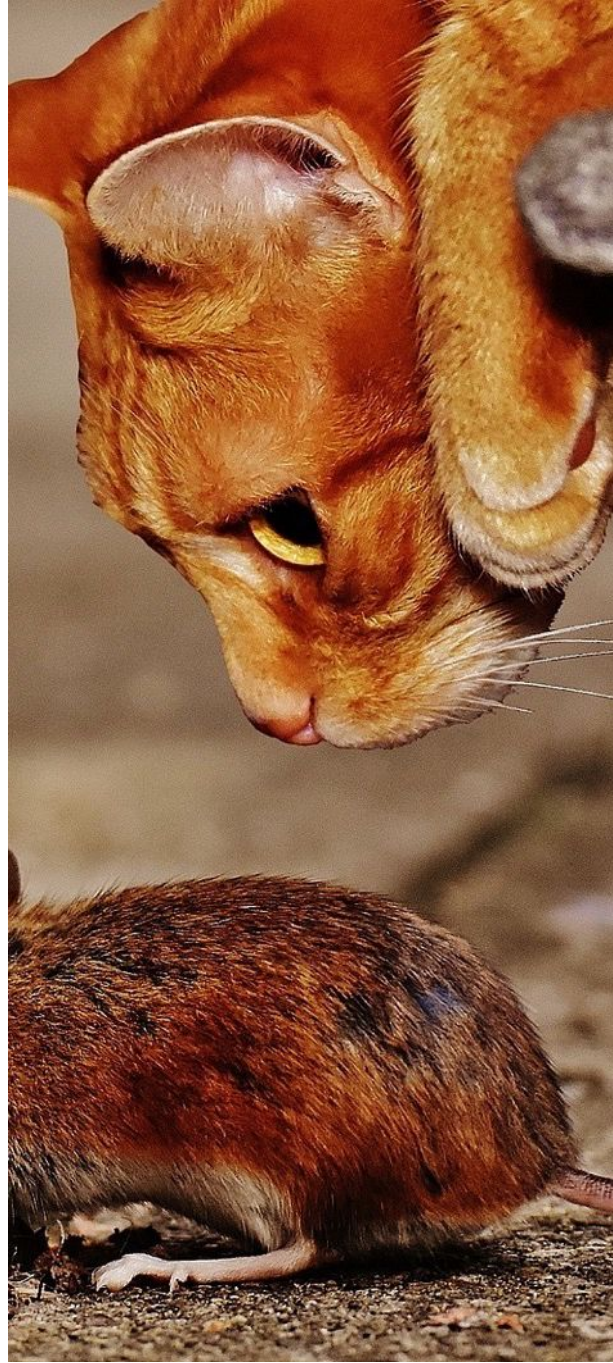
# Request parameter checking

- As obvious as it sounds, yet often neglected
- Protection from broken/malicious request parameters
  - Especially do not forget to check for Null values
- Quite good library support
  - But often do not cover all checks needed
- Consider specific data types


Resilience starter's toolbox
Failure type
Detection
Response
w/o redundancy
Response
w/ redundancy
Accessing
other systems
(downstream)
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Wrong request
(response failure)
Overload
(timing failure)
Error checking
Circuit
breaker
Timeout
Response value
checking
Request parameter
checking
Monitoring
Retry
Backup
request
Fallback
Failover
Caching
Custom response
Postel's law
Being accessed
by other systems
(upstream)

# Monitoring

- Indispensable when running distributed systems
- Good tool support available
- Usually needs application-level support for best performance
  - Application-level and business-level metrics
- Should be combined with self-healing measures
  - Alarms should only be sent if self-healing fails

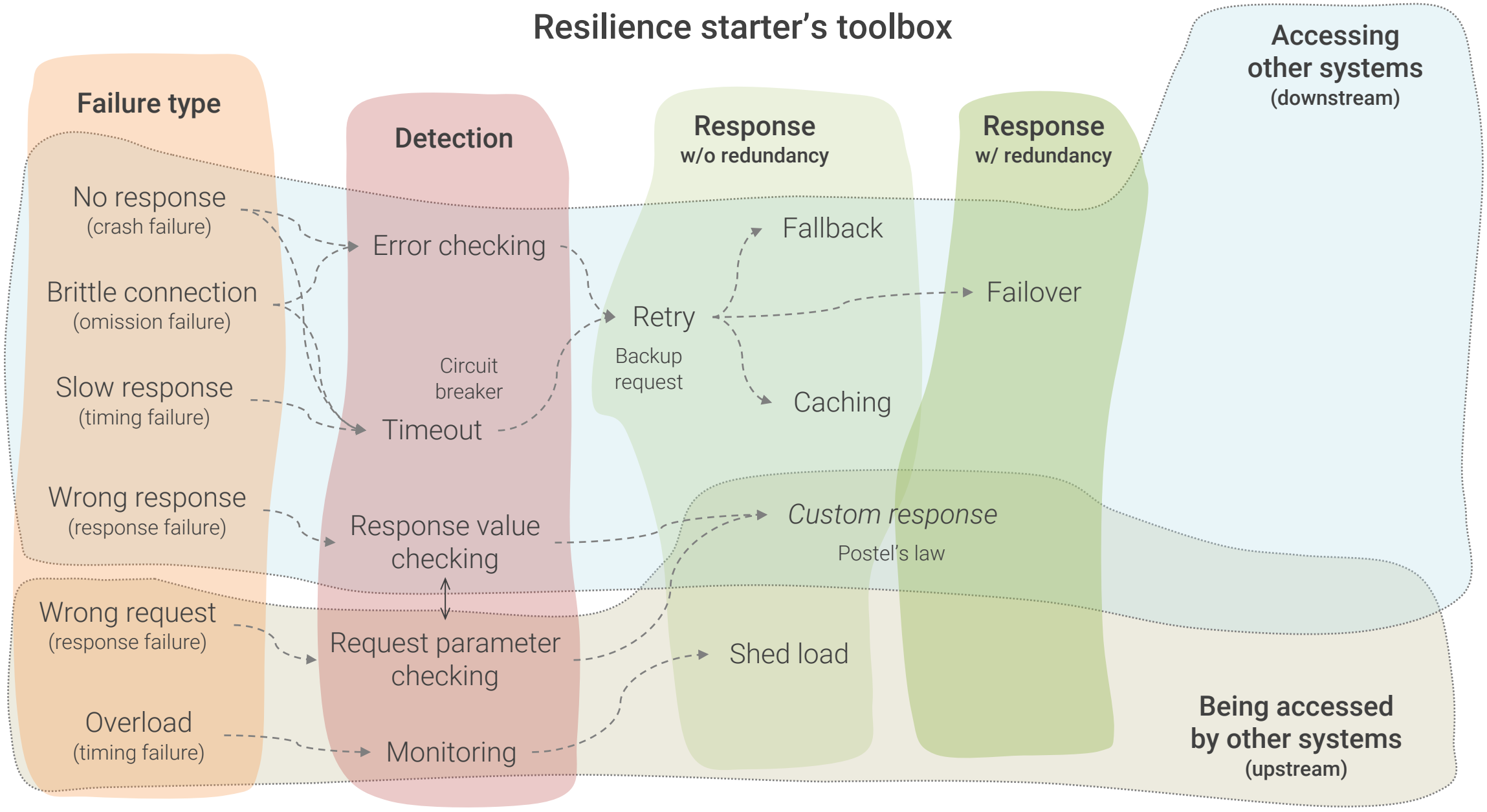



# Adding parameter checking

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/square/{number}")
def read_root(number):
    n = int(number)
    return {"result": n*n}
```

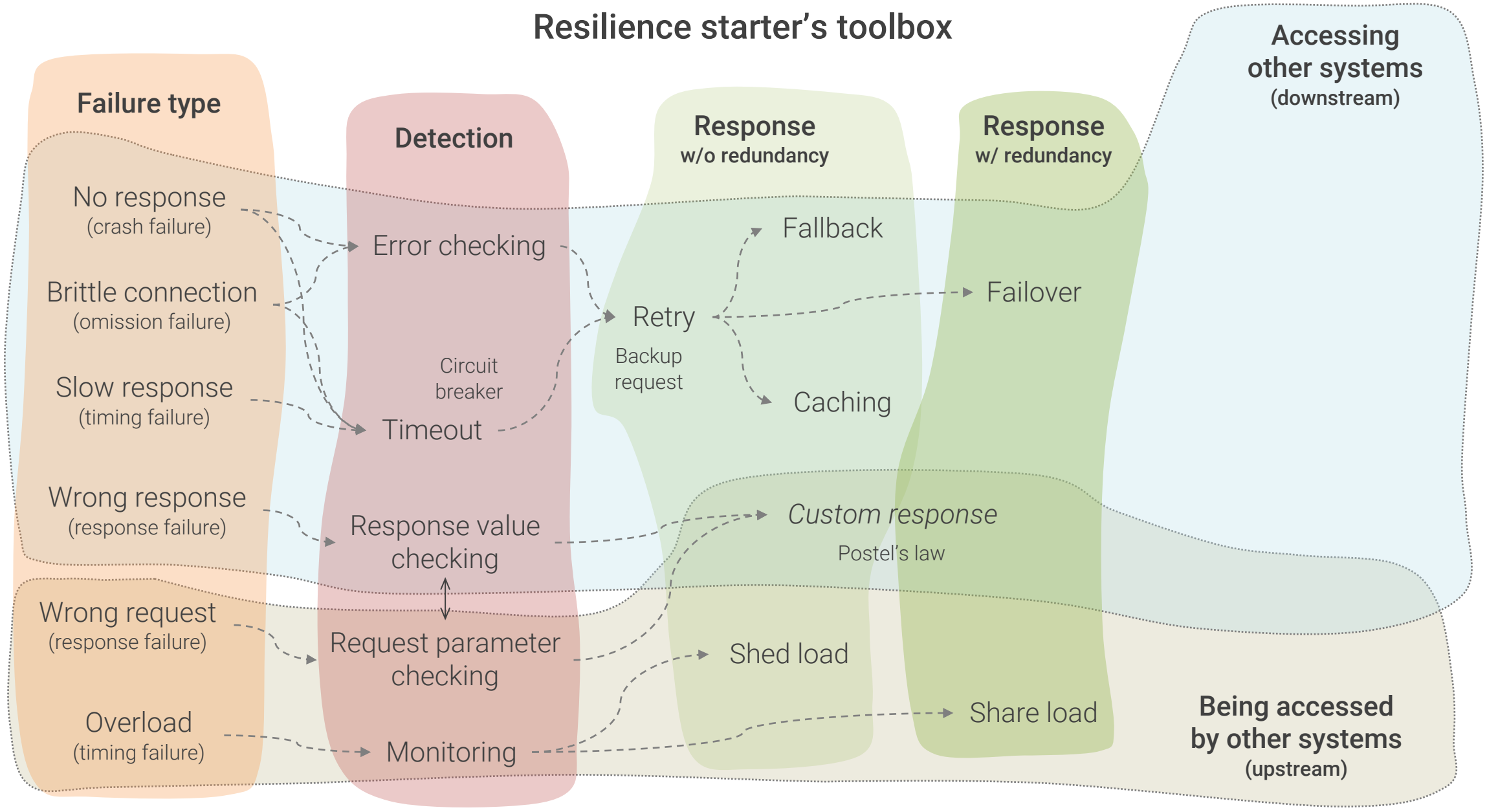
https://fastapi.tiangolo.com/

```
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/square/{number}")
def read_root(number: int = Path(..., gt=0, lt=100)):
    return {"result": number*number}
```


Resilience starter's toolbox
Failure type
Detection
Response
w/o redundancy
Response
w/ redundancy
Accessing
other systems
(downstream)
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Wrong request
(response failure)
Overload
(timing failure)
Error checking
Circuit
breaker
Timeout
Response value
checking
Request parameter
checking
Monitoring
Retry
Backup
request
Fallback
Caching
Failover
Custom response
Postel's law
Shed load
Being accessed
by other systems
(upstream)


# Shed load

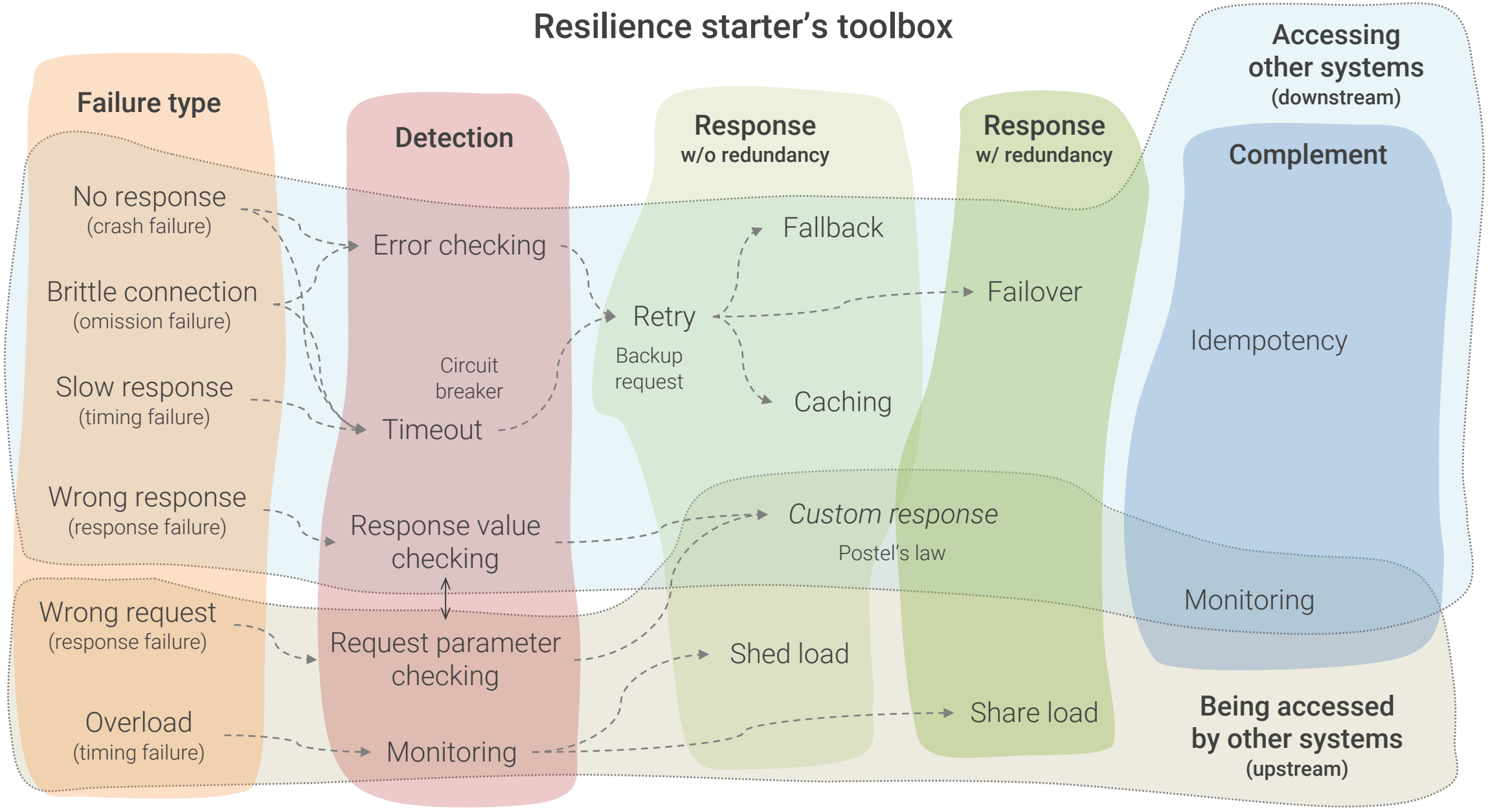
- Limit load to keep throughput of resource acceptable
  - Reject (shed) requests (“rate limiting”)
- Best shed load at periphery
  - Minimize impact on resource itself
  - Good tool support available
- Usually requires monitoring data to watch load of resource
- Try not to break ongoing multi-request sessions


Resilience starter's toolbox
Failure type
Detection
Response
w/o redundancy
Response
w/ redundancy
Accessing
other systems
(downstream)
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Wrong request
(response failure)
Overload
(timing failure)
Error checking
Circuit
breaker
Timeout
Response value
checking
Request parameter
checking
Monitoring
Retry
Backup
request
Fallback
Caching
Failover
Custom response
Postel's law
Shed load
Share load
Being accessed
by other systems
(upstream)

# Share load

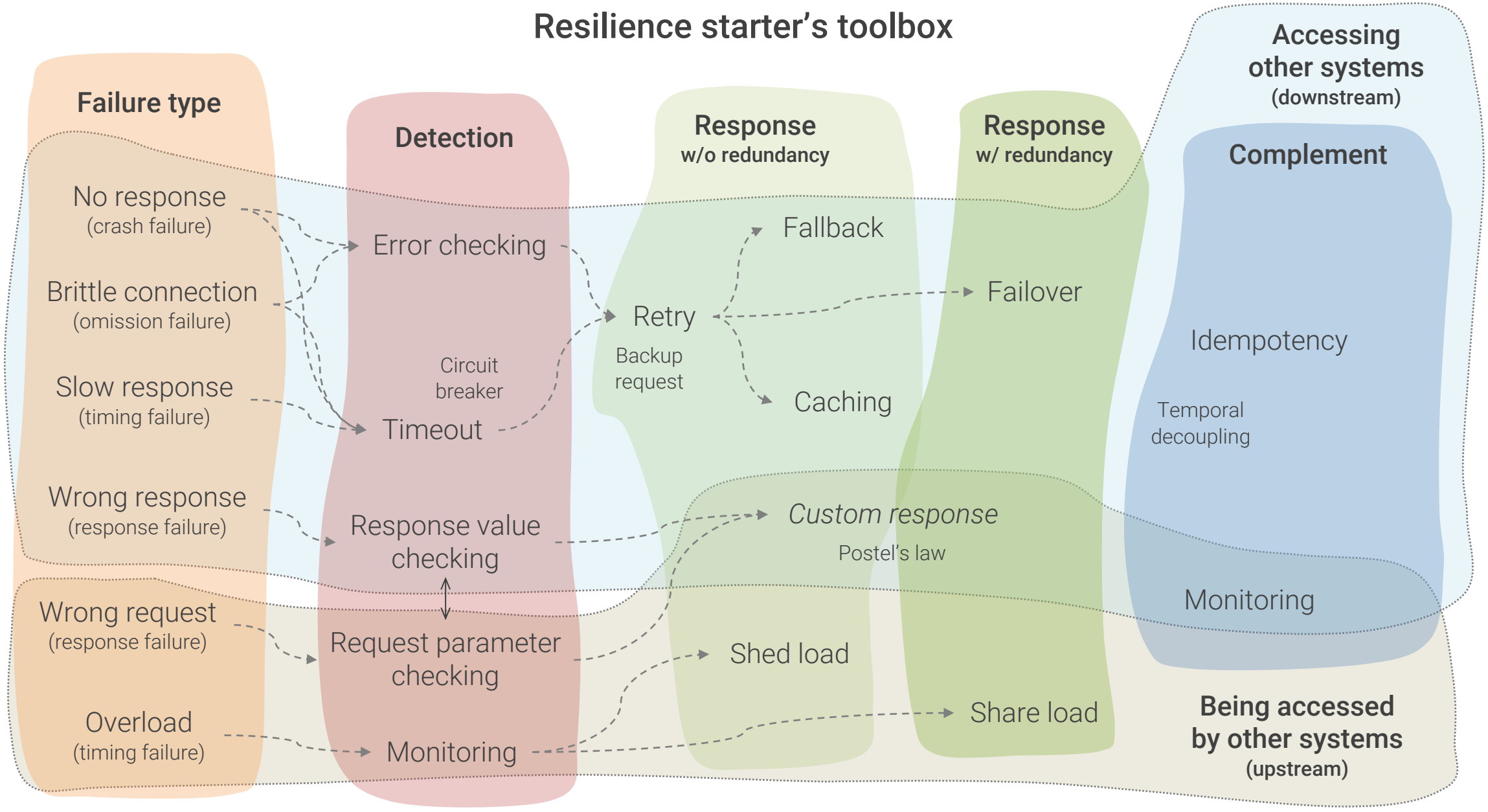
- Share load between resources to keep throughput good
- Use if additional resources for load sharing can be used
- Can be implemented statically or dynamically (“auto-scaling”)
- Very good tool support available
- Minimize synchronization needed between resources
    - Synchronization needs kill scalability

# Useful complementing patterns


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Wrong request
(response failure)
Overload
(timing failure)
Detection
Error checking
Circuit
breaker
Timeout
Response value
checking
Request parameter
checking
Monitoring
Response
w/o redundancy
Retry
Backup
request
Fallback
Caching
Custom response
Postel's law
Shed load
Response
w/ redundancy
Failover
Share load
Accessing
other systems
(downstream)
Complement
Idempotency
Monitoring
Being accessed
by other systems
(upstream)

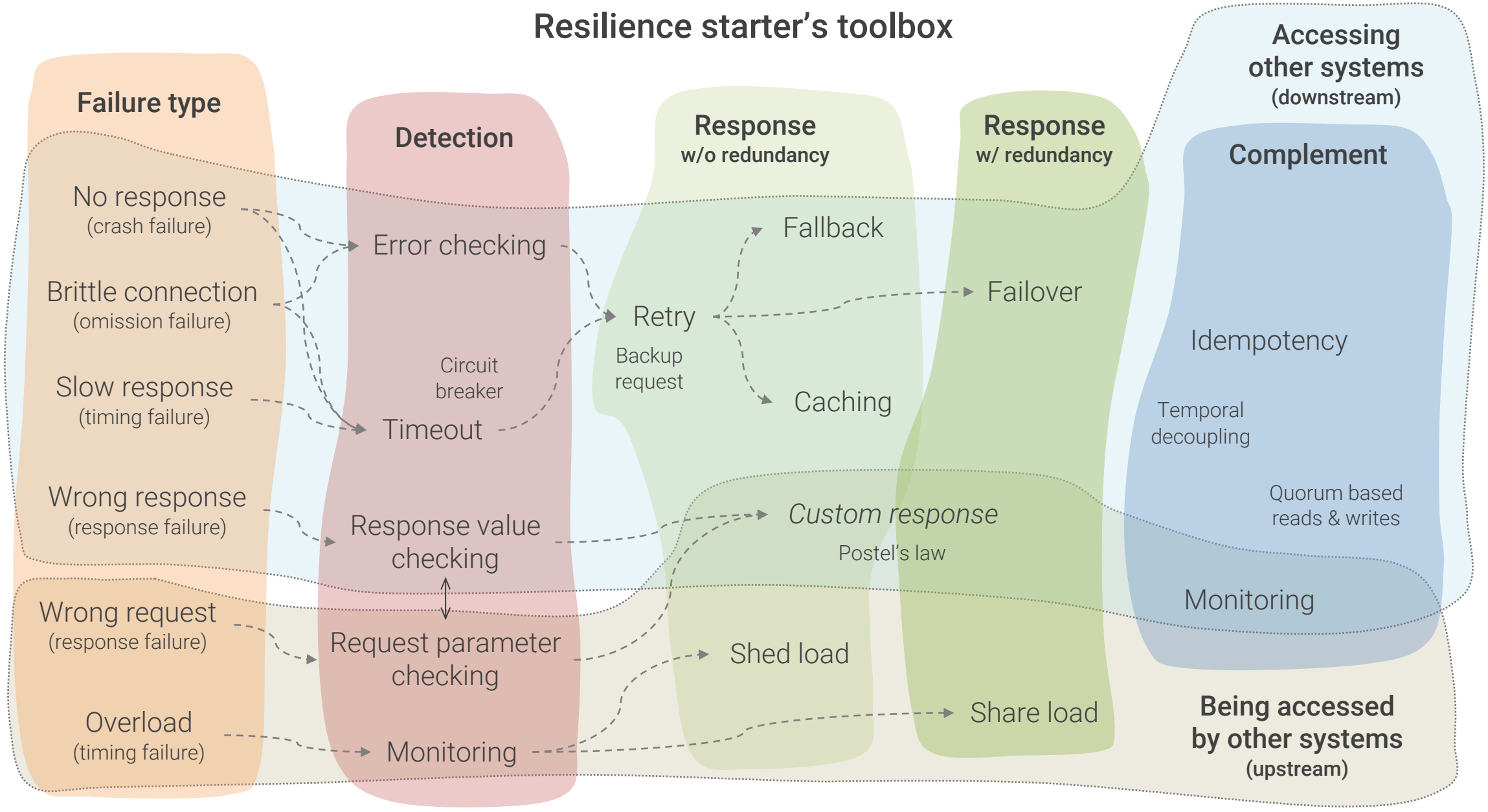
# Idempotency

- Non-idempotent calls become very complicated if they fail
- Idempotent calls can be repeated without problems
  - Always return the same result
  - Do not trigger any cumulating side-effects
- Reduces coupling between nodes
  - Simplifies responding to most failure types a lot
- Very fundamental resilience and scalability pattern


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Wrong request
(response failure)
Overload
(timing failure)
Detection
Error checking
Circuit
breaker
Timeout
Response value
checking
Request parameter
checking
Monitoring
Response
w/o redundancy
Retry
Backup
request
Fallback
Caching
Custom response
Postel's law
Shed load
Response
w/ redundancy
Failover
Share load
Accessing
other systems
(downstream)
Complement
Idempotency
Temporal
decoupling
Monitoring
Being accessed
by other systems
(upstream)

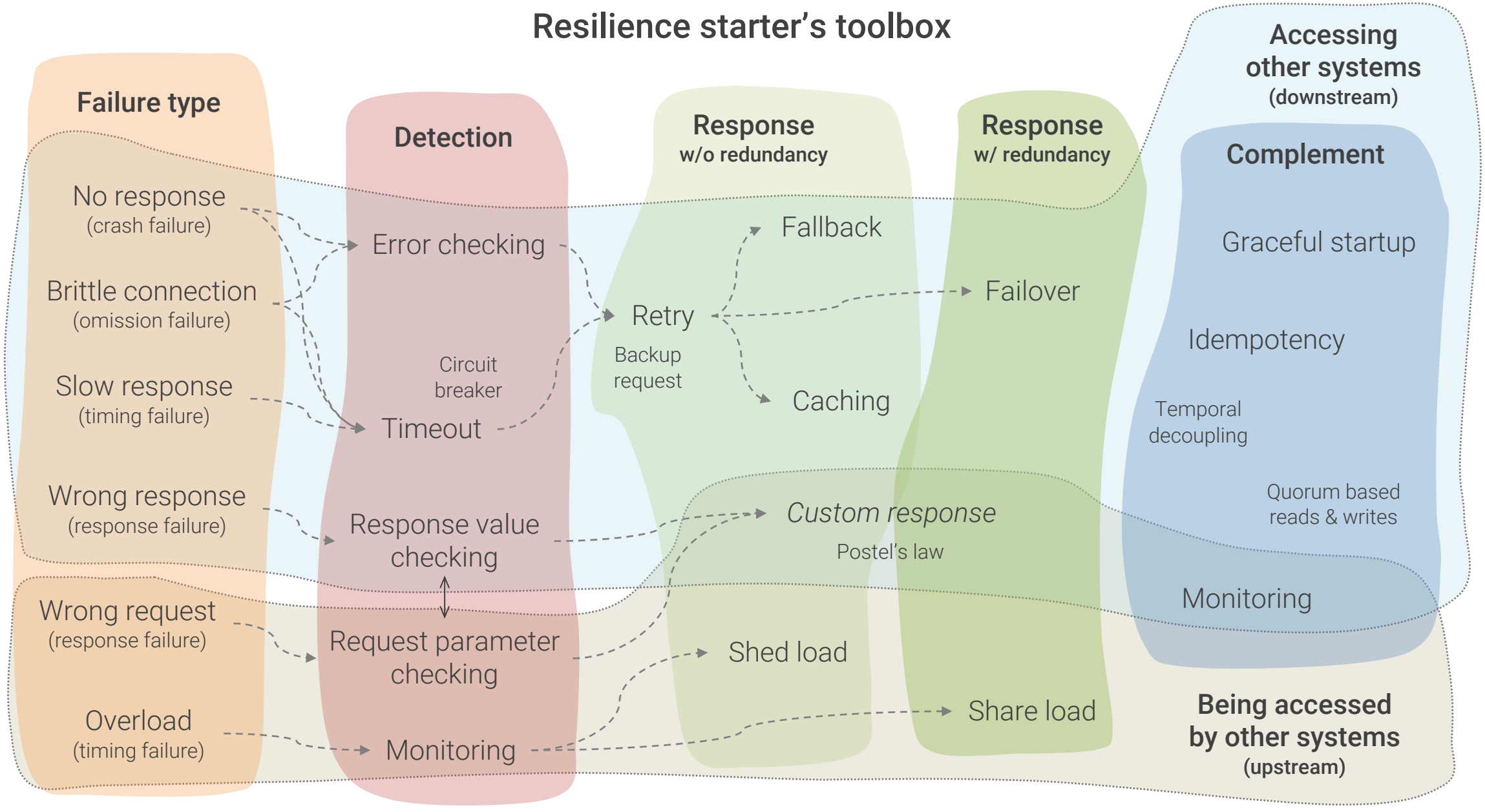
# Temporal decoupling

- Request, processing and response are temporally decoupled
- Simplifies responding to timing failures a lot
  - Not necessary to recover from failures within caller's response time expectations
- Functional design issue
  - Technology only augments it
- Enables simpler and more robust communication types
  - E.g., batch processing


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Wrong request
(response failure)
Overload
(timing failure)
Detection
Error checking
Circuit
breaker
Timeout
Response value
checking
Request parameter
checking
Monitoring
Response
w/o redundancy
Retry
Backup
request
Fallback
Caching
Custom response
Postel's law
Shed load
Response
w/ redundancy
Failover
Share load
Accessing
other systems
(downstream)
Complement
Idempotency
Temporal
decoupling
Quorum based
reads & writes
Monitoring
Being accessed
by other systems
(upstream)

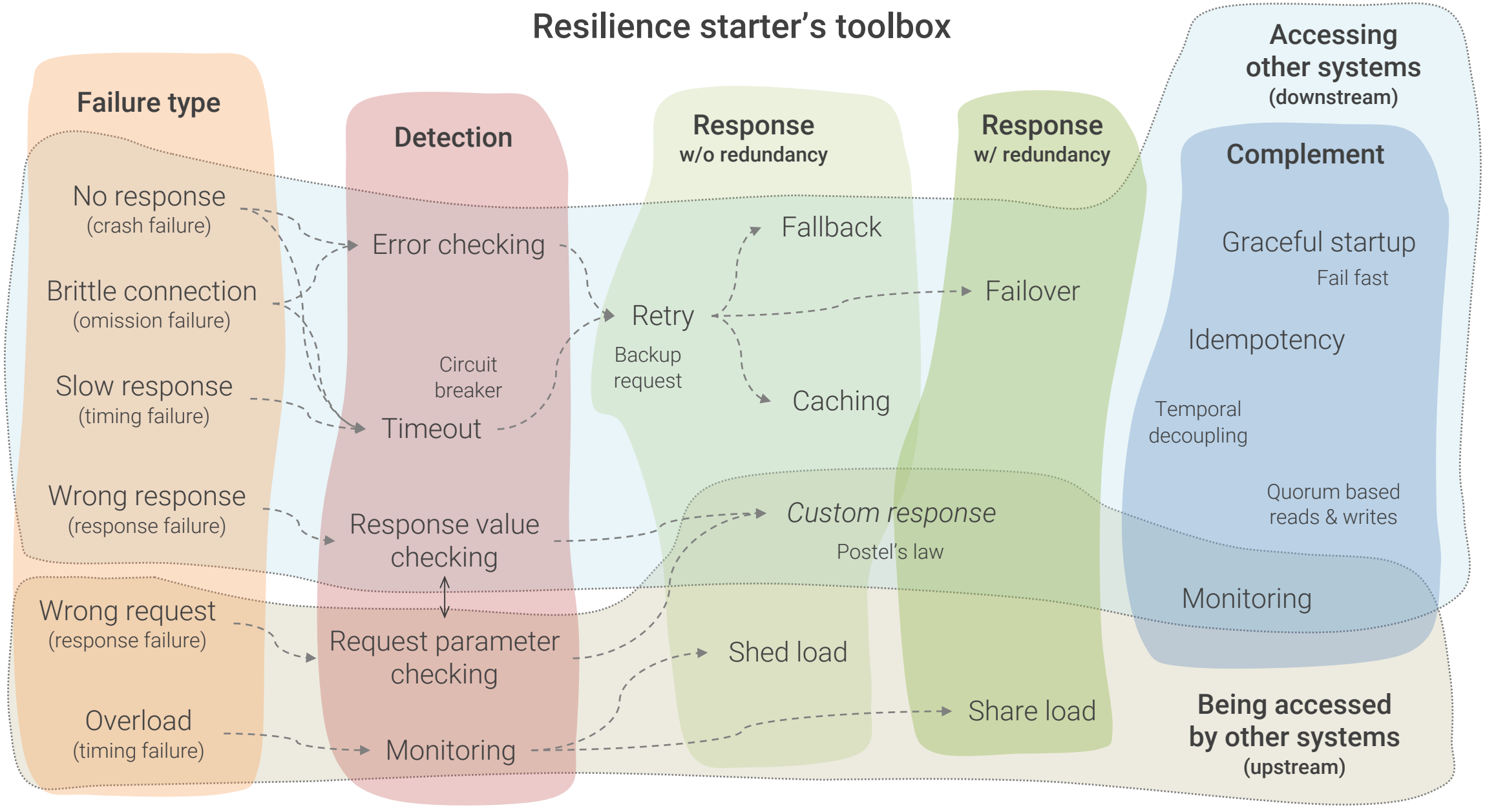
# Quorum-based reads and writes

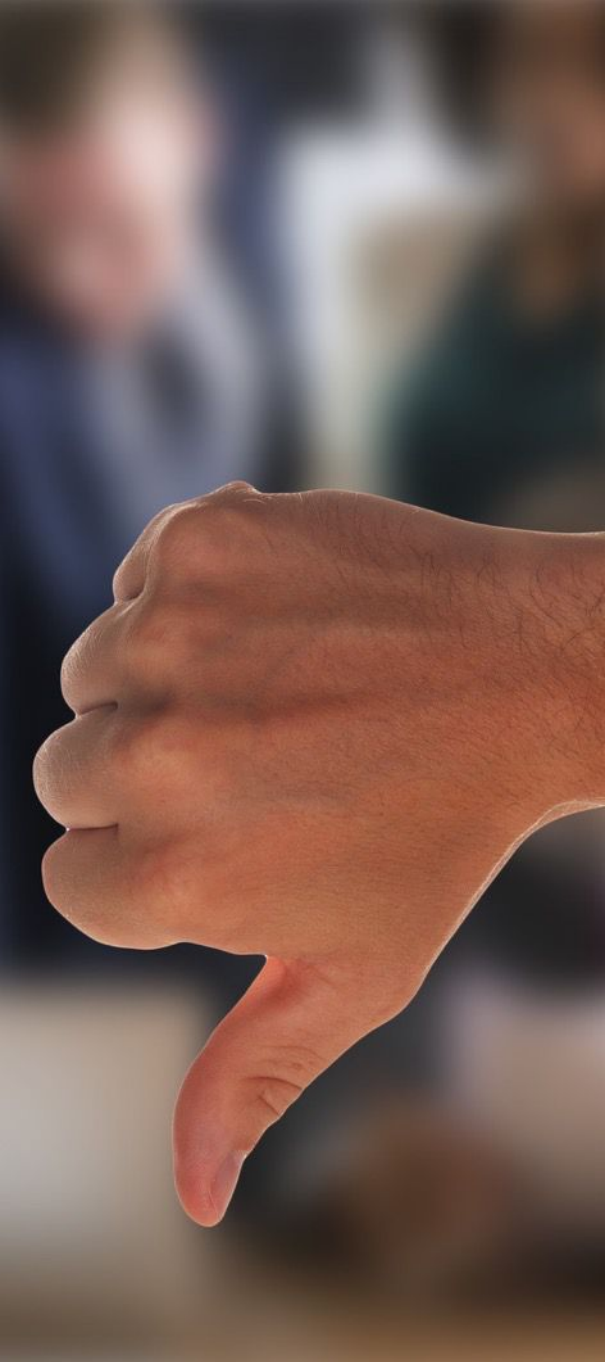
- Became popular with the rise of NoSQL databases
- Useful pattern for distributed, replicated data stores
  - Relaxes consistency constraints while writing
  - Detects inconsistencies due to a (temporally) failed prior write
- Not a replacement for response value checking
- Not to be confused with ACID transactions


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Wrong request
(response failure)
Overload
(timing failure)
Detection
Error checking
Circuit
breaker
Timeout
Response value
checking
Request parameter
checking
Monitoring
Response
w/o redundancy
Retry
Backup
request
Fallback
Caching
Custom response
Postel's law
Shed load
Response
w/ redundancy
Failover
Share load
Accessing
other systems
(downstream)
Complement
Graceful startup
Idempotency
Temporal
decoupling
Quorum based
reads & writes
Monitoring
Being accessed
by other systems
(upstream)

# Graceful startup

- Implement graceful startup mode
  - Wait until all required resources and services are available before switching to runtime mode
- Makes application startup order interchangeable
- Crucial for quick recovery after bigger failures
- Simple and powerful, but often neglected pattern

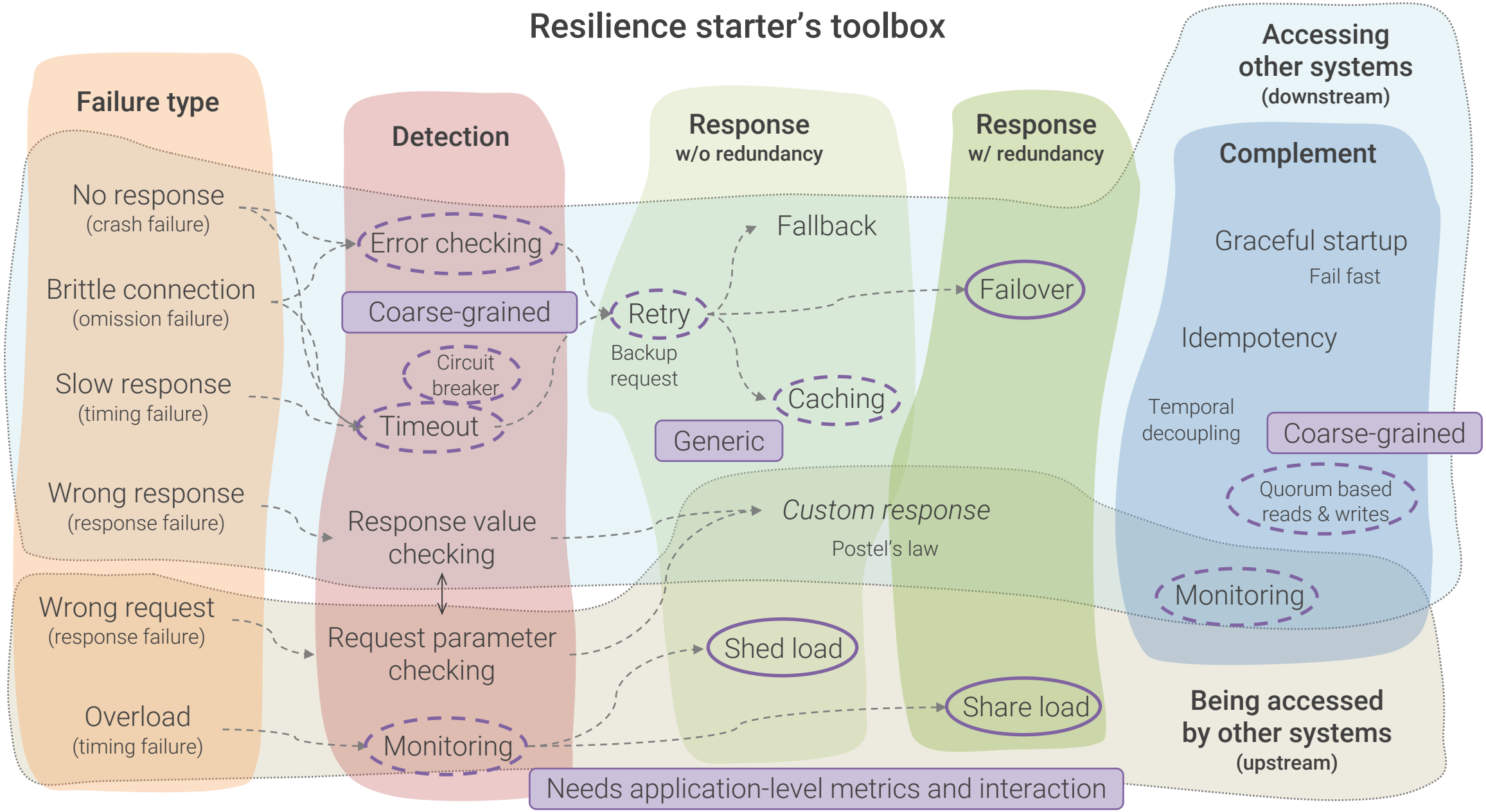

Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Wrong request
(response failure)
Overload
(timing failure)
Detection
Error checking
Circuit
breaker
Timeout
Response value
checking
Request parameter
checking
Monitoring
Response
w/o redundancy
Retry
Backup
request
Fallback
Caching
Custom response
Postel's law
Shed load
Response
w/ redundancy
Failover
Share load
Accessing
other systems
(downstream)
Complement
Graceful startup
Fail fast
Idempotency
Temporal
decoupling
Quorum based
reads & writes
Monitoring
Being accessed
by other systems
(upstream)



# Fail fast

- “If you know you’re going to fail, you better fail fast”
- Usually implemented in front of costly actions
- Saves time and resources by avoiding foreseeable failures
- Useful in normal operations mode
- Can be counterproductive in startup mode


What can we delegate to the infrastructure level?


Resilience starter's toolbox
Failure type
No response
(crash failure)
Brittle connection
(omission failure)
Slow response
(timing failure)
Wrong response
(response failure)
Wrong request
(response failure)
Overload
(timing failure)
Detection
Error checking
Coarse-grained
Circuit breaker
Timeout
Response value checking
Request parameter checking
Monitoring
Response
w/o redundancy
Fallback
Retry
Backup request
Caching
Generic
Custom response
Postel's law
Shed load
Response
w/ redundancy
Failover
Share load
Accessing other systems
(downstream)
Complement
Graceful startup
Fail fast
Idempotency
Temporal decoupling
Coarse-grained
Quorum based reads & writes
Monitoring
Being accessed by other systems
(upstream)
Needs application-level metrics and interaction


But that is still a lot to implement


But what should be the alternative?

Should we let the application crash
whenever something goes wrong?

Always keep in mind …

The question is no longer, **if** failures will hit you

The only question left is, **when** and **how bad** they will hit you

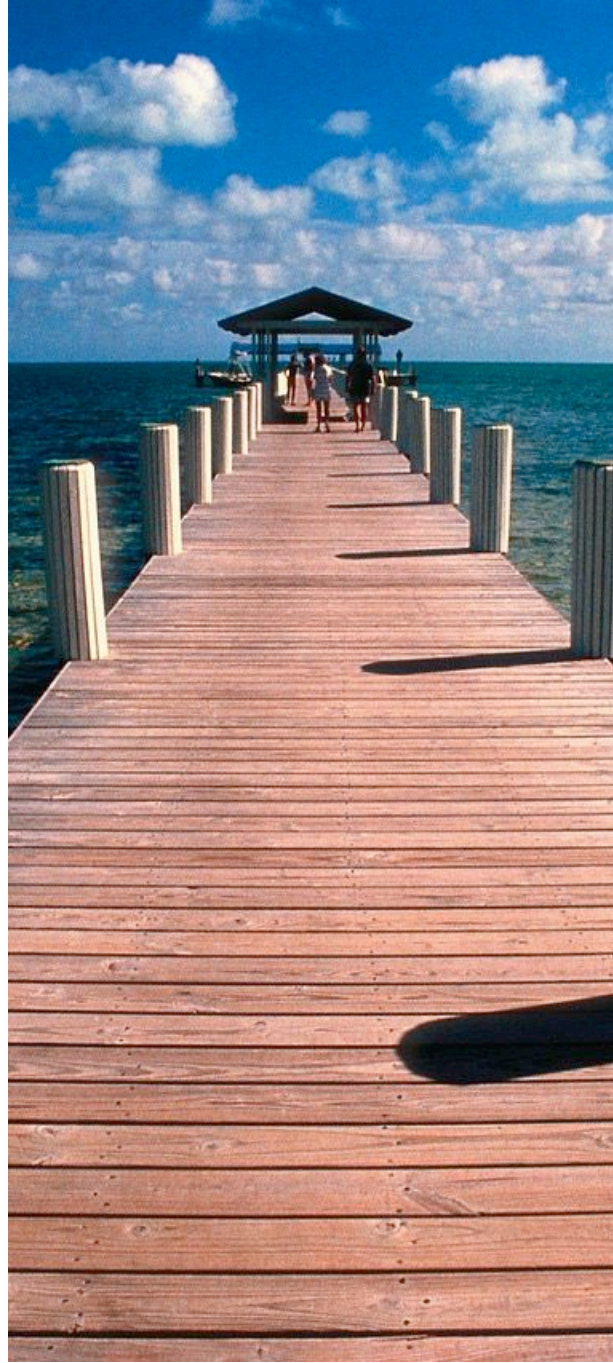
Thus, look for library and framework support … but do the work!


Everyone loves resilient applications

# Wrap-up

# Wrap-up

- Resilience is a huge topic
- Distribution makes resilient software design mandatory
- It will hit you at the application level
- The starter's toolbox
- Delegate to the infrastructure what is possible
  - ... but consider the limitations
- Look for library and framework support



# Recommended readings

**Release It! Design and Deploy Production-Ready Software**,
Michael Nygard, 2nd edition, Pragmatic Bookshelf, 2018

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

**Distributed Systems – Principles and Paradigms**,
Andrew Tanenbaum, Marten van Steen, 3rd Edition, 2017,
https://www.distributed-systems.net/index.php/books/ds3/

**On Designing and Deploying Internet-Scale Services**,
James Hamilton, 21st LISA Conference 2007

**Site Reliability Engineering**,
Betsy Beyer et al., O'Reilly, 2016


Questions & discussion

# Uwe Friedrichsen

Works @ codecentric

https://twitter.com/ufried

https://www.speakerdeck.com/ufried

https://ufried.com/