



10 patterns for more resilient applications

A gentle start into resilient software design

Uwe Friedrichsen (codecentric AG) – JUG Saxony Day – Radebeul, 23. September 2022

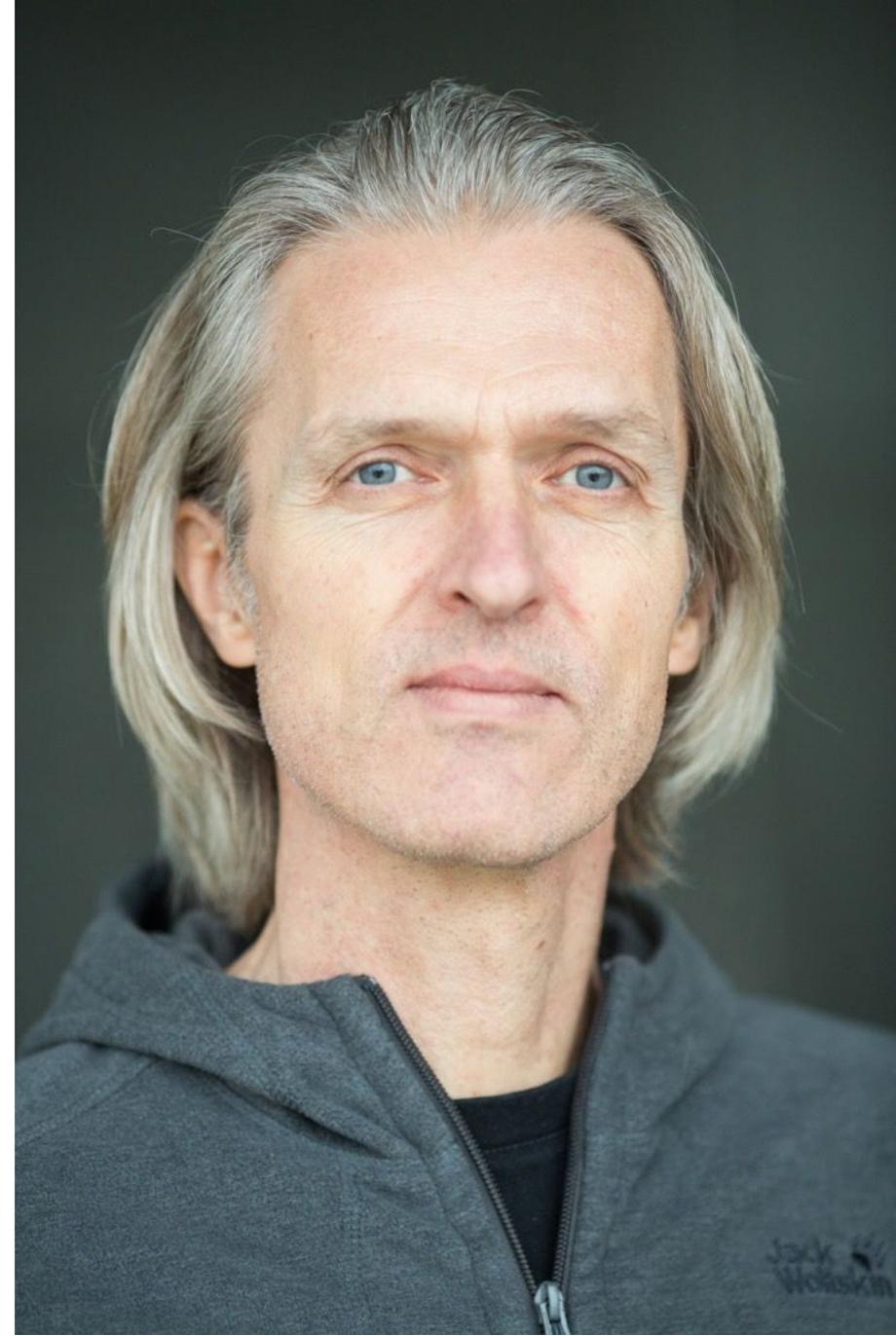
Uwe Friedrichsen

Works @ codecentric

<https://twitter.com/ufried>

<https://www.speakerdeck.com/ufried>

<https://ufried.com/>



A group of meerkats is shown on a rock. One meerkat stands in the background, looking forward. Another meerkat is in the middle ground, looking to the right. In the foreground, several meerkats are sitting on the rock, looking in various directions. The background is a blurred green, suggesting a natural habitat. A semi-transparent white box is overlaid on the image, containing the text.

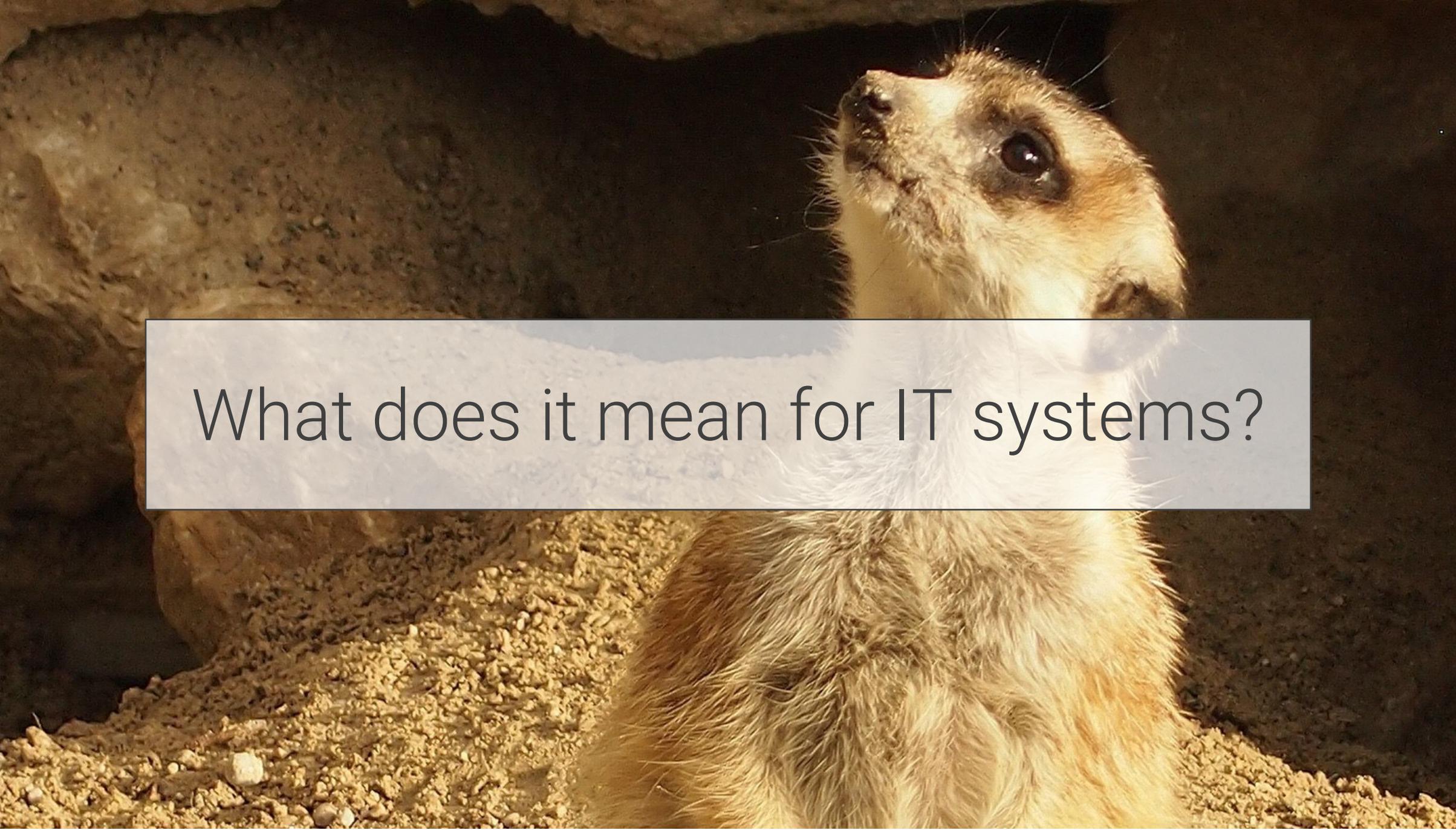
What is that “resilience” thing?

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.

American Heritage® Dictionary of the English Language, Fifth Edition. Copyright © 2016 by Houghton Mifflin Harcourt Publishing Company. Published by Houghton Mifflin Harcourt Publishing Company. All rights reserved.

A close-up photograph of a meerkat's head and shoulders. The meerkat is looking upwards and to the left, with its mouth slightly open. It has light brown fur with darker stripes on its back. The background is dark and appears to be the interior of a burrow, with some soil visible at the bottom.

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 and quick recovery to normal operations (non-ideal case)

The cautious attempt to provide a useful definition for resilience in the context of software systems.
No copyright attached, but also no guarantee that this definition is sufficient for all relevant purposes.



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

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

(Almost) every system is a distributed system.

-- Chas Emerick

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

Properties of 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 know how to detect if it hit you and how to respond



A group of meerkats is perched on a large, light-colored rock. In the foreground, a meerkat is looking towards the camera. Behind it, another meerkat is looking to the right. A third meerkat is visible in the background, also looking right. A semi-transparent white box with a thin black border is centered over the image, containing the text "Okay, I buy it. But how do I start?".

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

Let us start simple ... *

* which often improves the situation amazingly much

A top-down view of a collection of hand tools laid out on a dark, textured wooden surface. The tools include a hammer with a red handle, several pairs of pliers with yellow and black handles, multiple screwdrivers with different handle colors (red, yellow, black), a large adjustable wrench, a smaller open-end wrench, and a hand saw with a yellow handle. A semi-transparent white rectangular box is centered over the tools, containing the text "Let us create our starter's toolbox".

Let us create our starter's toolbox

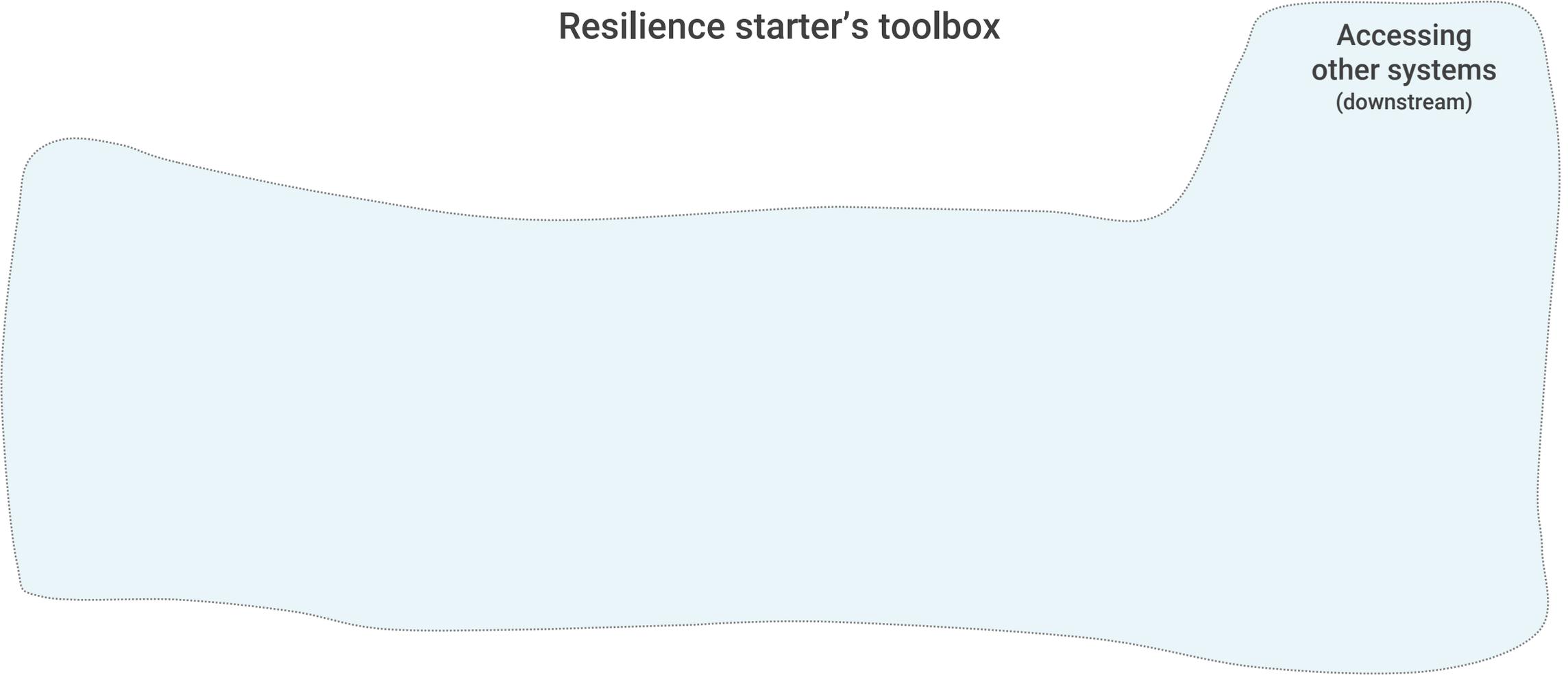
Resilience starter's toolbox

A photograph of two golden snub-nosed monkeys sitting on a tree branch. The monkey on the left is looking towards the right, while the monkey on the right is looking down. A semi-transparent white box with a black border is overlaid on the center of the image, containing the text "Accessing other systems".

Accessing other systems

Resilience starter's toolbox

**Accessing
other systems
(downstream)**



```
from urllib3 import PoolManager
```

```
URL = <...>
```

```
http = PoolManager()
```

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

Resilience starter's toolbox

Failure type

No response
(crash failure)

The other system does not respond at all

Brittle connection
(omission failure)

The other system does not respond reliably

Slow response
(timing failure)

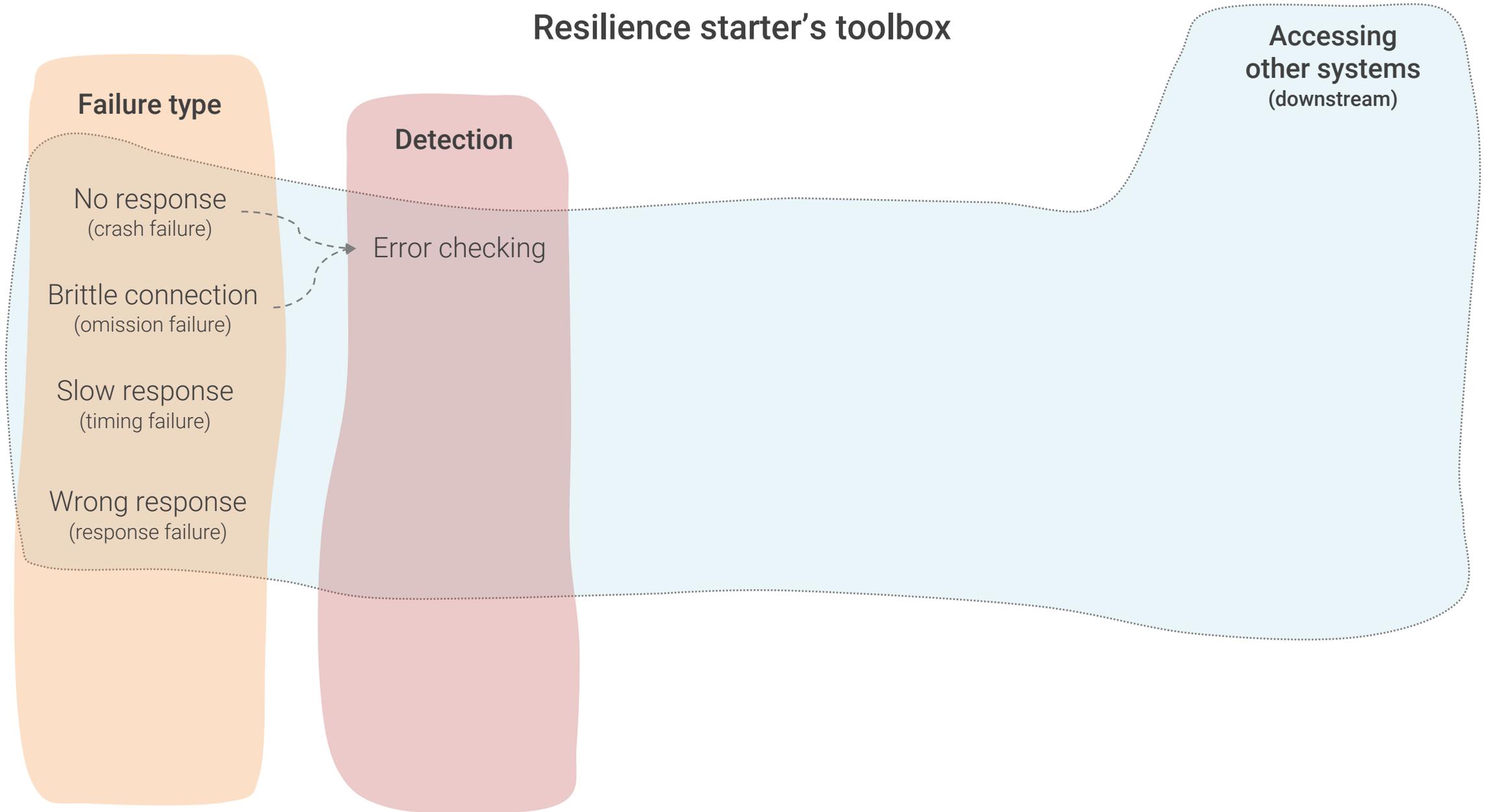
It takes too long until the other system responds

Wrong response
(response failure)

The other system responds, but the response is not okay

Accessing
other systems
(downstream)

Resilience starter's toolbox

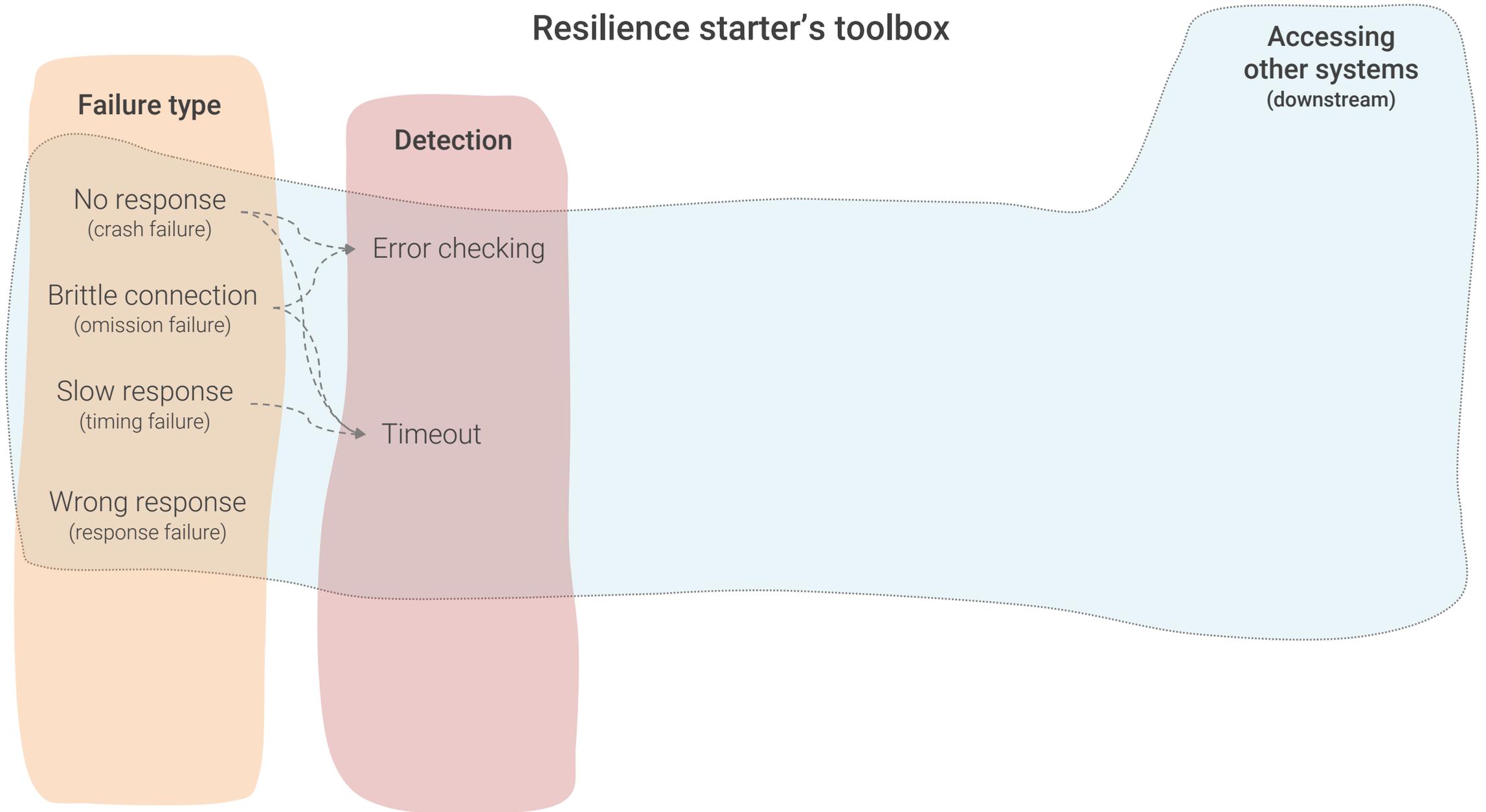




Error checking

- 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

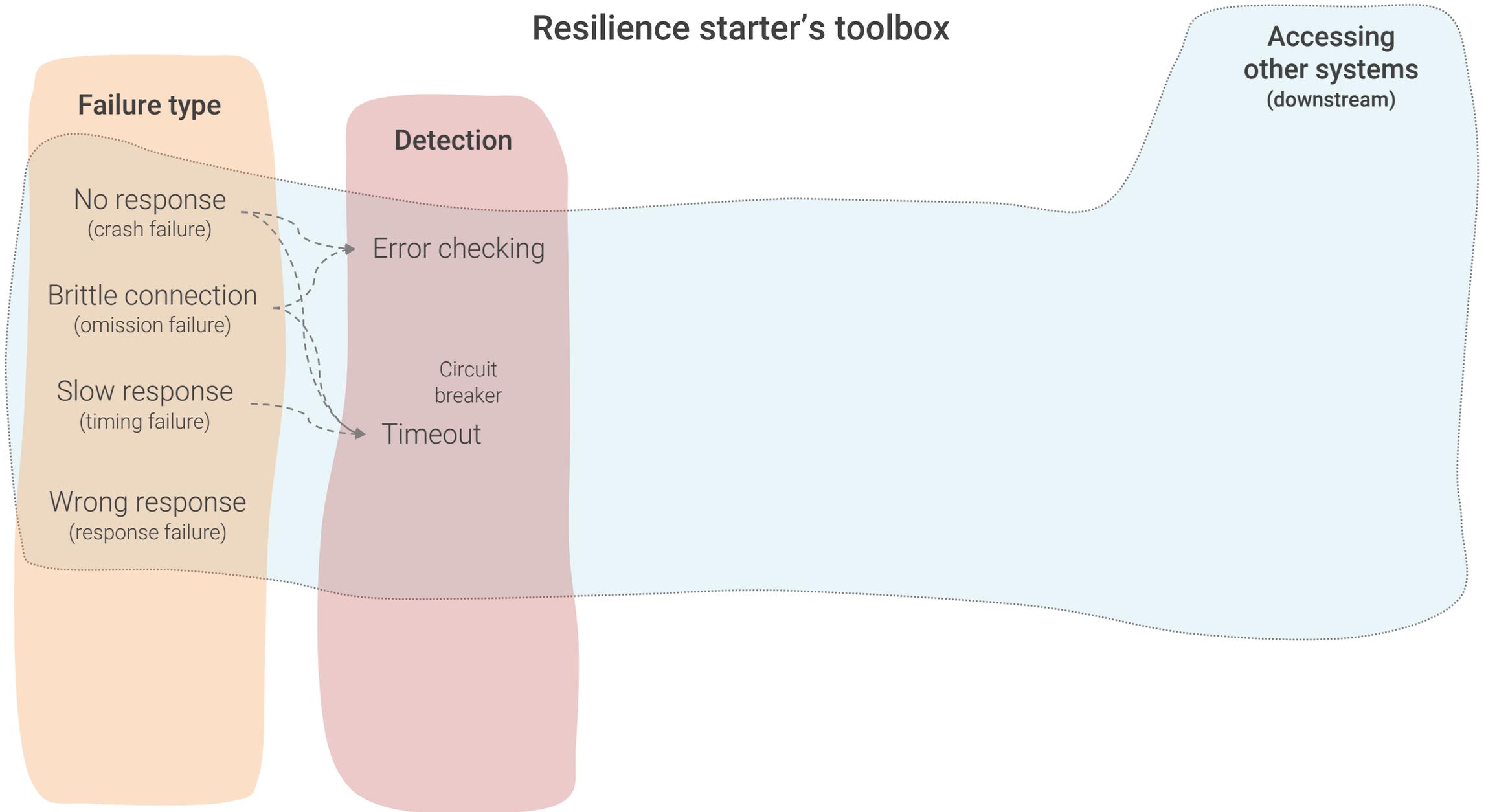
Accessing
other systems
(downstream)

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


```
from urllib3 import PoolManager
```

```
URL = <...>
```

```
http = PoolManager()
```

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

```
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)
```

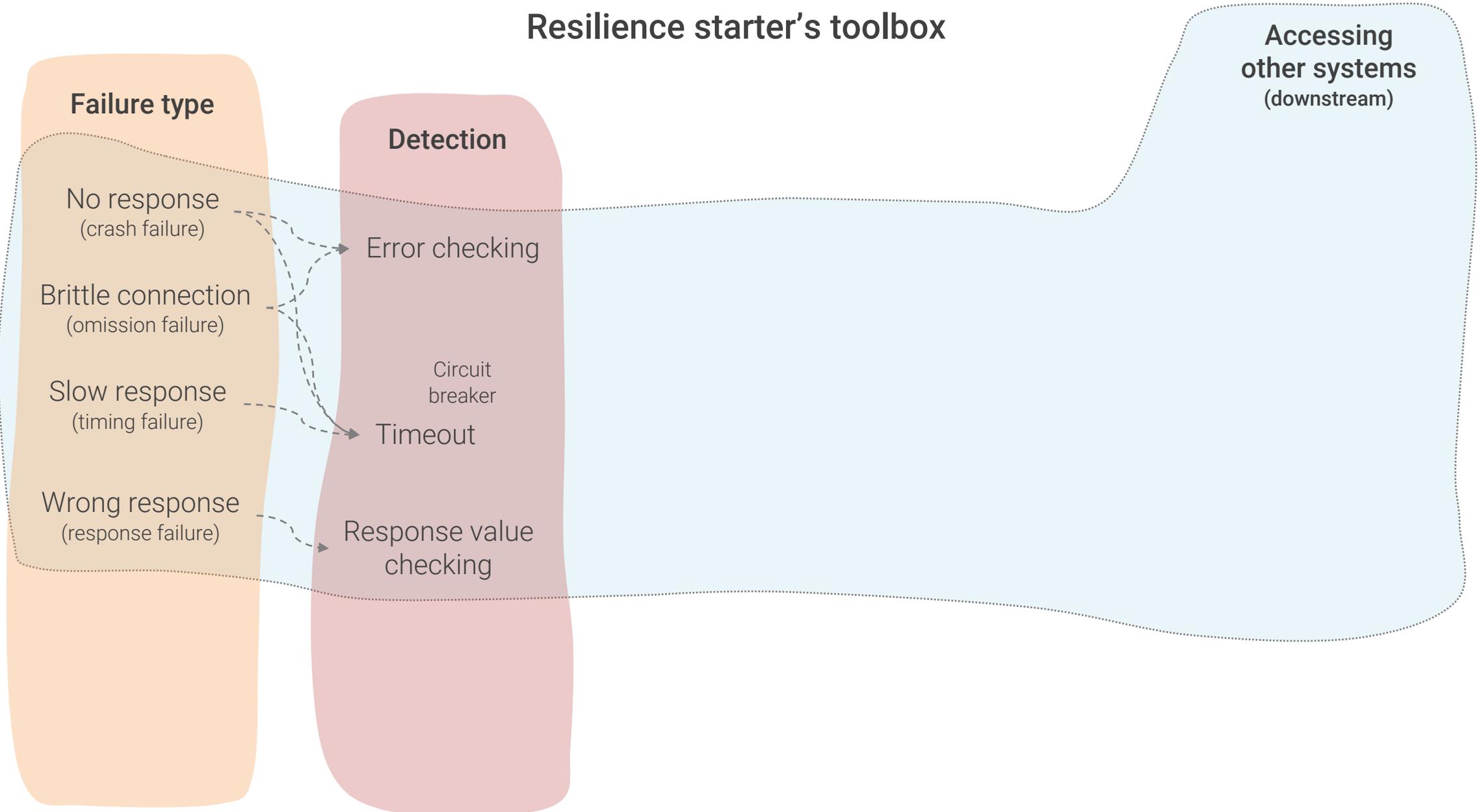
```
from urllib3 import PoolManager
from urllib3.exceptions import HTTPError

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

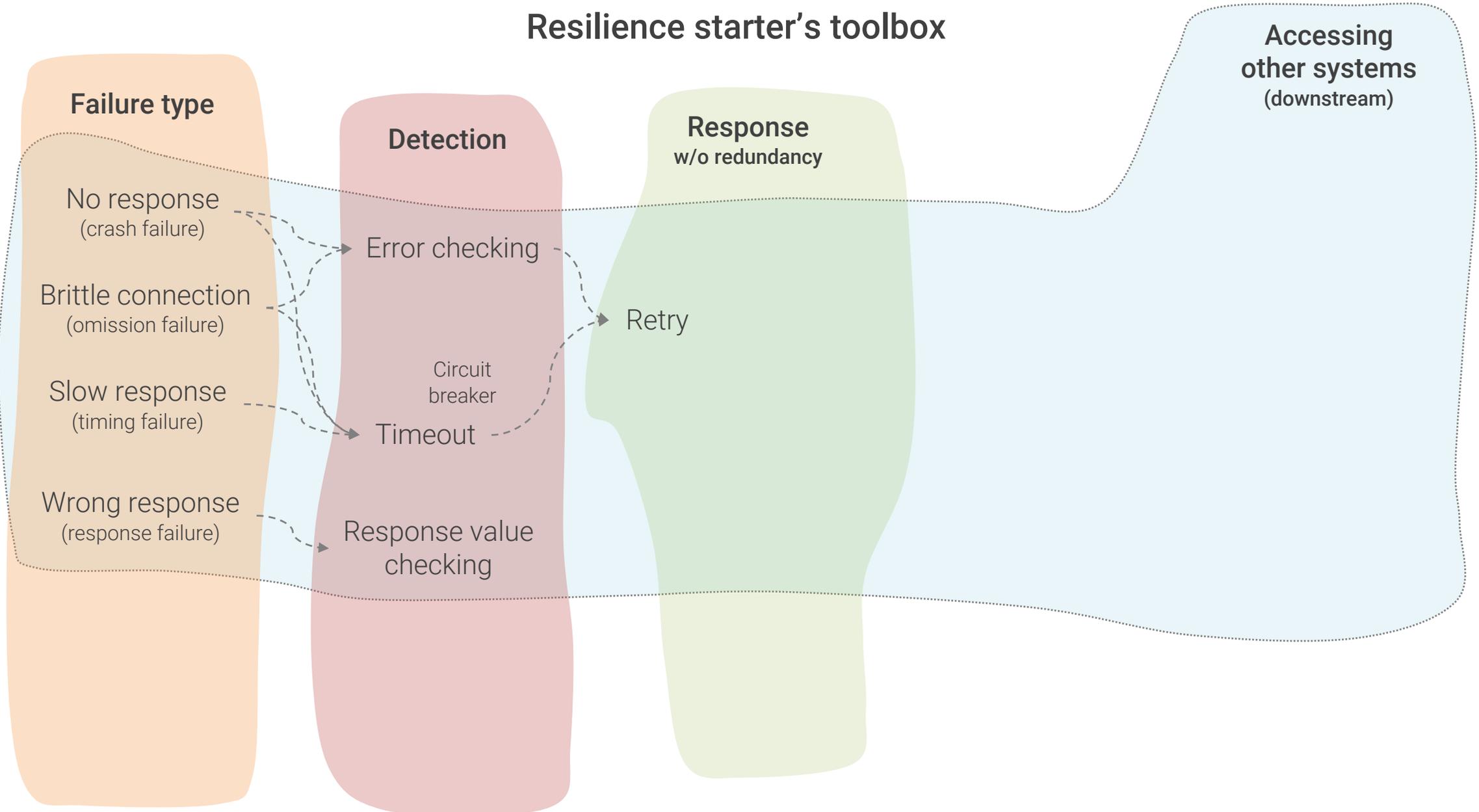


Response value checking

- 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



Resilience starter's toolbox

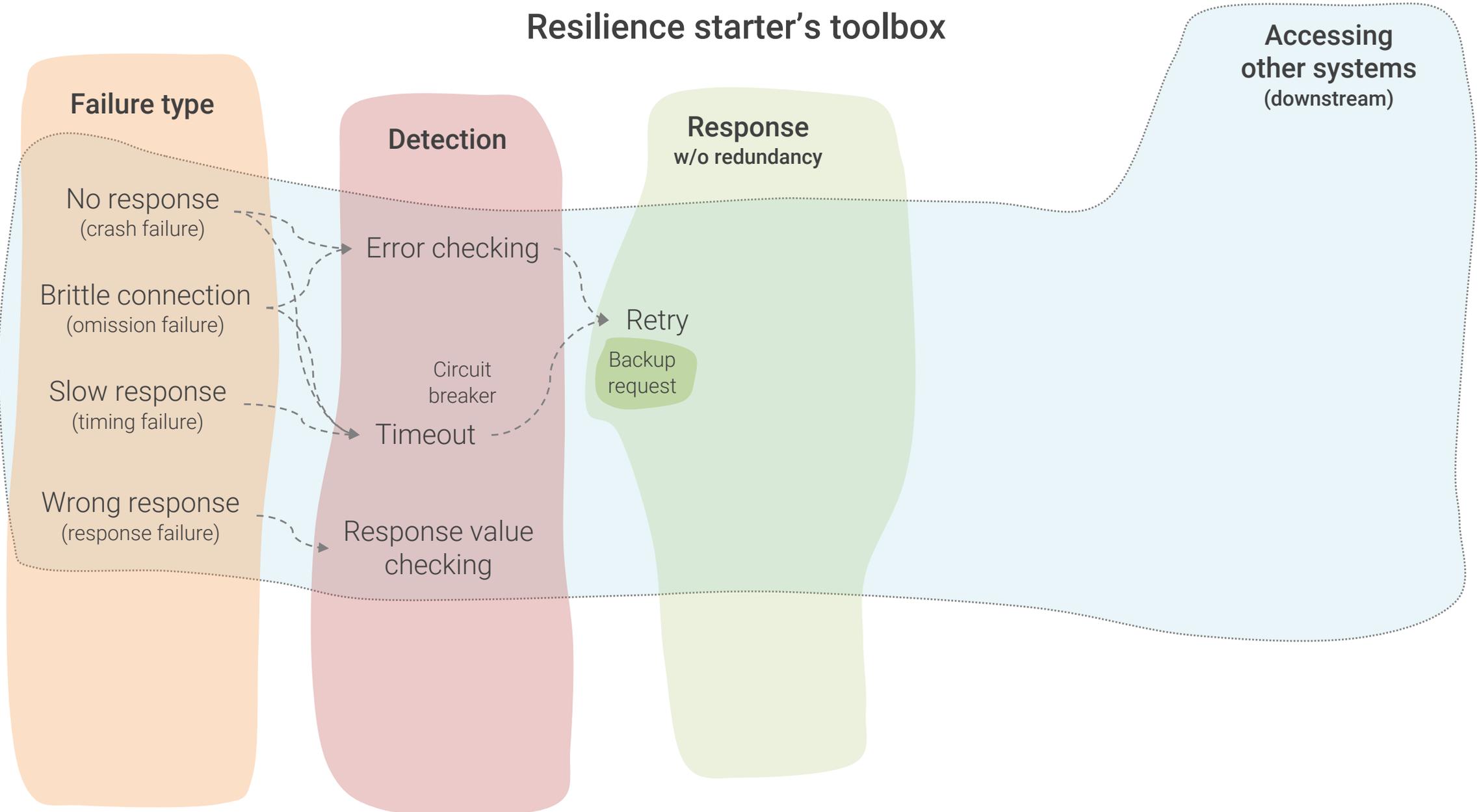




Retry

- 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



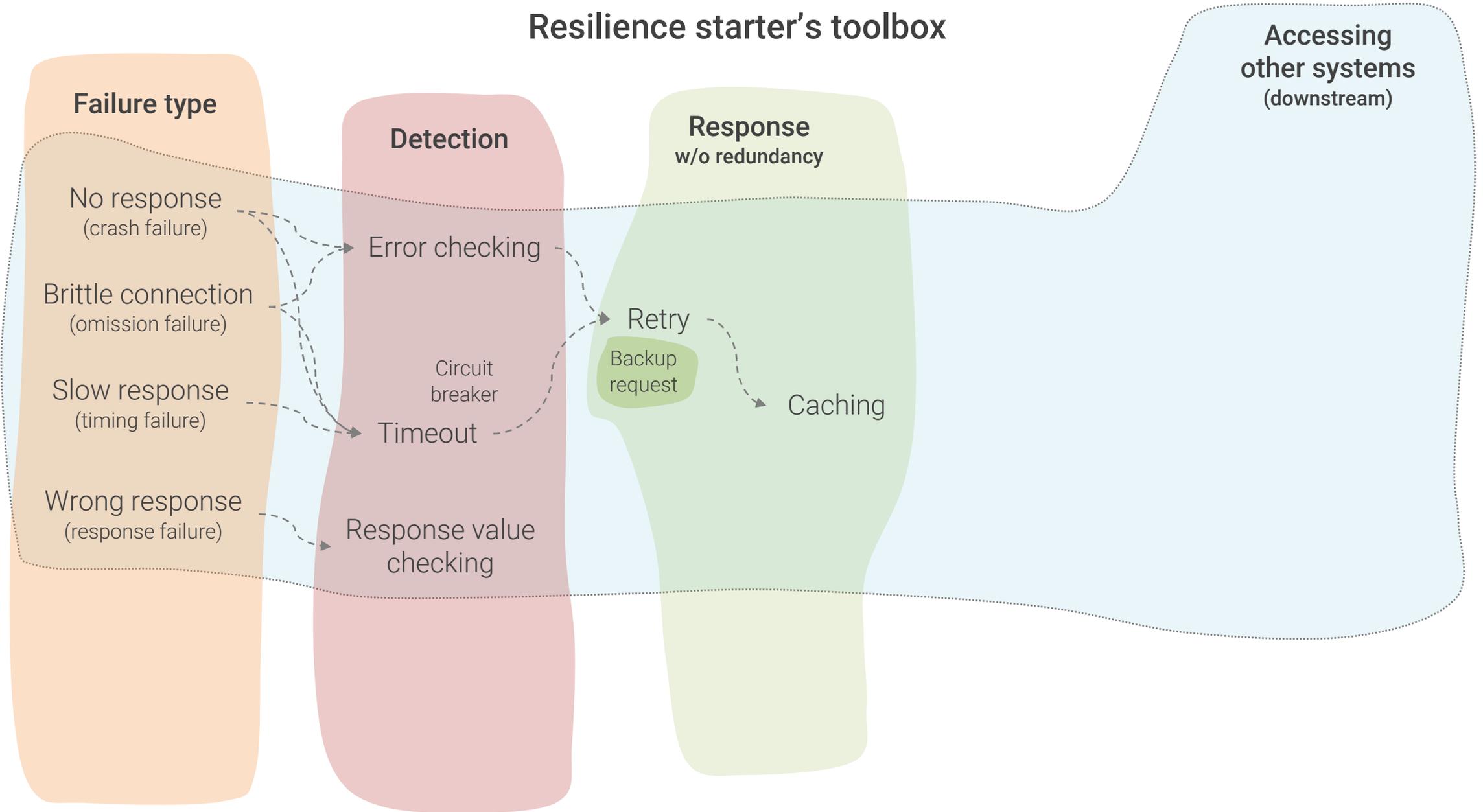
Backup request

- 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

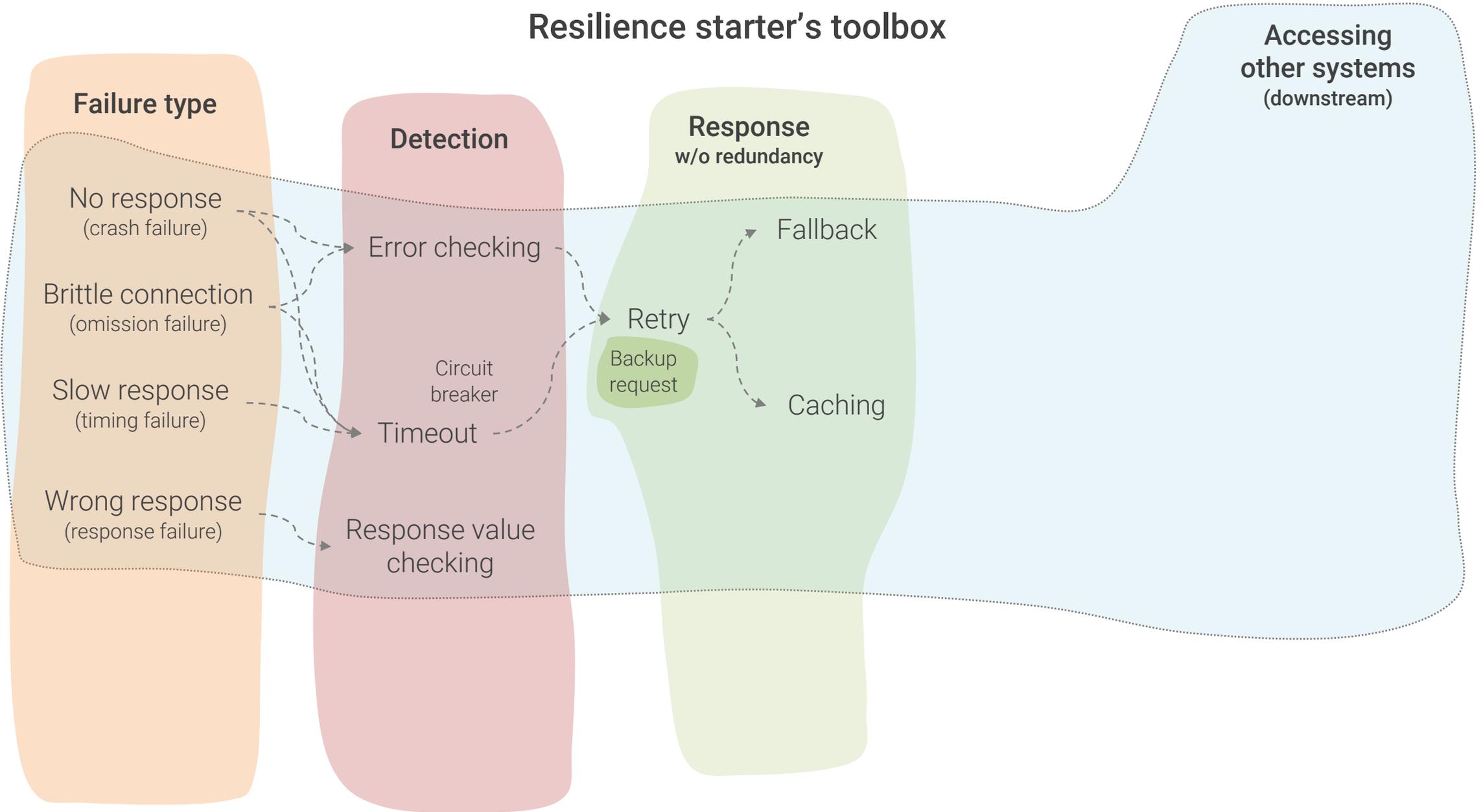




Caching

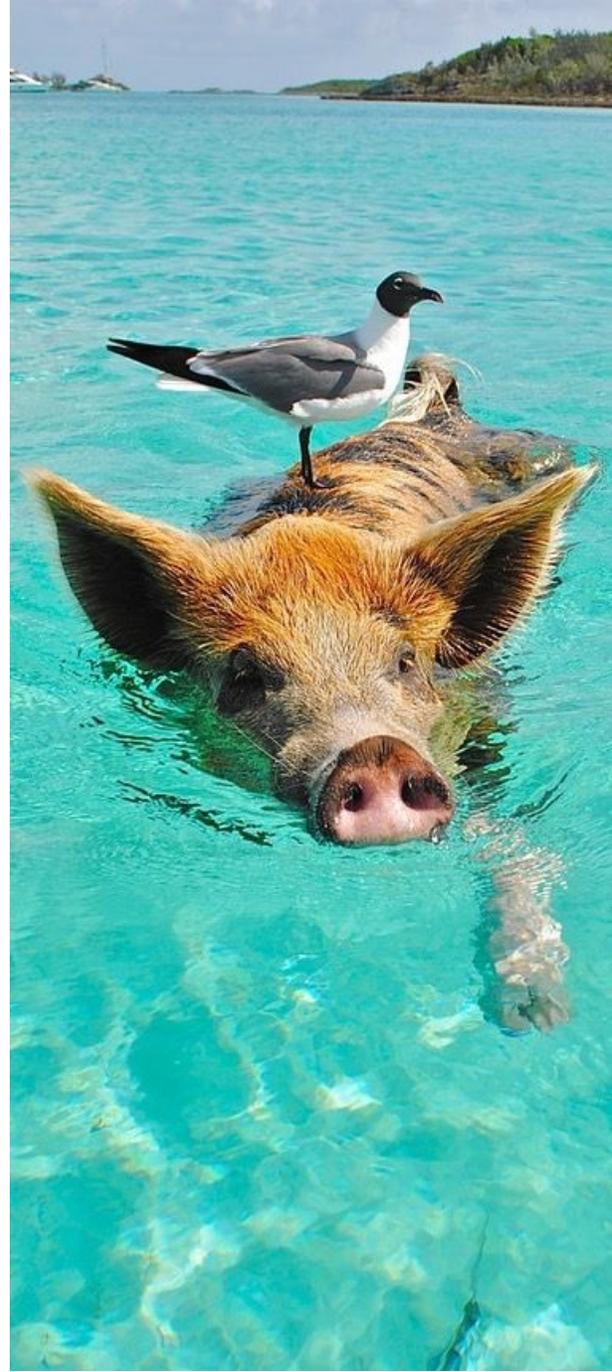
- Re-use responses from prior calls to downstream resources
- Can bridge temporary unavailability of resources
- Use with caution
 - Requires extra resources to store cached data
 - Leaves you with potentially stale data and all consistency issues associated with it
- Good tool and library support

Resilience starter's toolbox



Fallback

- Execute an alternative action if the original action fails
- Basis of 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



Adding retry and fallback

```
3 require File.expand_path("../config/initializers/spec_helper.rb", __FILE__)
4 # Prevent database truncation if the environment is production
5 abort("The Rails environment is running in production mode!") if Rails.env.production?
6 require 'spec_helper'
7 require 'rspec/rails'
8
9 require 'capybara/rspec'
10 require 'capybara/rails'
11
12 Capybara.javascript_driver = :webkit
13 Category.delete_all; Category.create(:name => "Category")
14 Shoulda::Matchers.configure do |config|
15   config.integrate do |with|
16     with.test_framework :rspec
17     with.library :rails
18   end
19 end
20 # Add additional requires below this line. You can use this to load
21 # Requires supporting ruby files with spec support, like support/
22 # spec/support/ and its subdirectories. Files starting with "rspec"
23 # run as spec files by default. Files starting with "helper"
24 # in _spec.rb will both be required and run as spec files.
25 # run twice. It is recommended that you use the "require" method
26 # to load spec.rb. You can configure the require method by using
```

```
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)
```

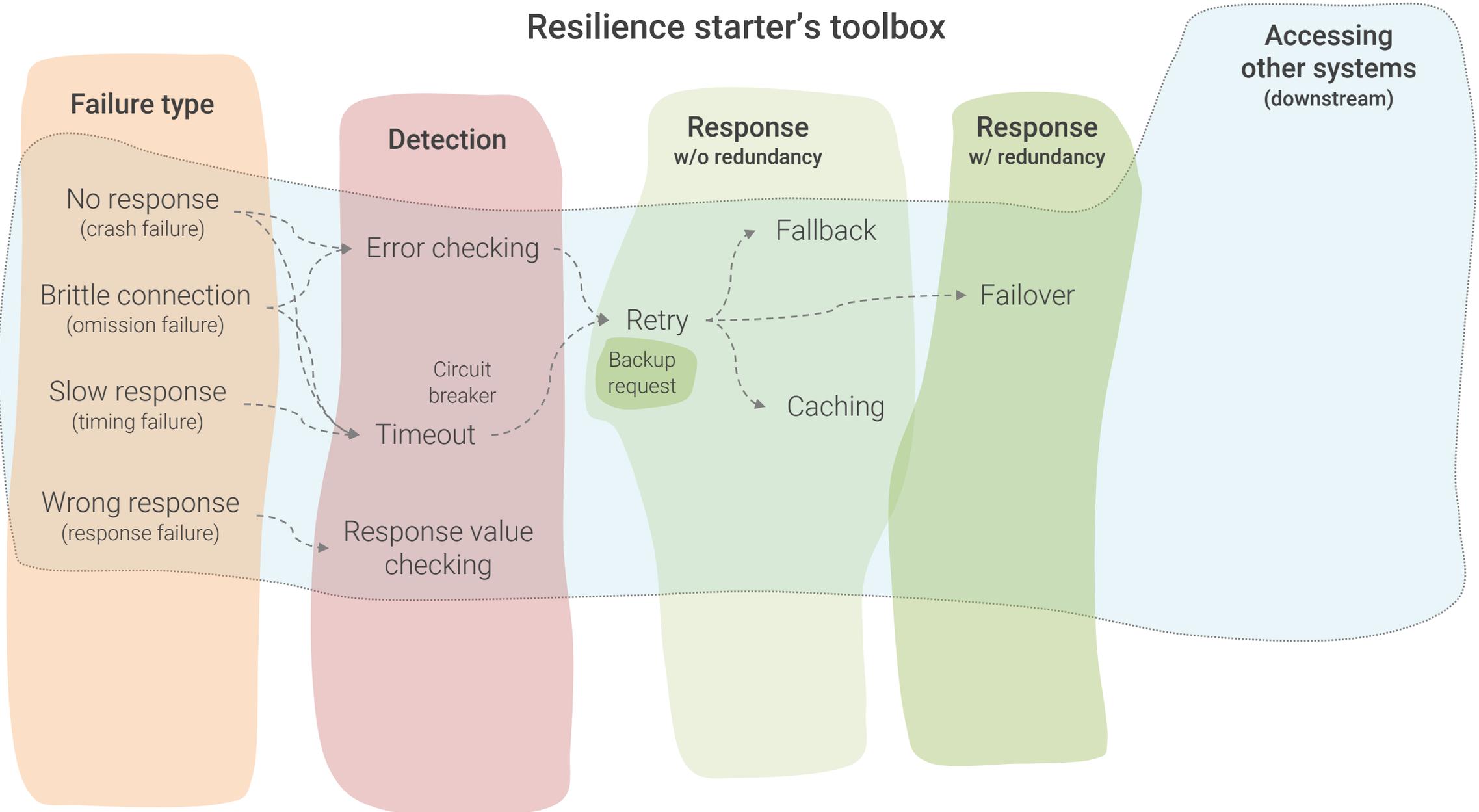
```
from urllib3 import PoolManager
from urllib3.exceptions import HTTPError

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)
```

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

Fallback

Retry

Backup
request

Caching

Response w/ redundancy

Failover

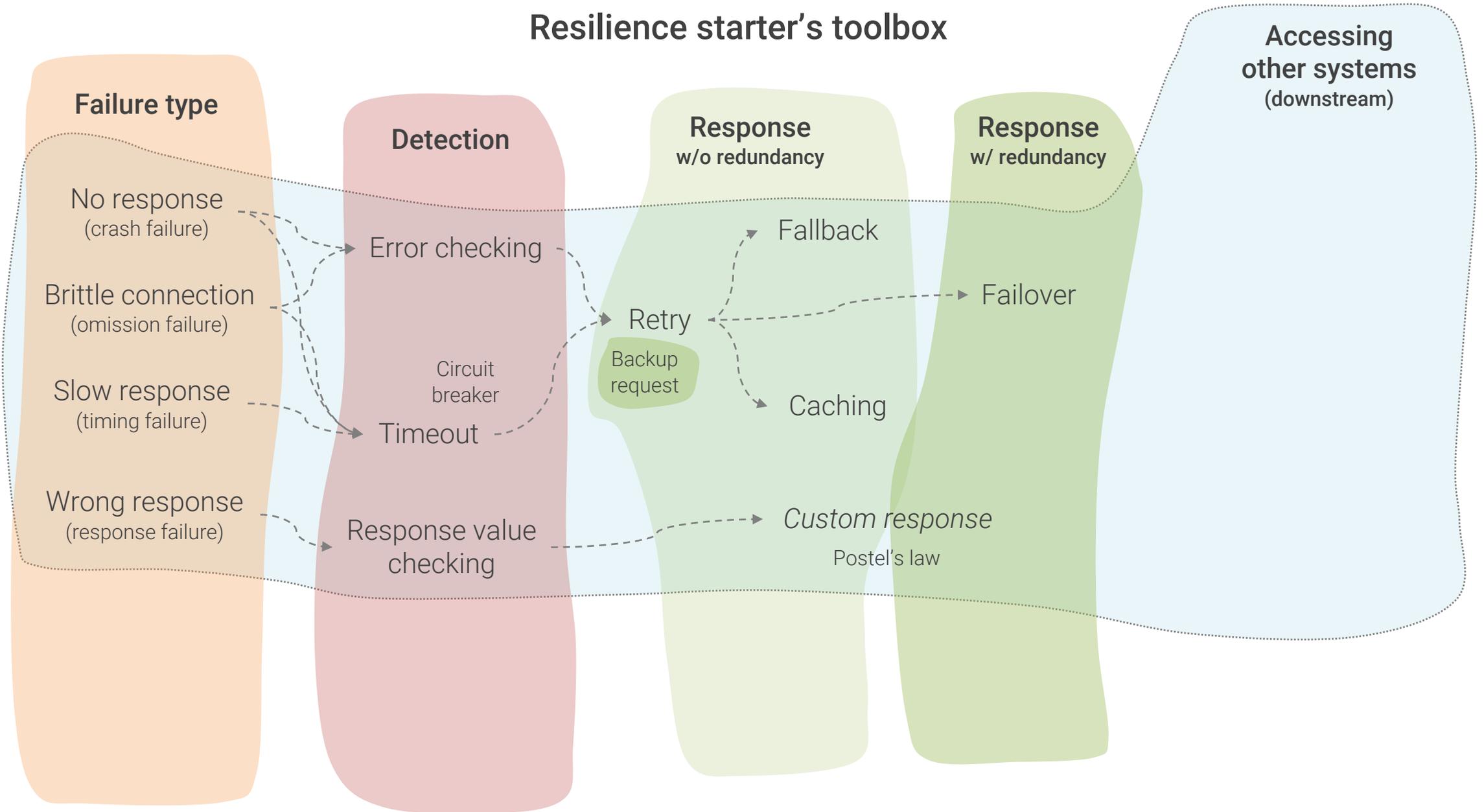
Accessing
other systems
(downstream)



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



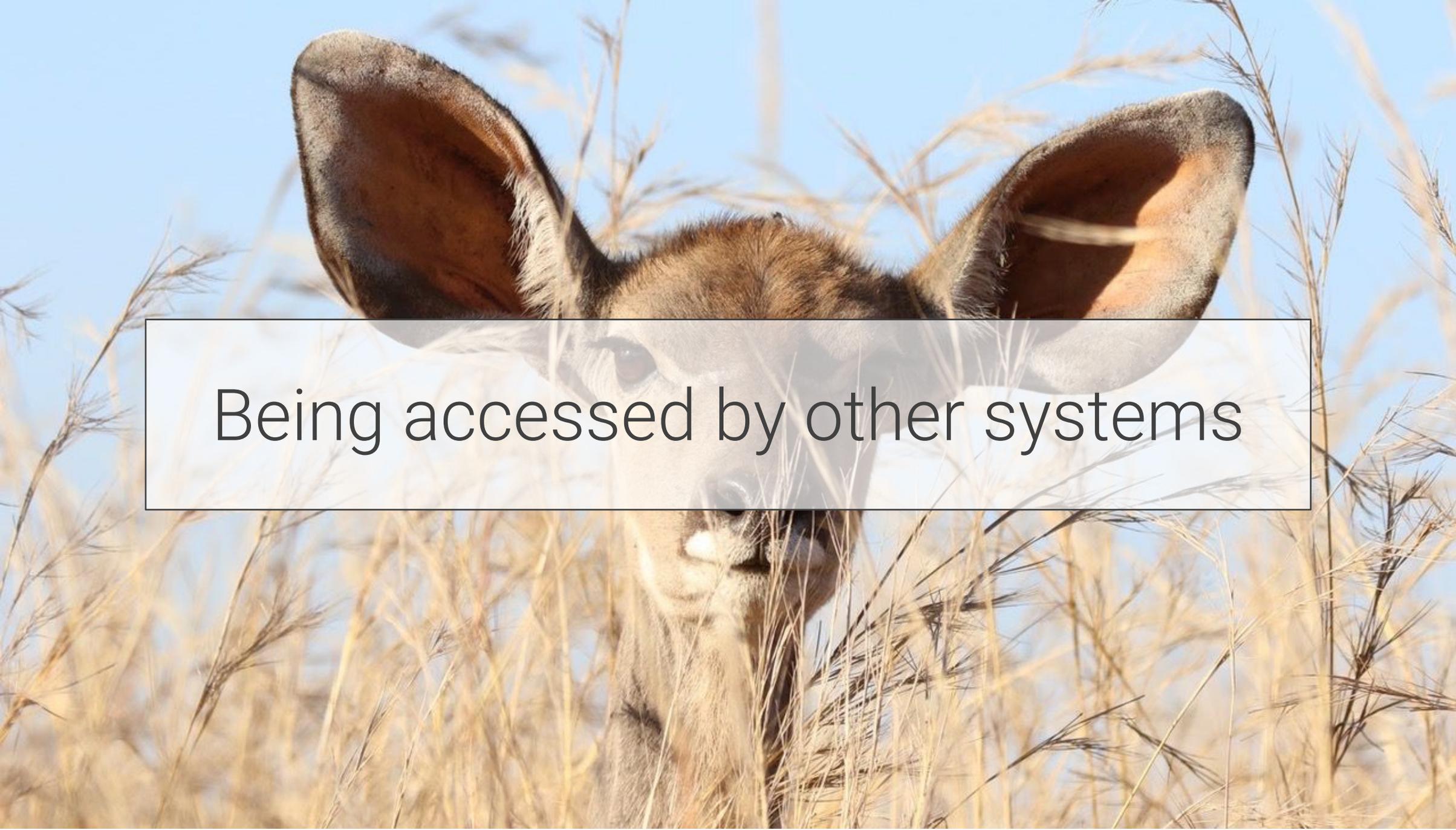
Remember Postel's law

“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”)

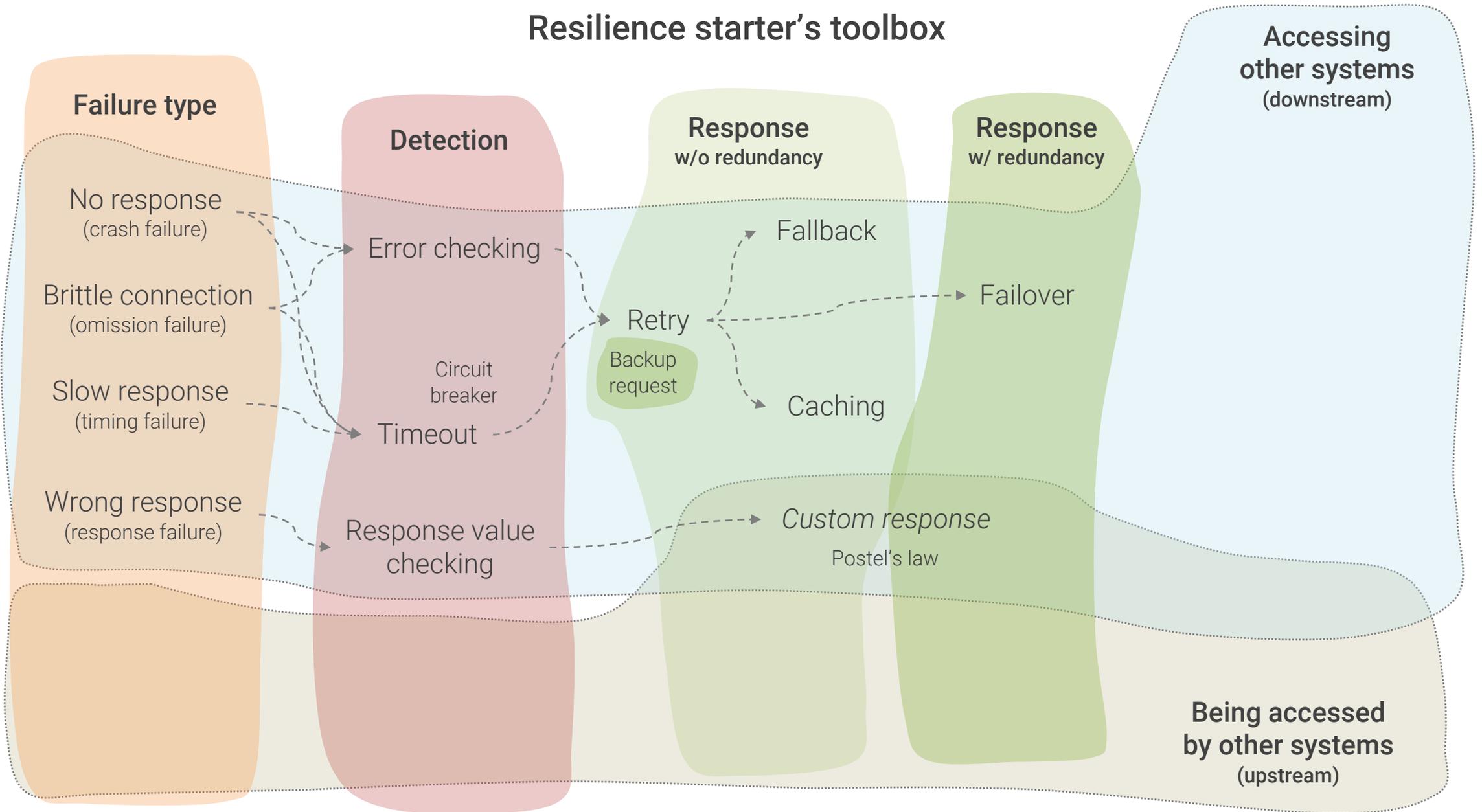
see also: https://en.wikipedia.org/wiki/Robustness_principle



A close-up photograph of a deer's head and large ears, partially obscured by a semi-transparent white box containing text. The deer is looking directly at the camera, and its ears are prominent. The background consists of dry, golden-brown grasses and a clear blue sky.

Being accessed by other systems

Resilience starter's toolbox

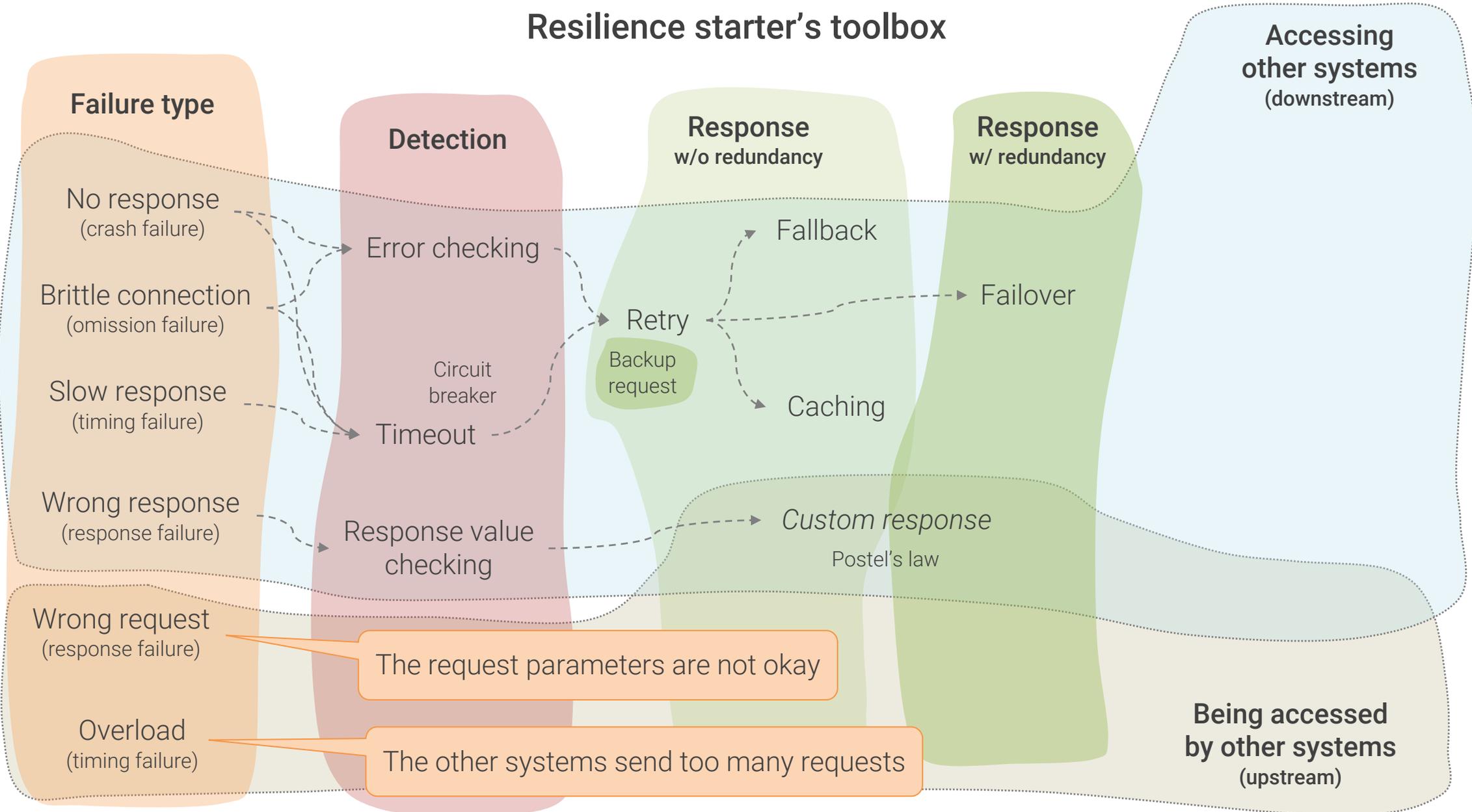


```
from fastapi import FastAPI

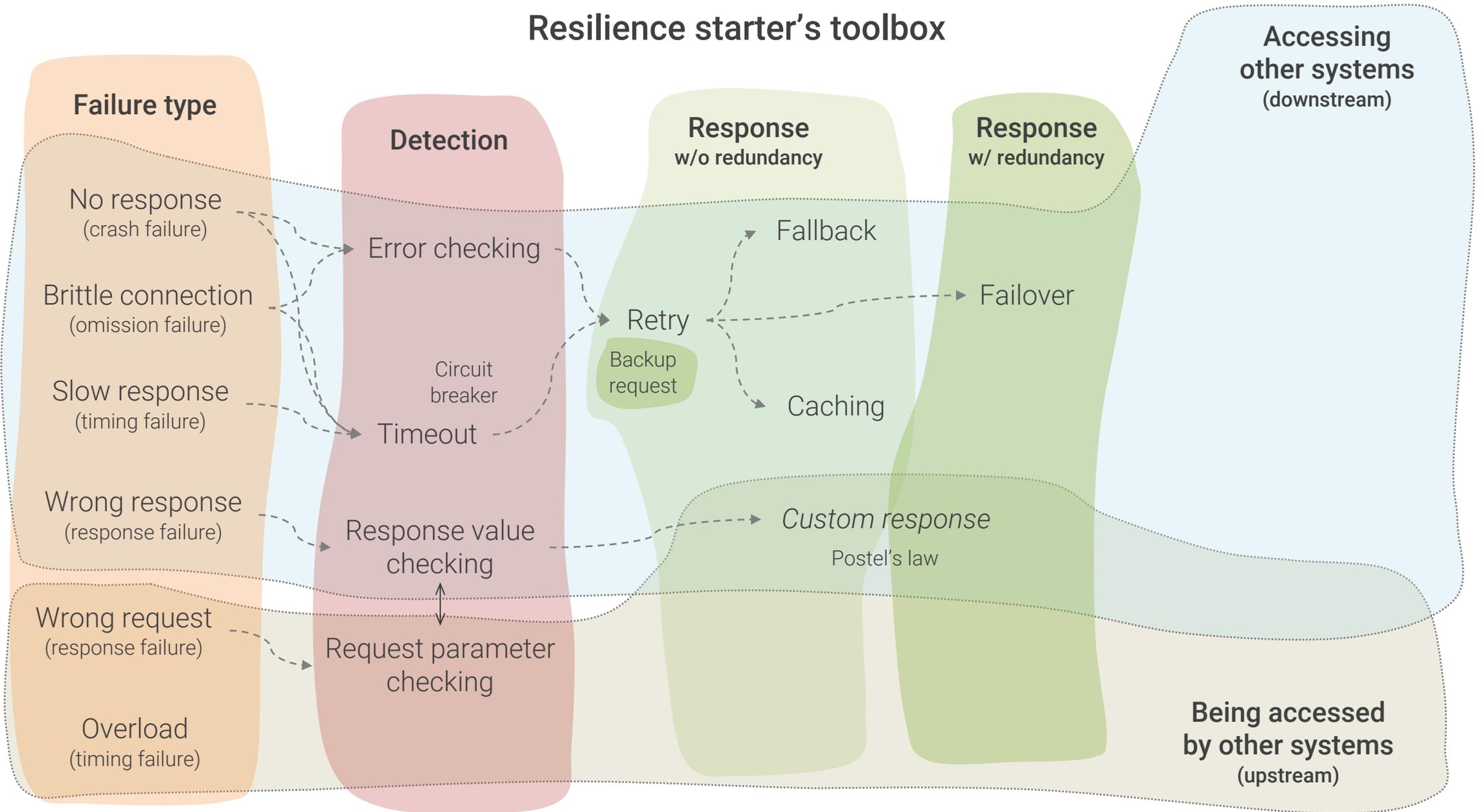
app = FastAPI()

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

Resilience starter's toolbox



Resilience starter's toolbox





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

Adding parameter checking

```
3 require File.expand_path("../support/paths", __FILE__)
4 # Prevent database truncation if the environment is production
5 abort("The Rails environment is running in production mode!")
6 require 'spec_helper'
7 require 'rspec/rails'
8
9 require 'capybara/rspec'
10 require 'capybara/rails'
11
12 Capybara.javascript_driver = :webkit
13 Category.delete_all; Category.create(:name => "Category")
14 Shoulda::Matchers.configure do |config|
15   config.integrate do |int|
16     with_test_frameworks [:rspec]
17     with_library :rails
18   end
19 end
20
21 # Add additional requires below this line. You can require
22 # Requires supporting ruby files with spec support, like
23 # spec/support/ and its subdirectories. Files starting with "rspec"
24 # run as spec files by default. Files starting with "helper"
25 # in _spec.rb will both be required and run as spec files.
26 # run twice. It is recommended that you use the "require"
27 # method to load spec.rb. You can configure the require method
```

```
from fastapi import FastAPI

app = FastAPI()

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

```
from fastapi import FastAPI, Path
```

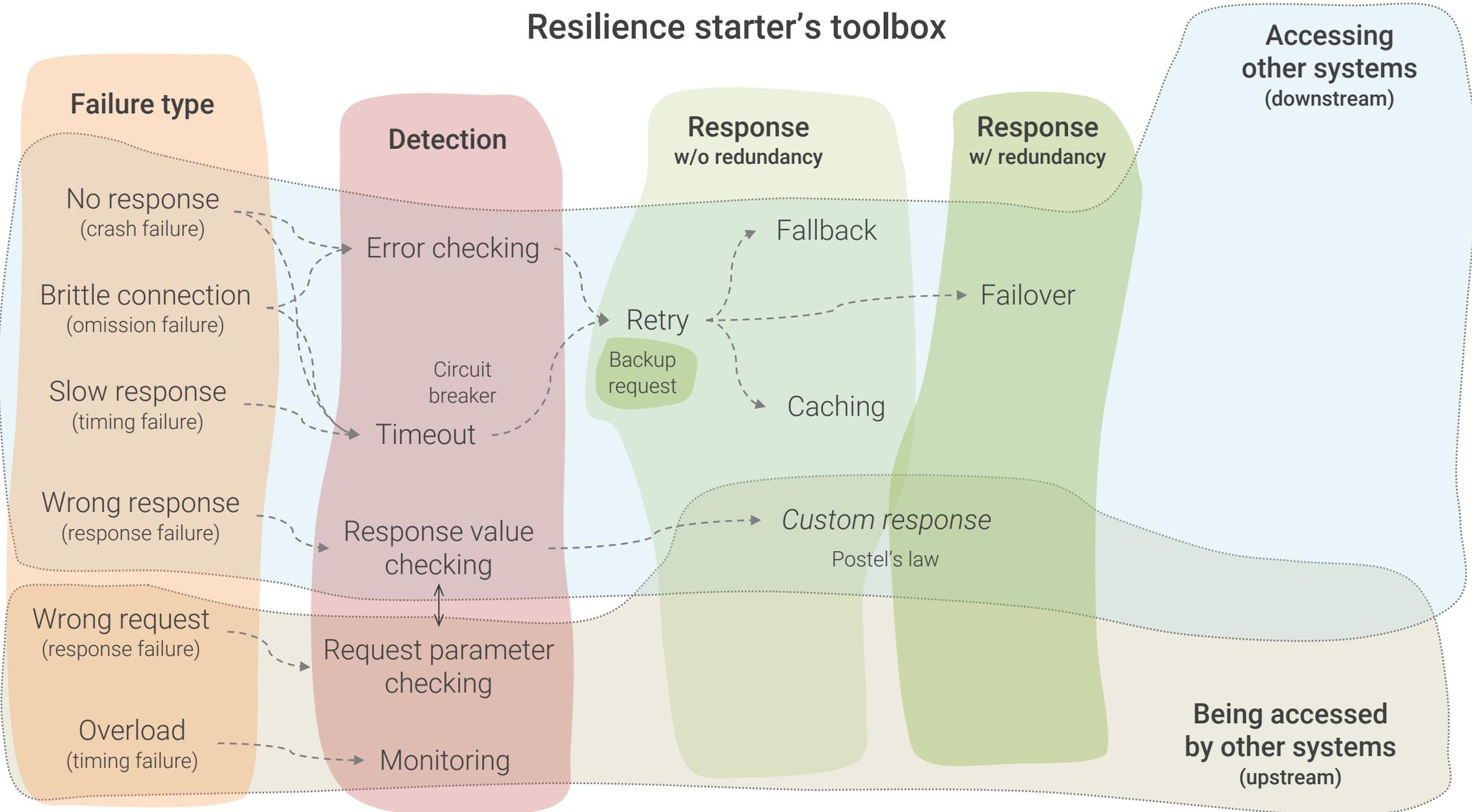
```
app = FastAPI()
```

```
@app.get("/square/{number}")
```

```
def read_square(number: int = Path(..., gt=0, lt=100)):
```

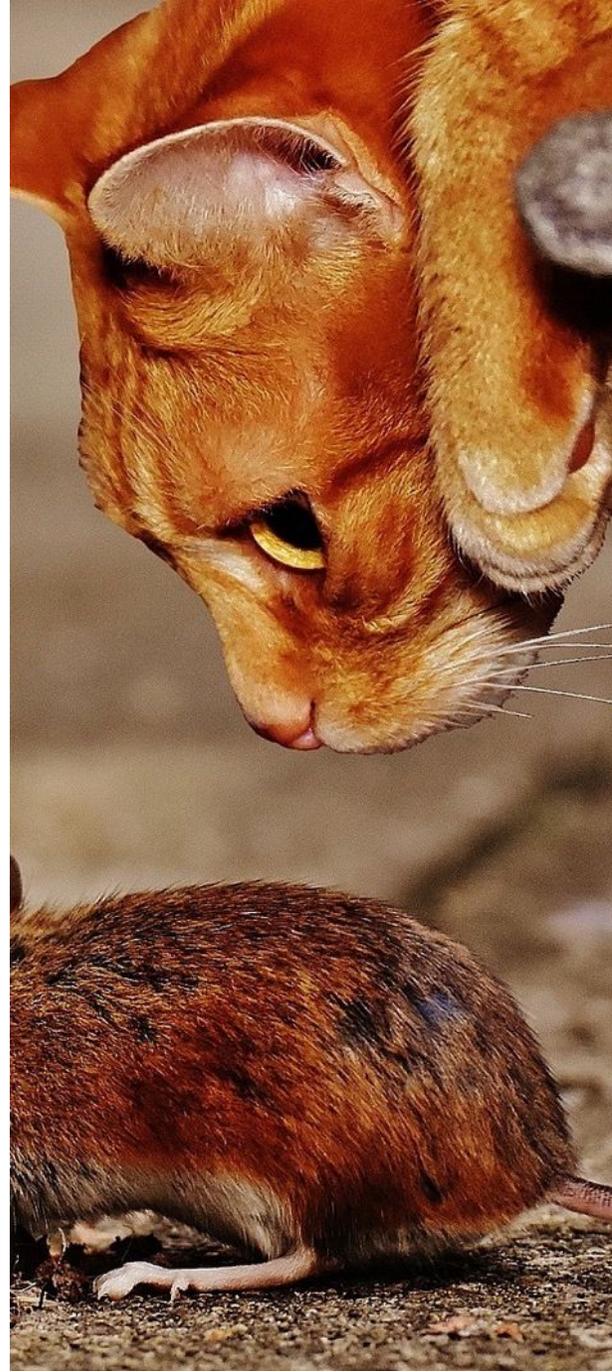
```
    return {"result": number*number}
```

Resilience starter's toolbox

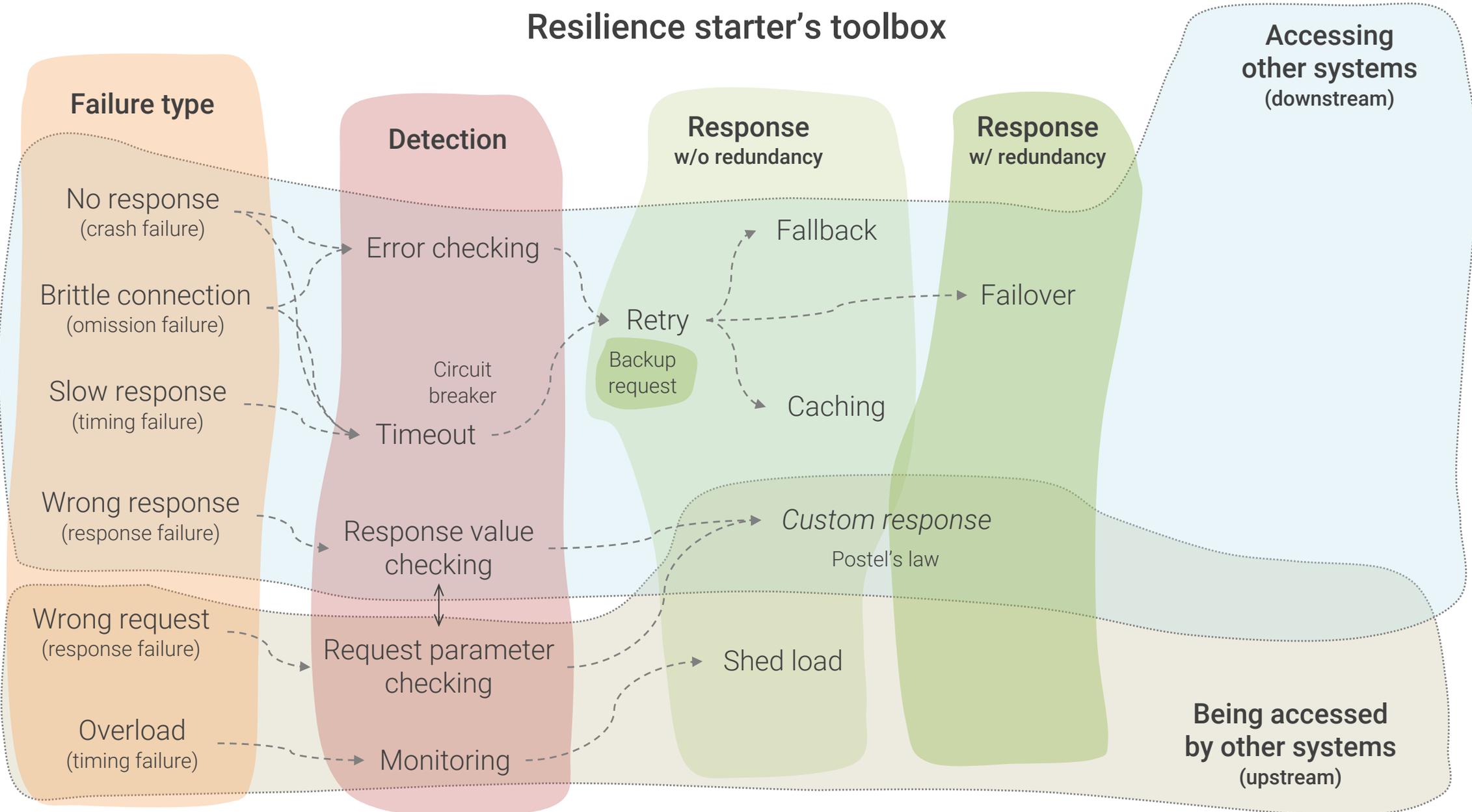


Monitoring

- 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



Resilience starter's toolbox

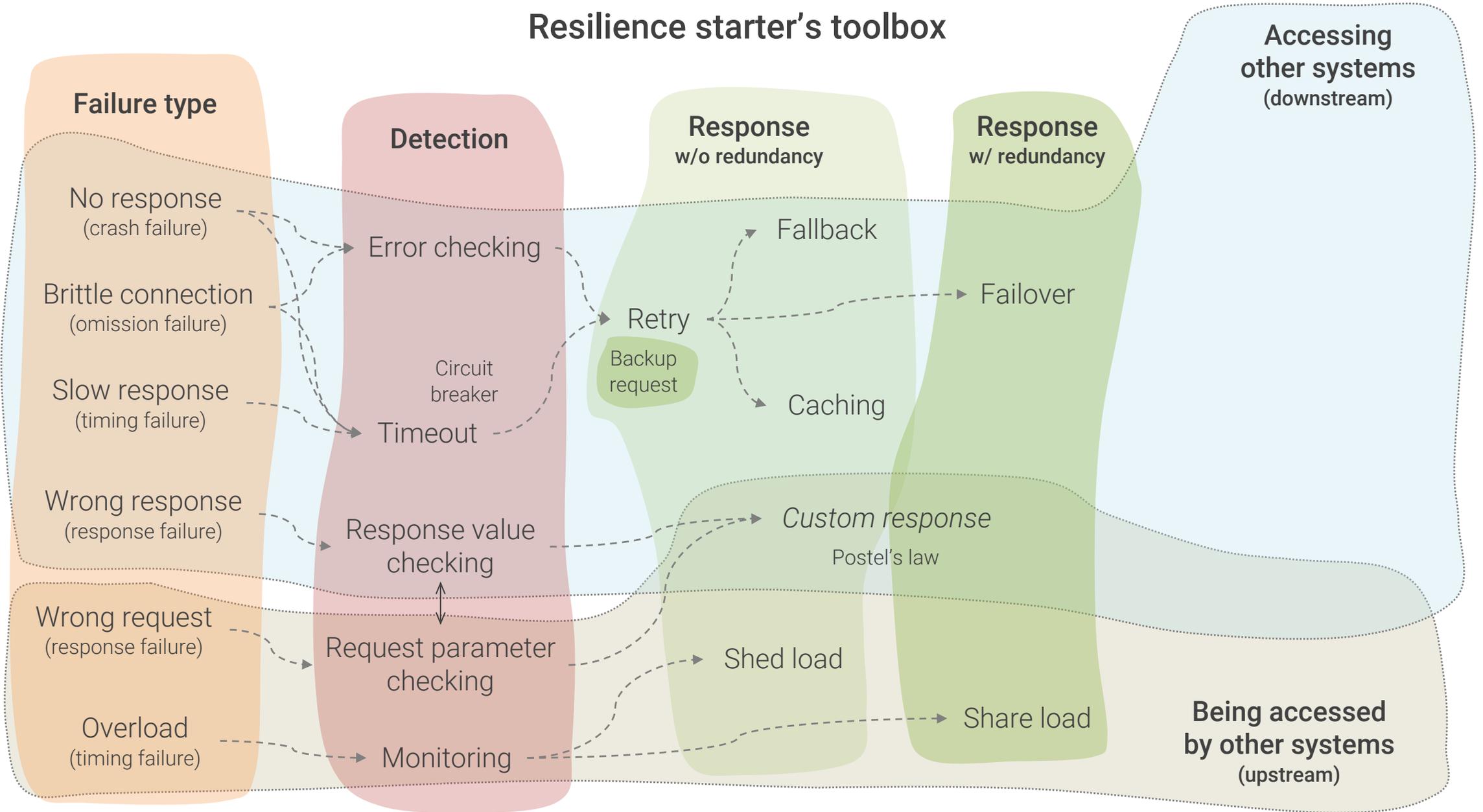




Shed load

- 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



Share load

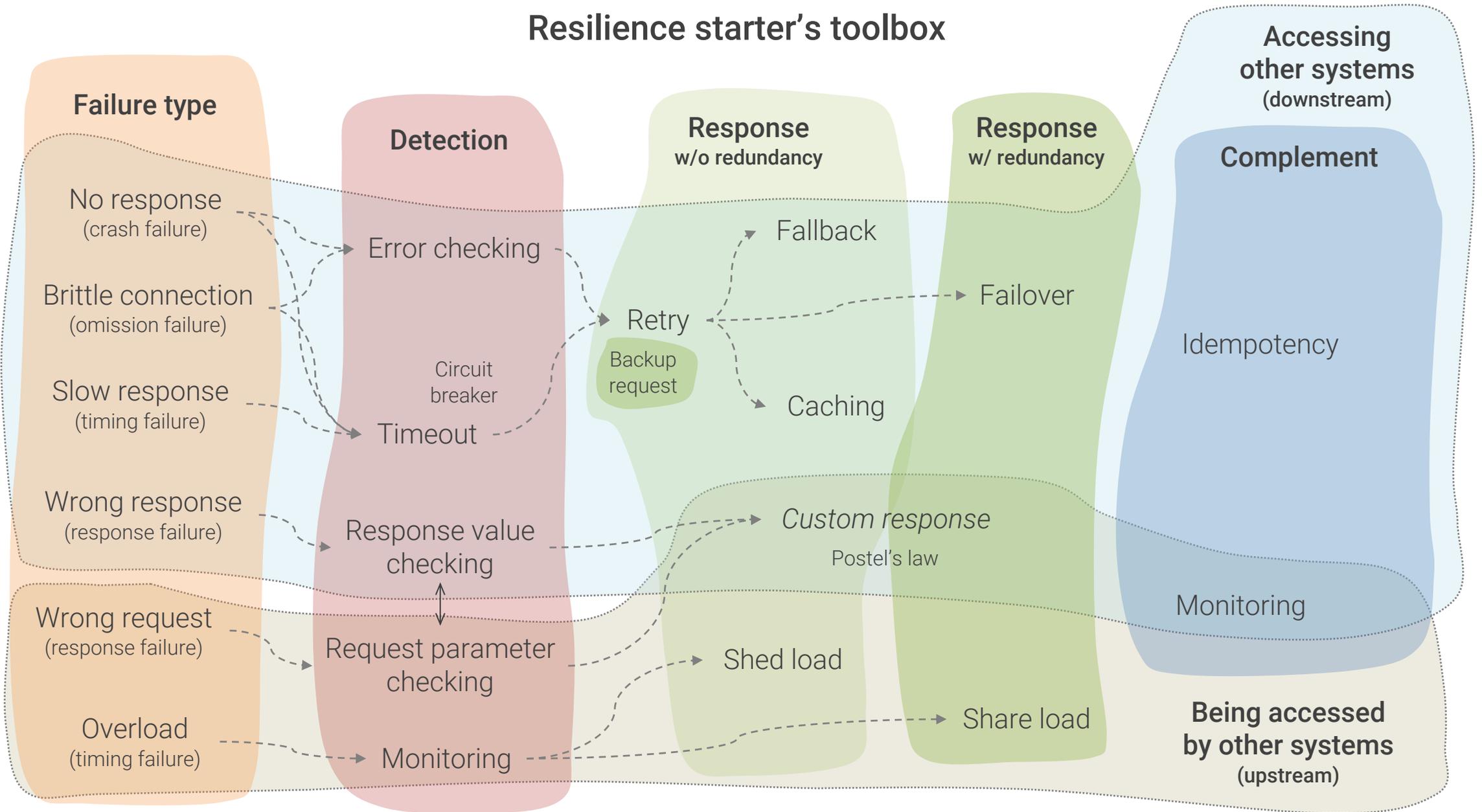
- Share load between resources to keep throughput good
- Use if additional resources for load sharing can be used
- Can be implemented statically or dynamically (“auto-scaling”)
- Very good tool support available
- Minimize synchronization needed between resources
 - Synchronization needs kill scalability



A large African elephant and its calf are walking across a dry, open savanna landscape. The adult elephant is in the foreground, facing left, with its trunk reaching down to touch the ground. The calf is walking behind it, also facing left. The ground is a mix of dry grass and reddish-brown soil. The background shows a vast, open plain under a bright sky.

Useful complementing patterns

Resilience starter's toolbox

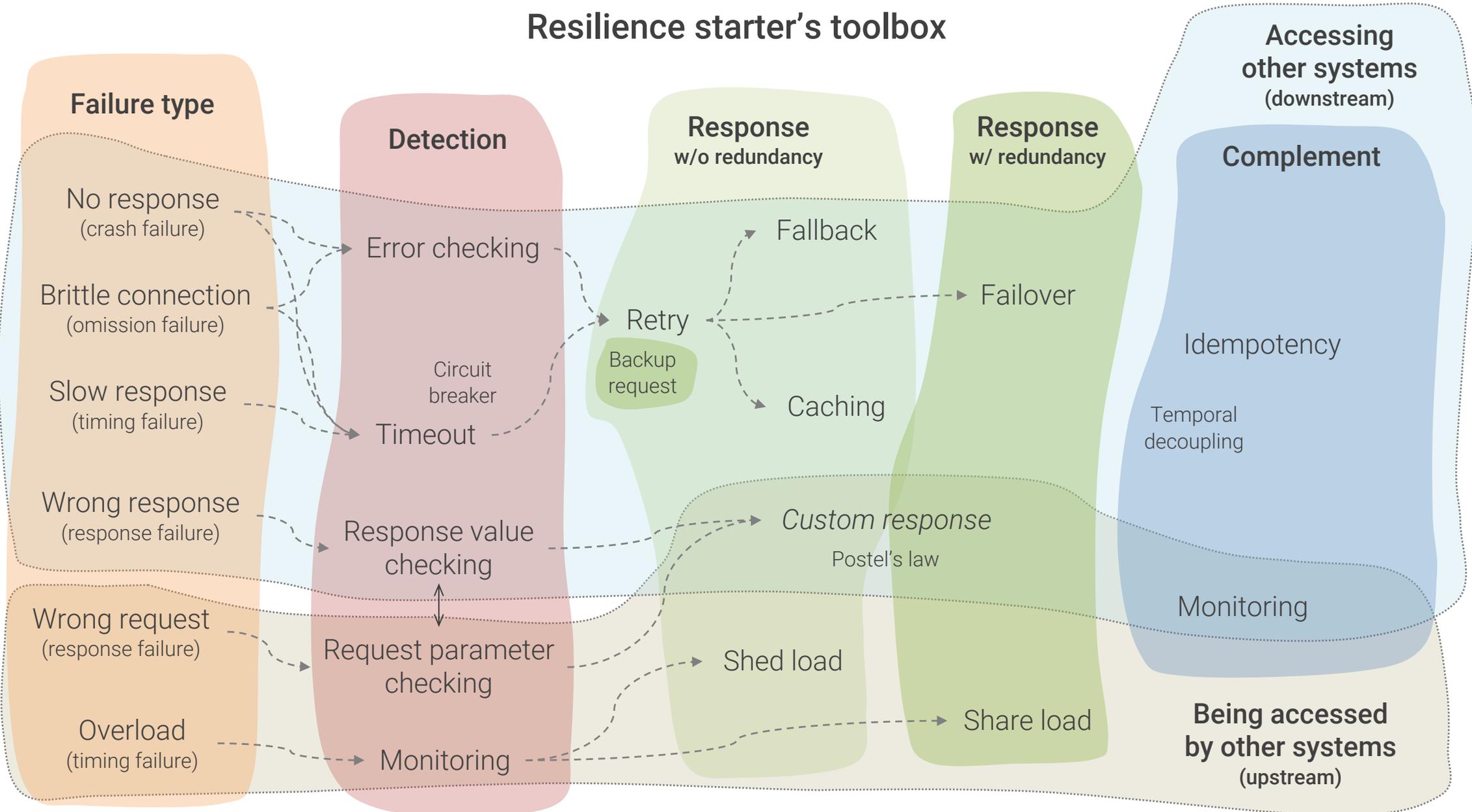




Idempotency

- 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

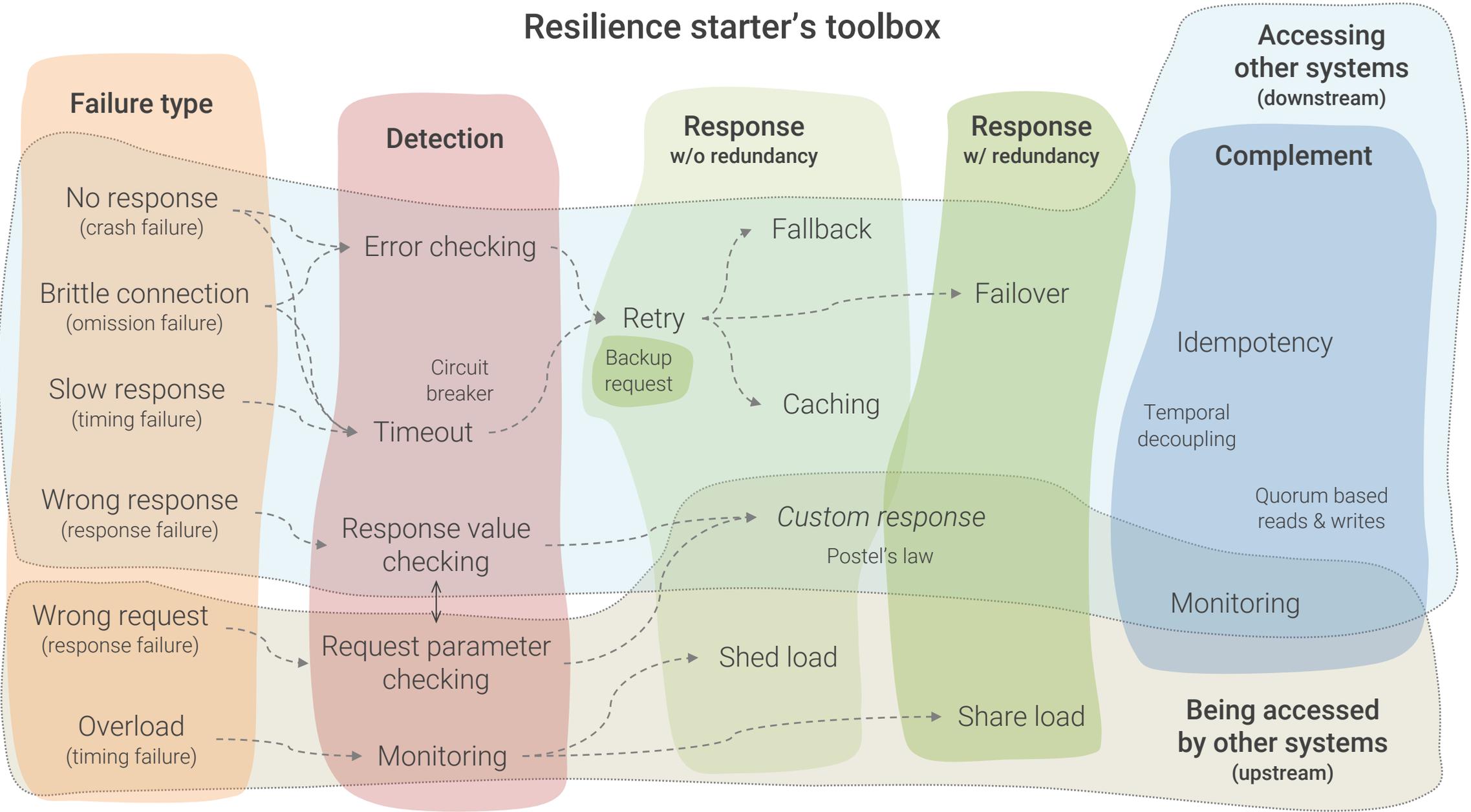


Temporal decoupling

- 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

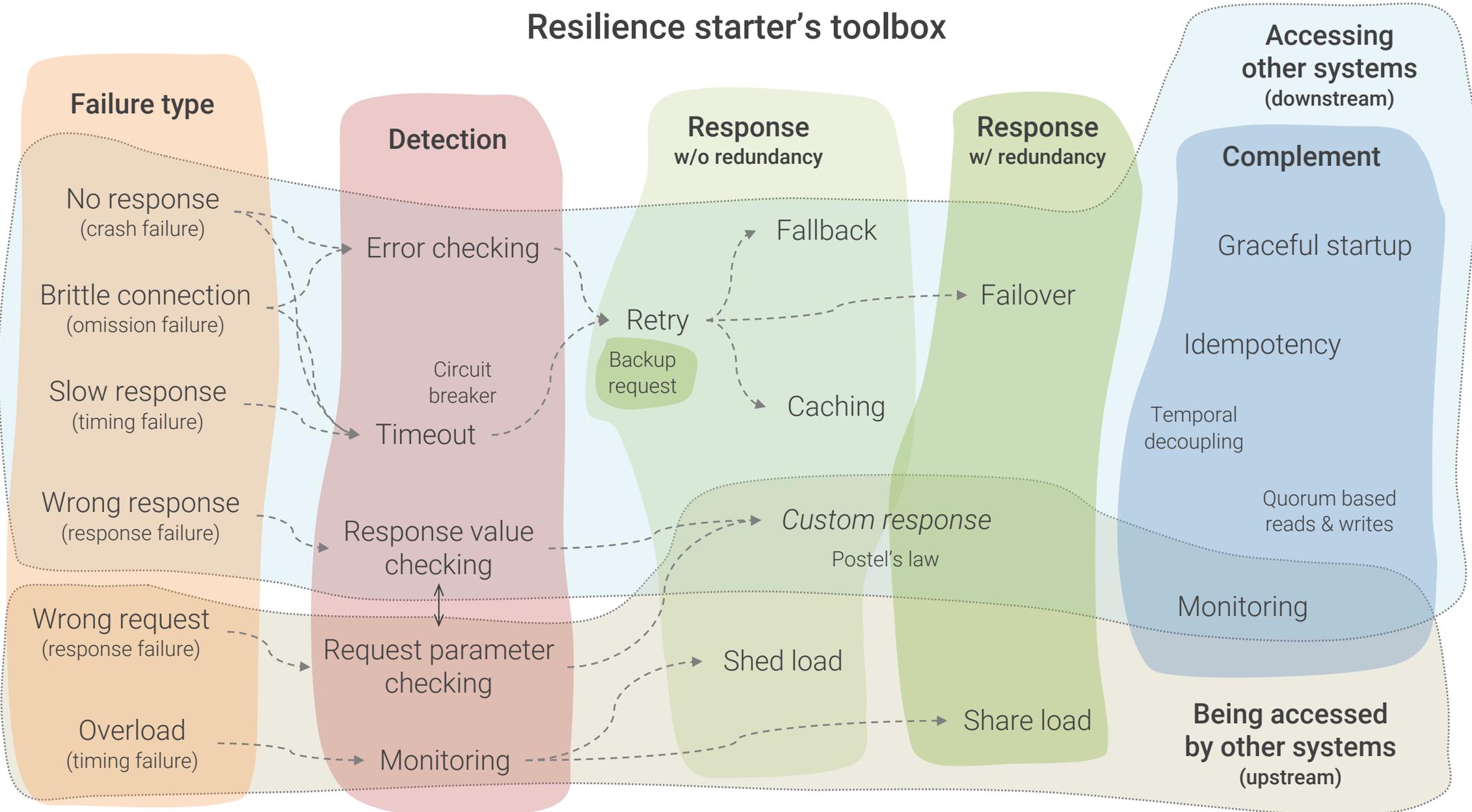




Quorum-based reads and writes

- 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

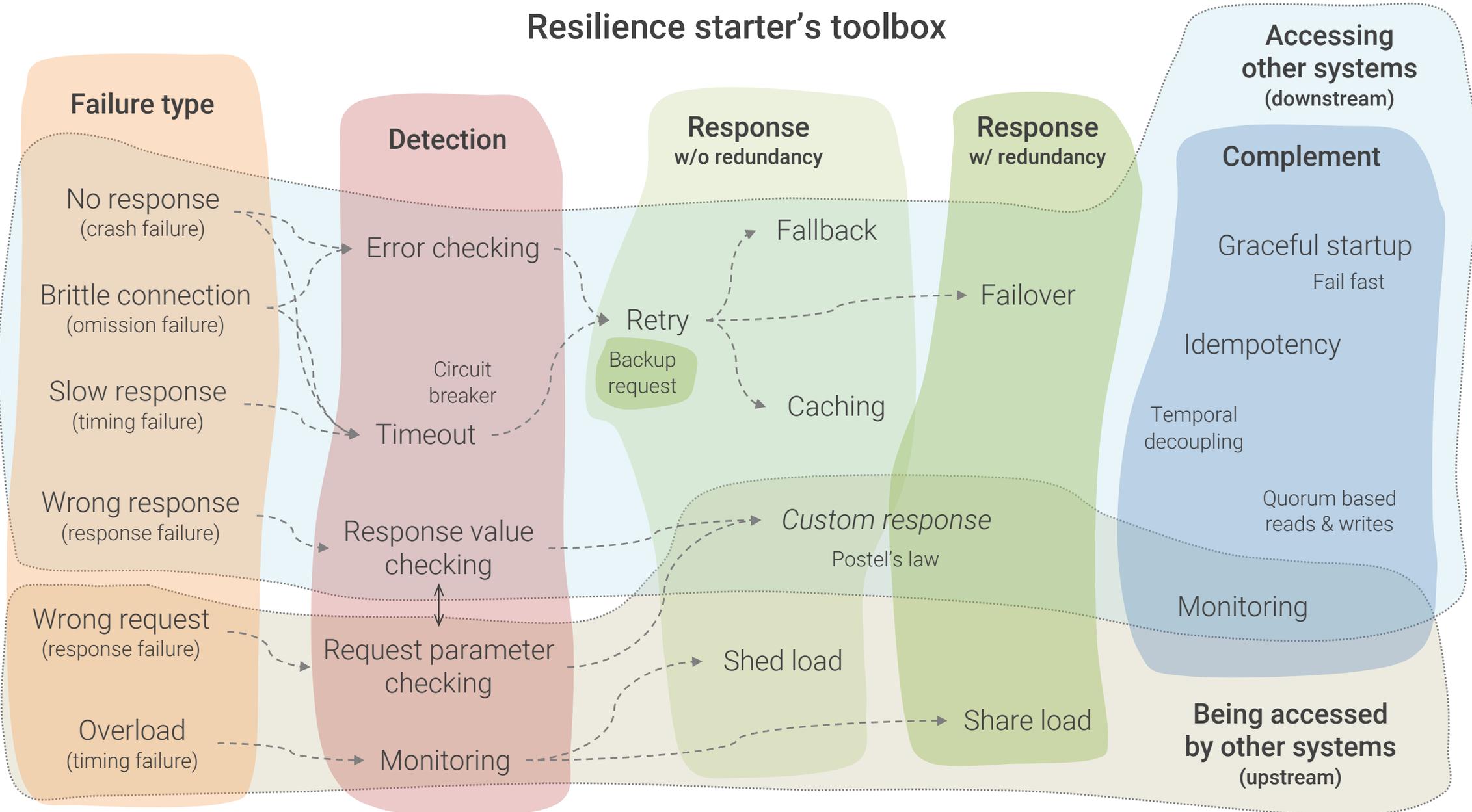


Graceful startup

- 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



Fail fast

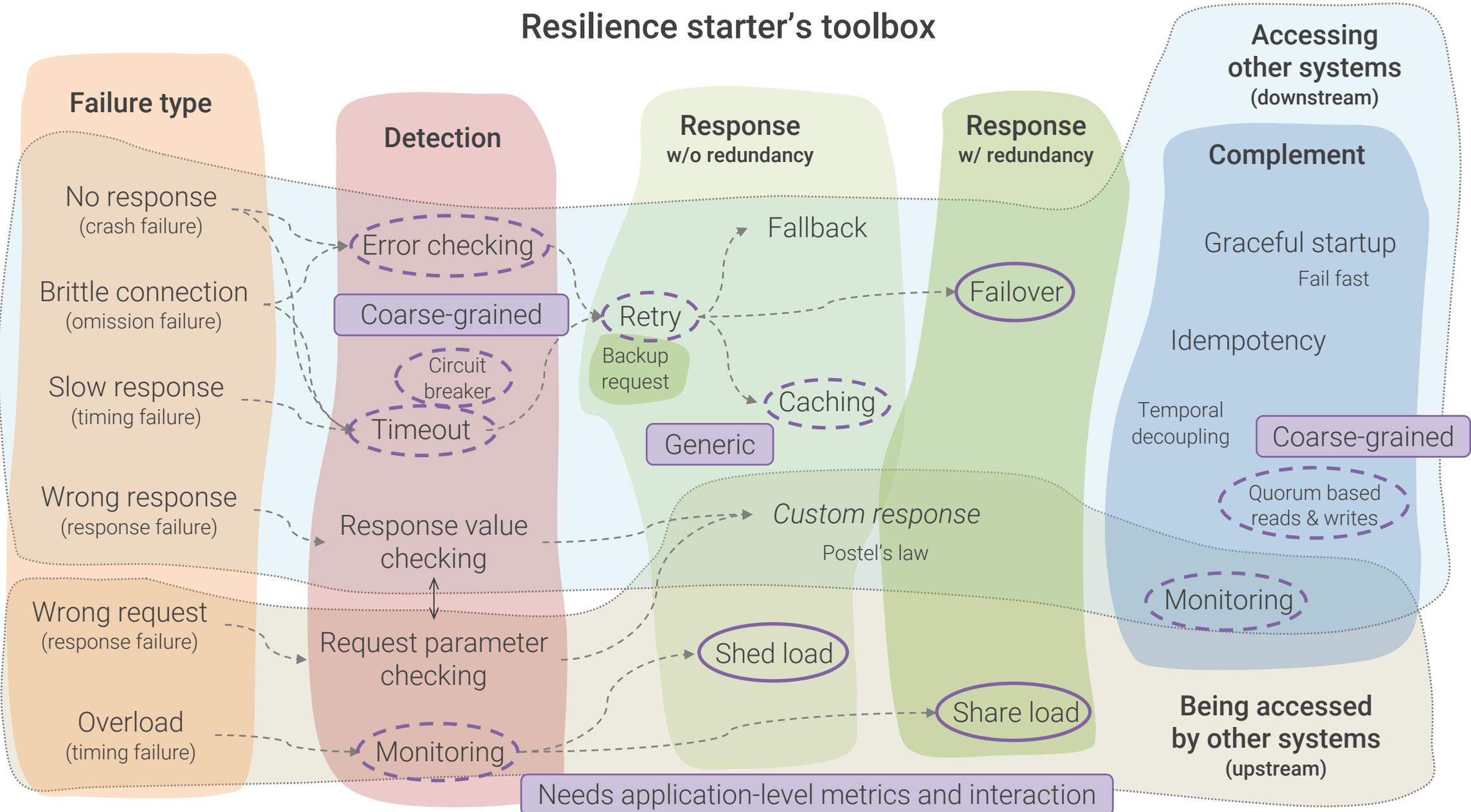
- “If you know you’re going to fail, you better fail fast”
- Usually implemented in front of costly actions
- Saves time and resources by avoiding foreseeable failures
- Useful in normal operations mode
- Can be counterproductive in startup mode



A group of meerkats is shown in profile, standing on a large, light-colored rock. The meerkats have brown and tan fur with dark markings around their eyes. One meerkat is in the foreground, looking towards the left. Another is behind it, and a third is to the right, looking towards the right. A fourth meerkat is partially visible in the background. A semi-transparent white box with a thin black border is overlaid on the middle of the image, containing the text "What can we delegate to the infrastructure level?".

What can we delegate to the infrastructure level?

Resilience starter's toolbox



A group of meerkats is perched on a large, light-colored rock. In the foreground, a meerkat is looking towards the camera. Behind it, another meerkat is looking to the right. A third meerkat is visible in the background, also looking right. A semi-transparent white box with a thin black border is centered over the image, containing the text "But that is still a lot to implement".

But that is still a lot to implement

A close-up photograph of a dog's face, showing its eyes, nose, and fur. The dog has dark eyes and a white patch on its muzzle. A semi-transparent white rectangular box is overlaid across the middle of the image, containing the text "But what should be the alternative?".

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!

The image features a close-up of two hands, one from the left and one from the right, silhouetted against a bright, glowing sunset or sunrise. The hands are positioned to hold each other, with fingers interlaced, creating a heart-like shape. The background is a soft, warm gradient of orange and yellow, with the sun's light creating a strong lens flare effect. A semi-transparent white rectangular box is centered over the image, containing the text.

Everyone loves resilient applications



Wrap-up

Wrap-up

- Distribution makes resilient software design mandatory
 - Failures 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

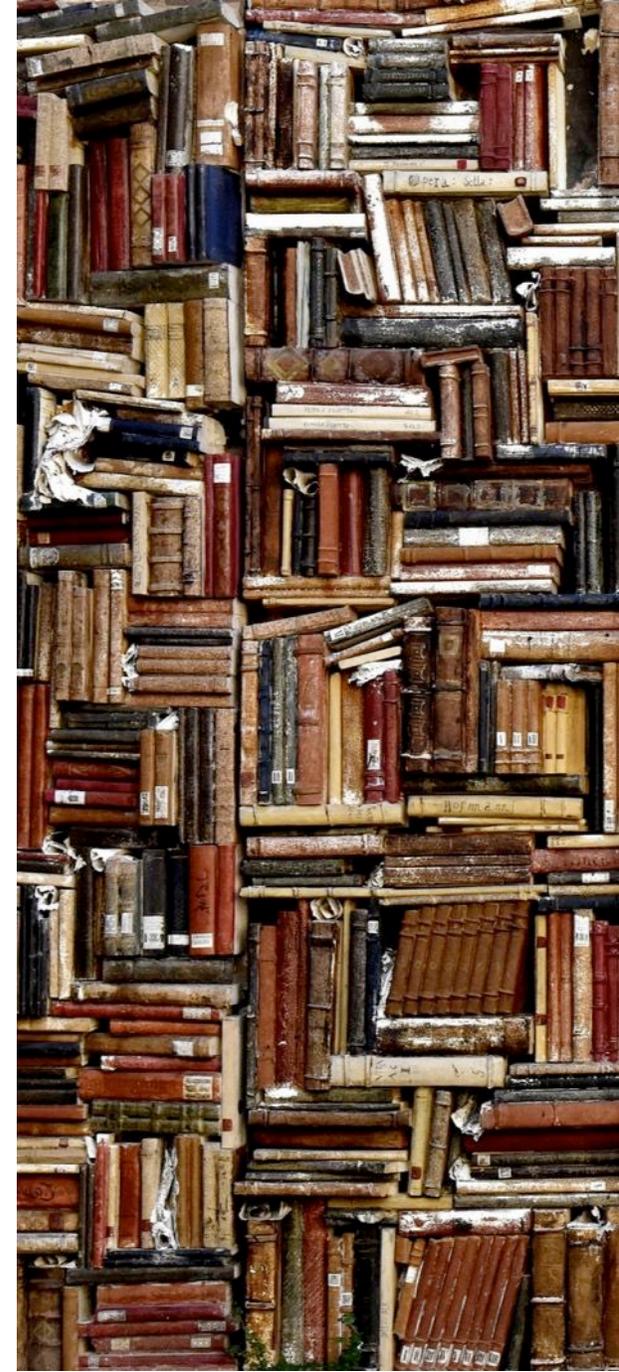
Release It! Design and Deploy Production-Ready Software,
Michael Nygard, 2nd edition, Pragmatic Bookshelf, 2018

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



Uwe Friedrichsen

Works @ codecentric

<https://twitter.com/ufried>

<https://www.speakerdeck.com/ufried>

<https://ufried.com/>

