Making Customers Happy With Reliable Systems

•••

Pedro Alves - SRE @ Zalando

A few numbers from Zalando

23 Markets (or countries)

> 46M active customers

>560M website visits per month

We build our own solutions for payments, logistic centers, content production, etc.

>2000 applications

>1200 engineers

Site Reliability Engineering

What is SRE?

A. A discipline
B. A set of practices
C. A mindset
D. All of the above

A bit of SRE history

"[SRE] is what happens when you ask a software engineer to design an operations function."

- Ben Treynor, VP Engineering @ Google

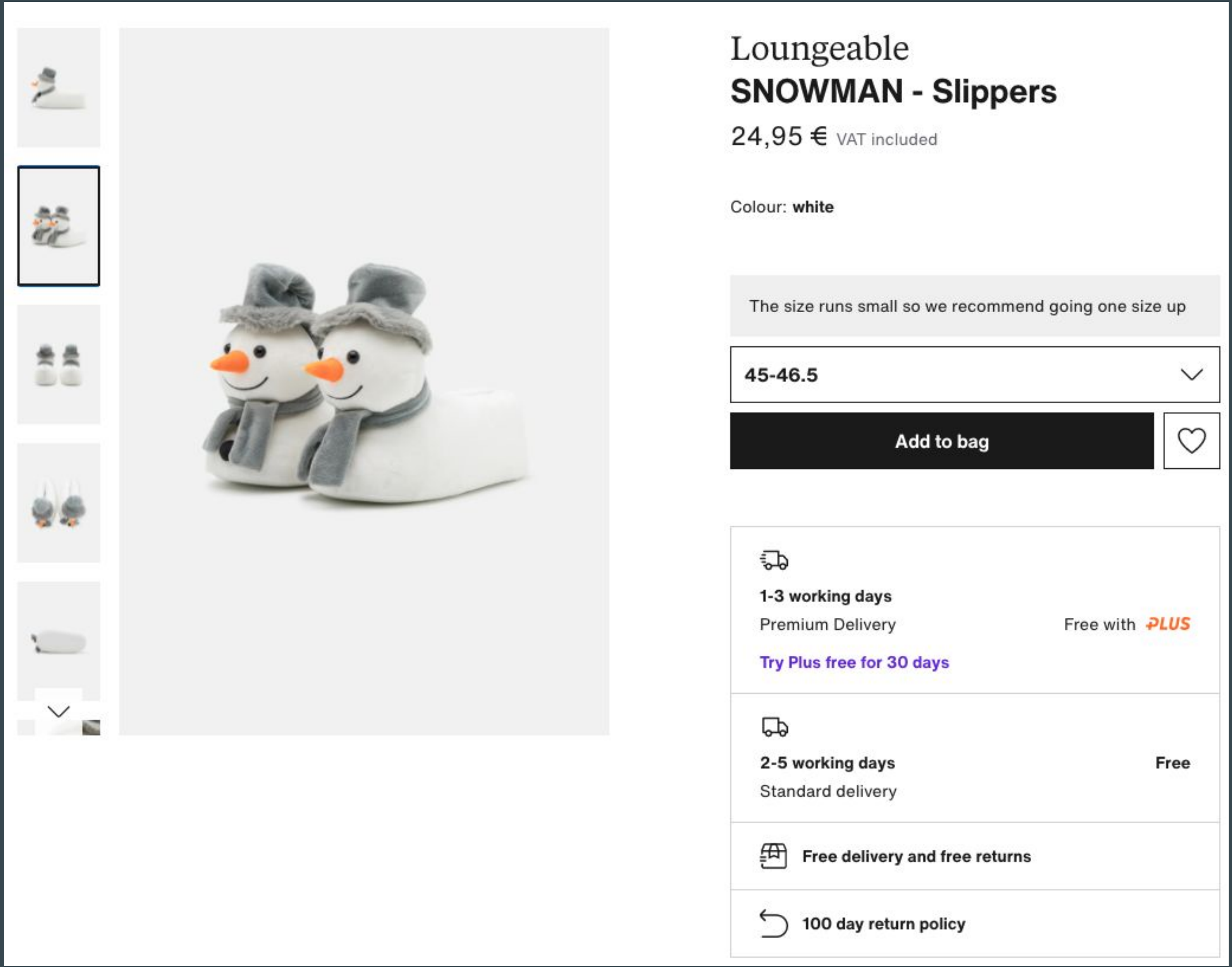
- Originally created at Google in 2003 as a way of handling software operations at Google's scale.
- To introduce SRE to the general public, Google published a book in 2016 that establishes the mindset and some of the SRE practices.
- Today SRE is widespread across the industry, but with many different flavors to it.

Reliability Engineering

Reliability Engineering

"Reliability is the most important feature [for your customers]."

- The SRE mindset aims at designing systems that are resilient and fault tolerant
- Those systems, when facing an issue, will either:
  - Self remediate
  - Tolerate faults
  - Make it easy for system operators to handle an issue
- SRE does this by accepting that **systems will fail**.
- We embrace failure to **learn from it** and make our systems **more reliable.**



Loungeable

SNOWMAN - Slippers

24,95 € VAT included

Colour: **white**

The size runs small so we recommend going one size up

45-46.5

Add to bag

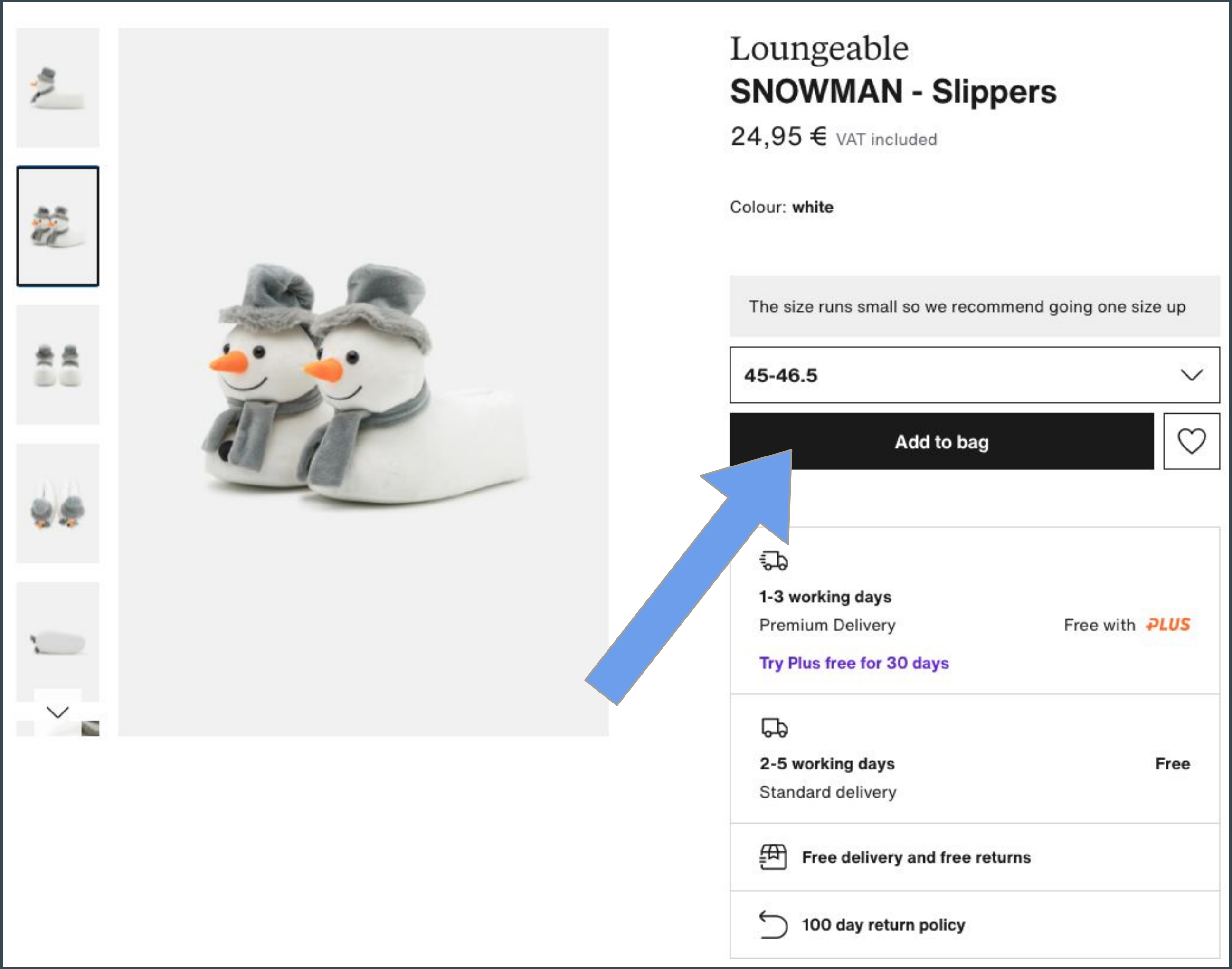
1-3 working days
Premium Delivery — Free with PLUS

Try Plus free for 30 days

2-5 working days — **Free**
Standard delivery

Free delivery and free returns

100 day return policy



Loungeable

SNOWMAN - Slippers

24,95 € VAT included

Colour: **white**

The size runs small so we recommend going one size up

45-46.5

Add to bag

1-3 working days

Premium Delivery Free with PLUS

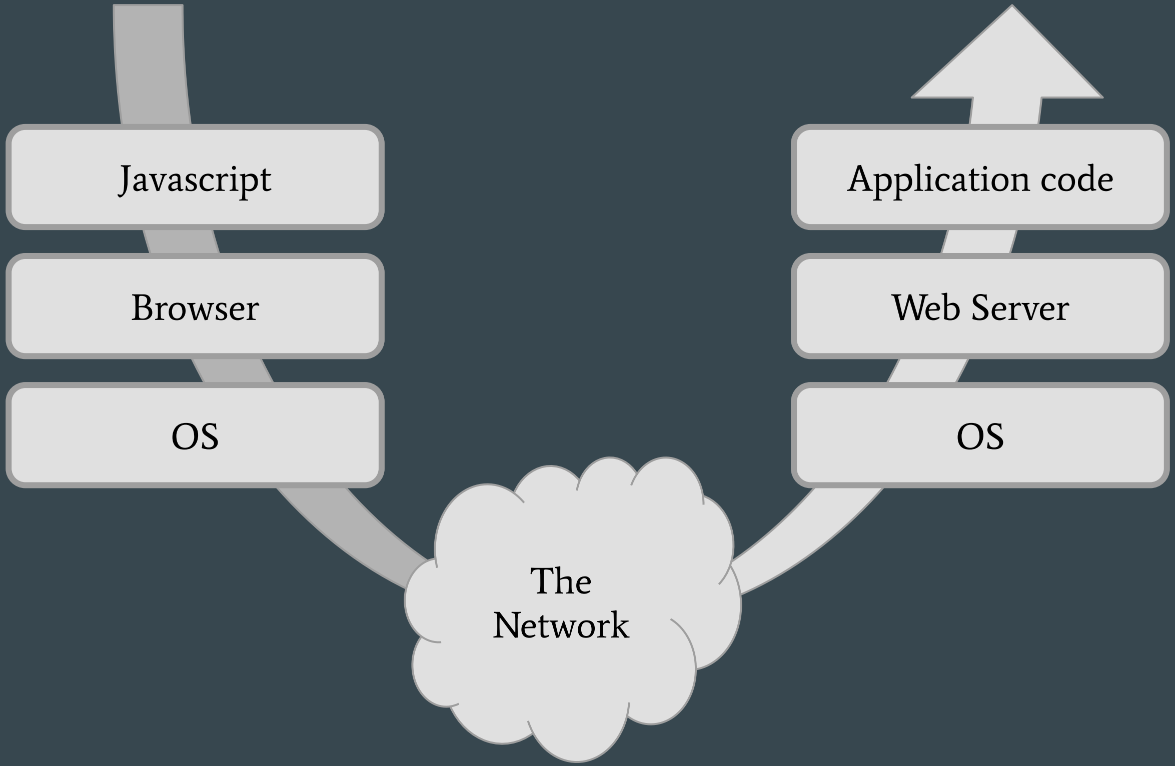
Try Plus free for 30 days

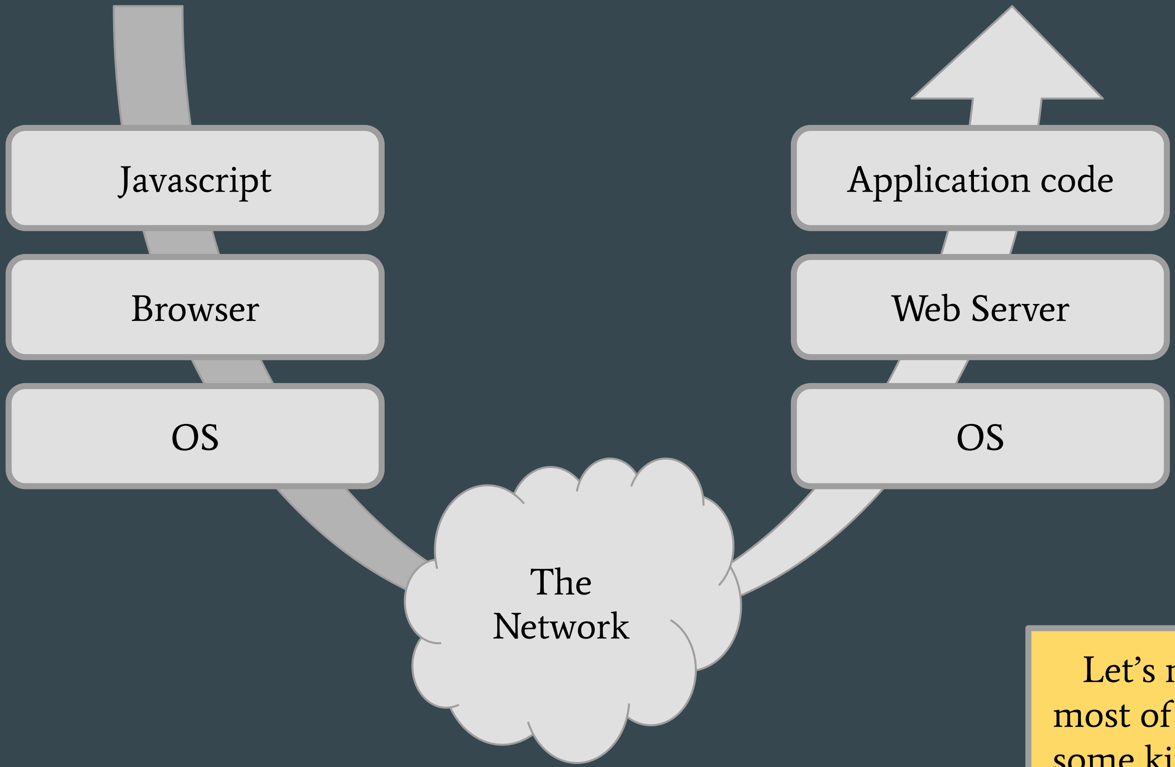
2-5 working days **Free**

Standard delivery

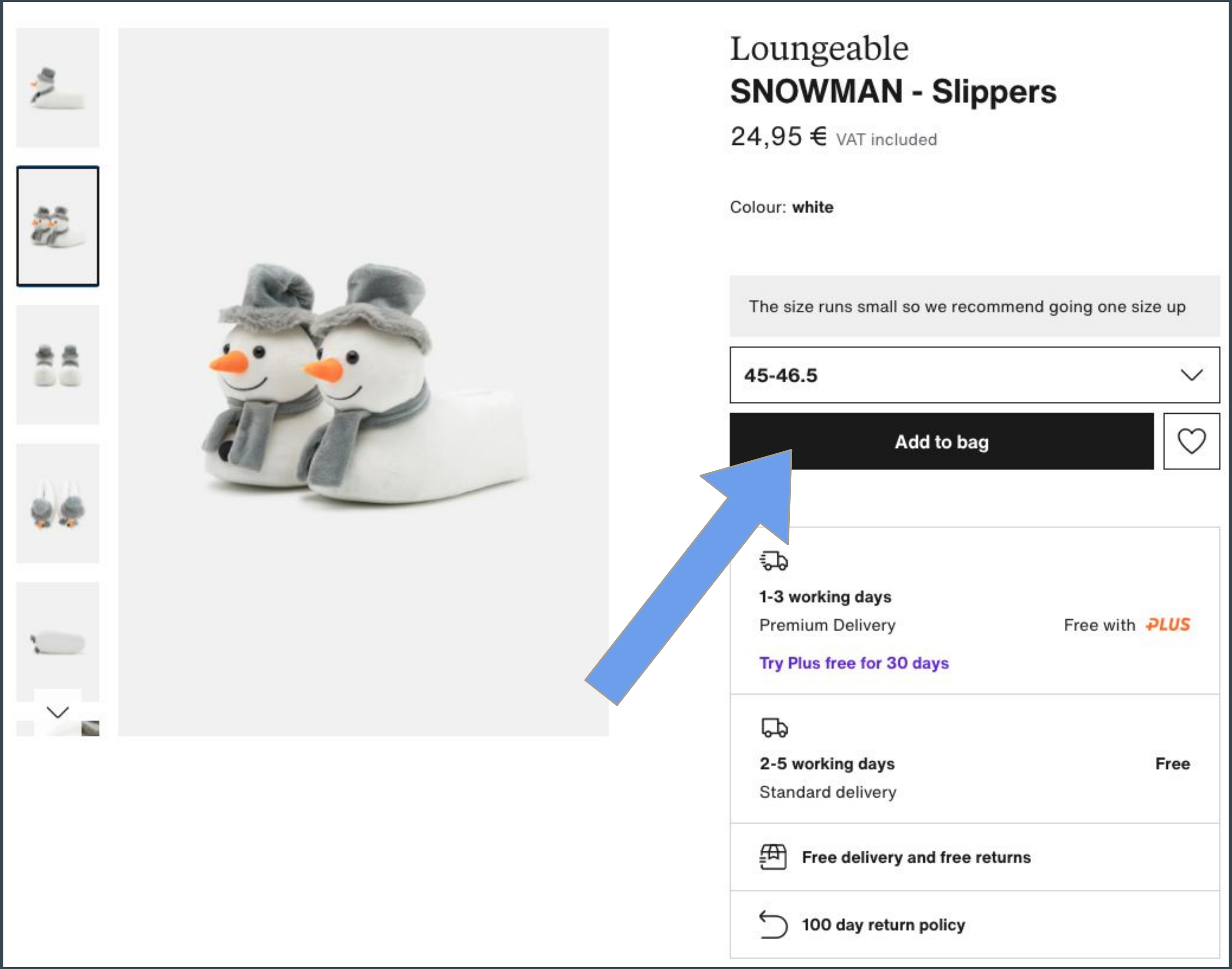
Free delivery and free returns

100 day return policy


Javascript
Browser
OS
The Network
Application code
Web Server
OS



Let's not forget that most of this will run on some kind of hardware!



Loungeable

SNOWMAN - Slippers

24,95 € VAT included

Colour: **white**

The size runs small so we recommend going one size up

45-46.5

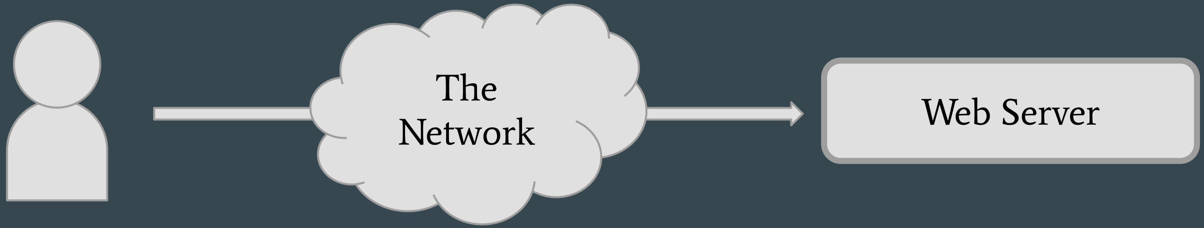
1-3 working days
Premium Delivery — Free with PLUS

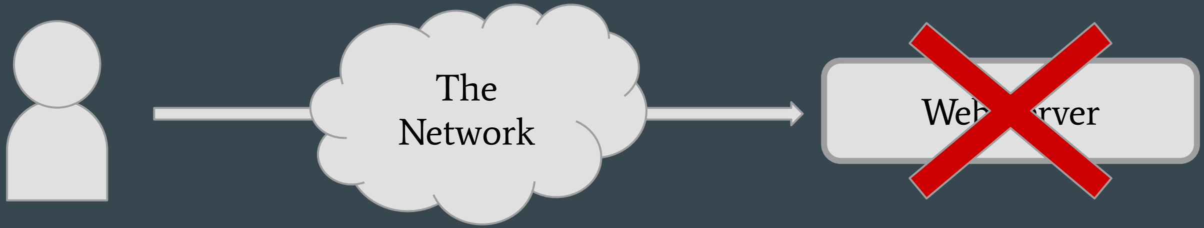
Try Plus free for 30 days

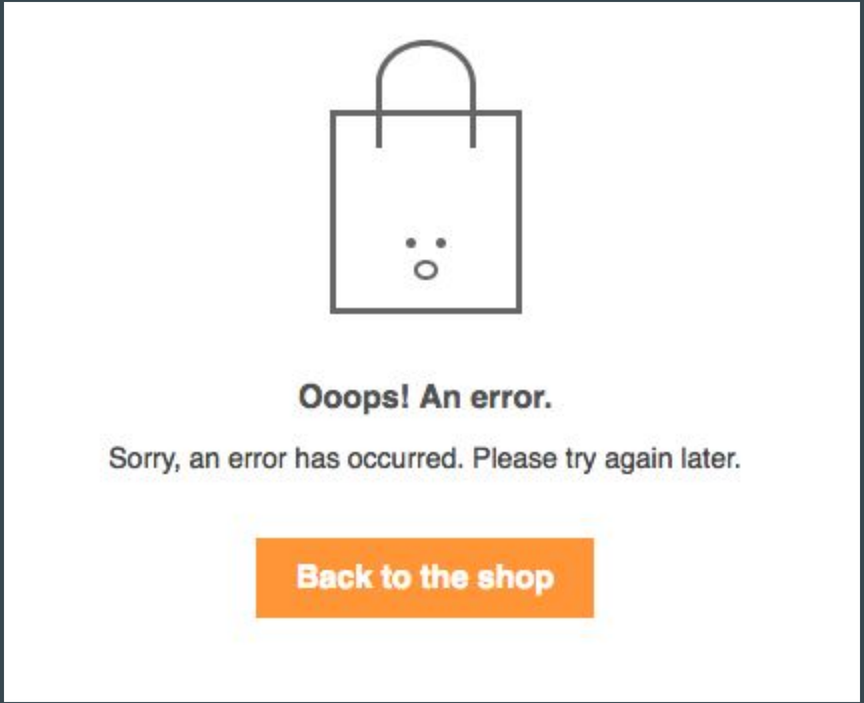
2-5 working days — **Free**
Standard delivery

Free delivery and free returns

100 day return policy

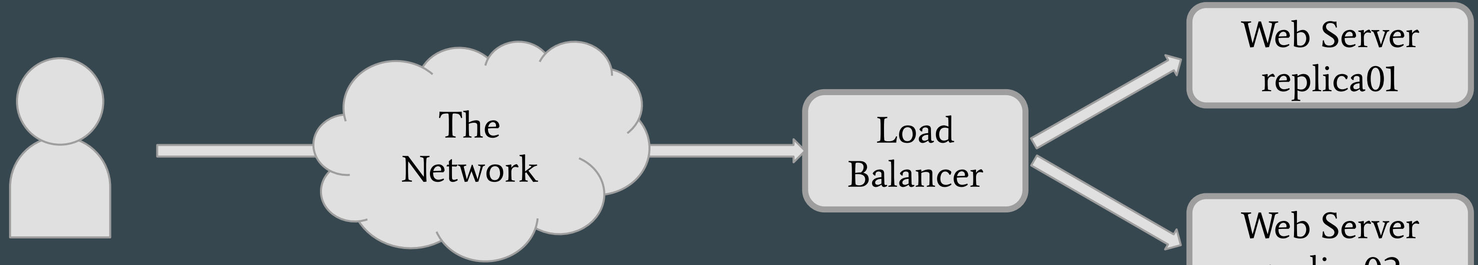

The Network
Web Server

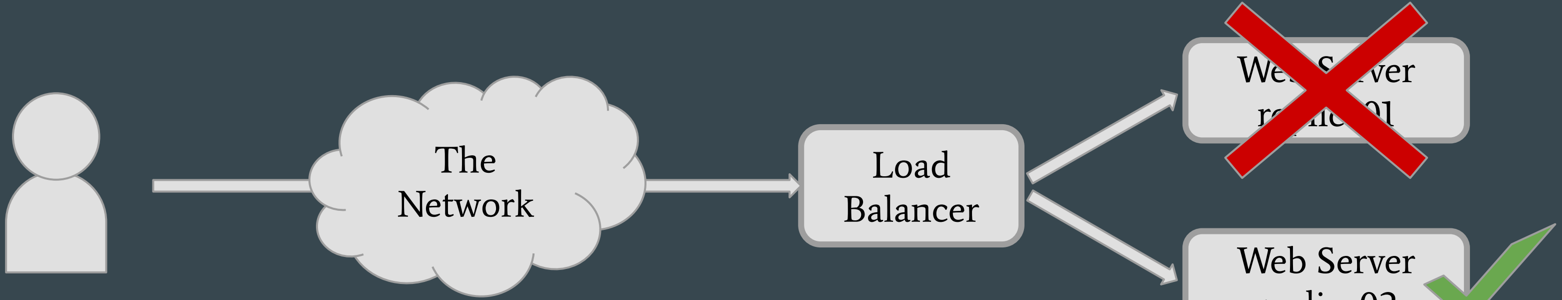

The Network
Web server

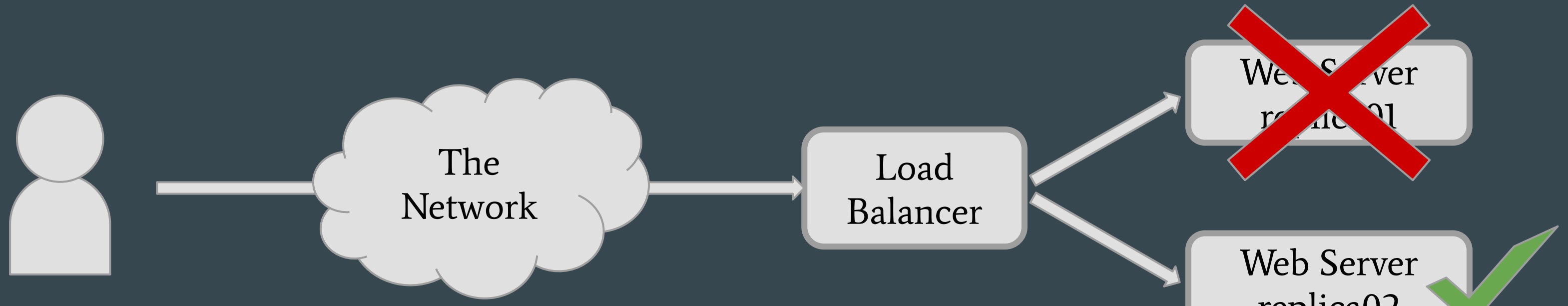


Ooops! An error.

Sorry, an error has occurred. Please try again later.


The Network
Load Balancer
Web Server replica01
Web Server replica02


The Network
Load Balancer
Web Server replica01
Web Server replica02

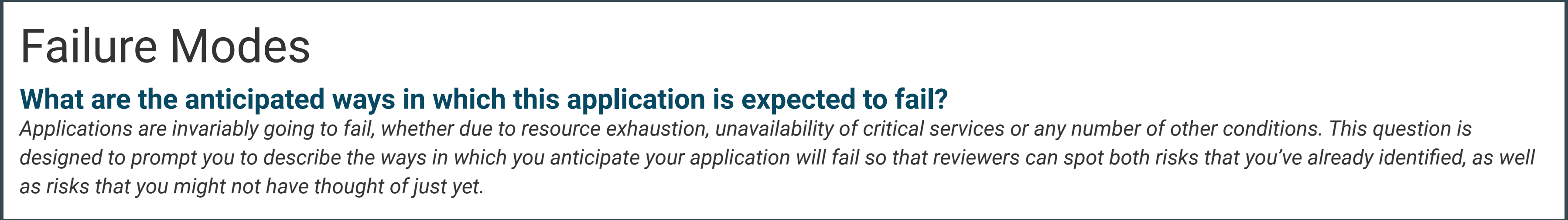


Through **redundancy** we achieved fault tolerance and the capacity to serve more requests

Production Readiness Reviews

A PRR is a high level review of a system before it goes into production

- Focuses on operational aspects
- A bit like code review, but on a higher level
- Originated at Google
- Example template from Grafana Labs
- A common practice at Zalando where we have the following question:

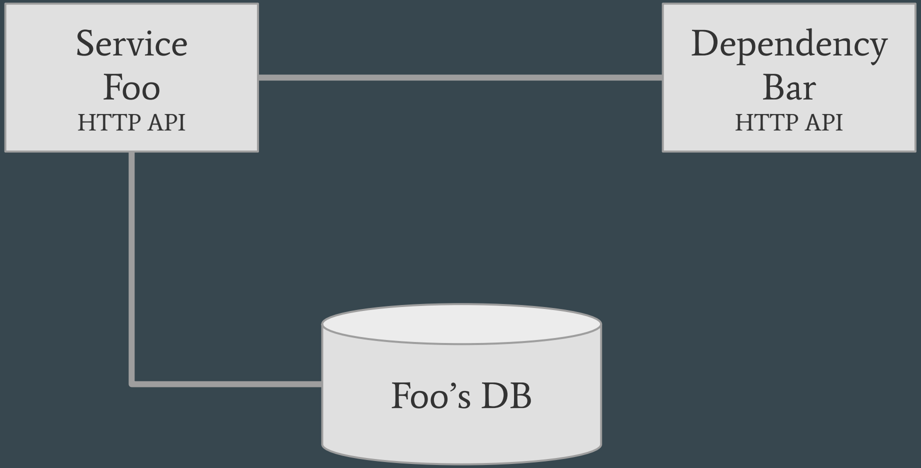


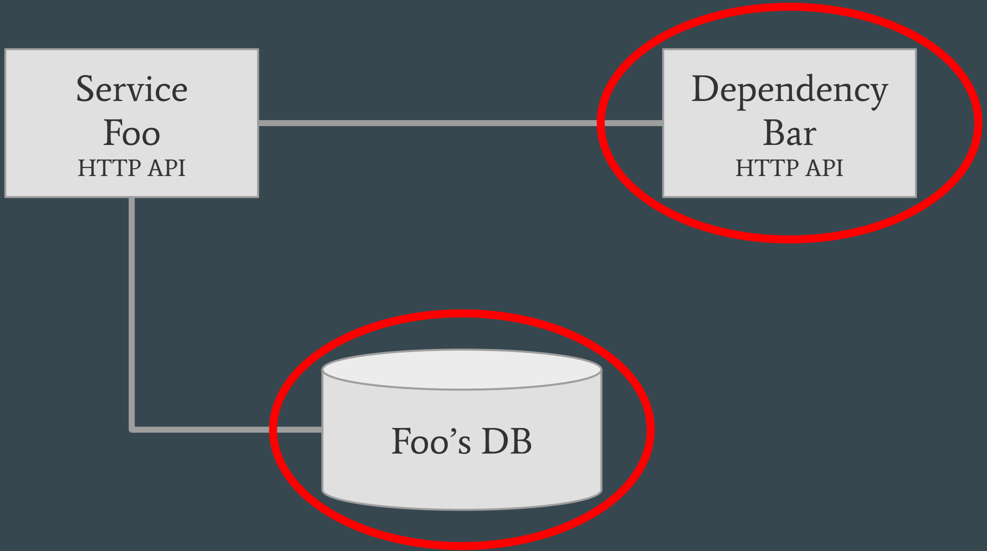
Failure Modes

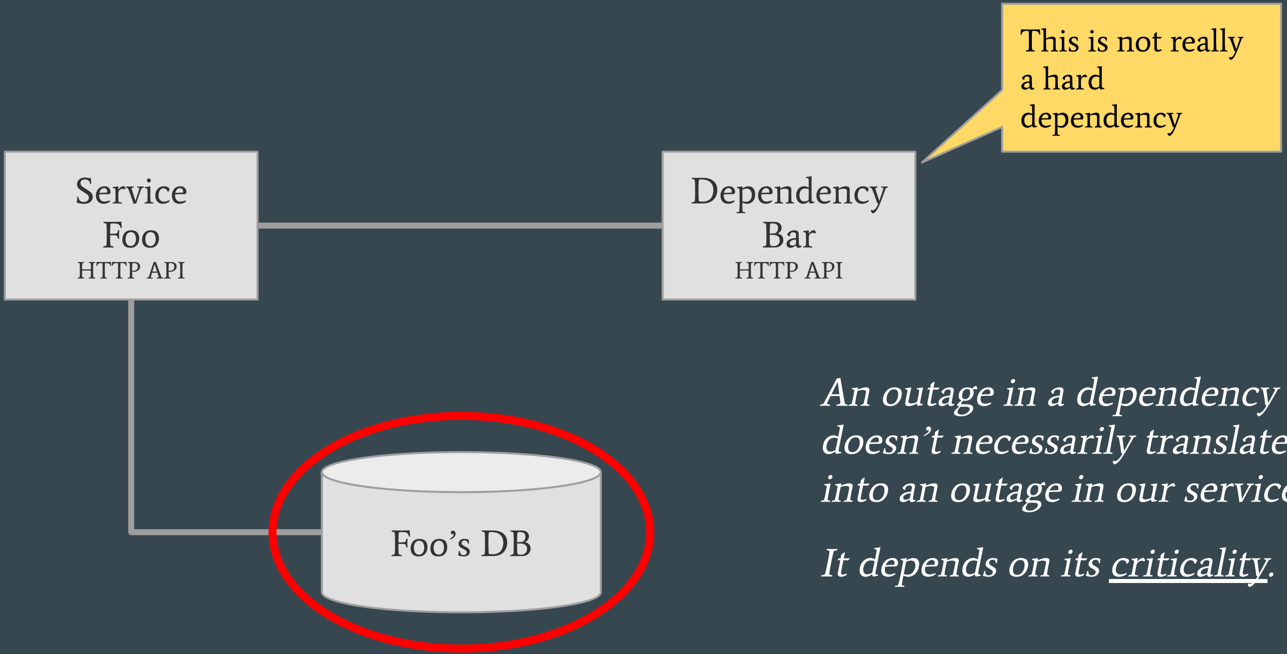
What are the anticipated ways in which this application is expected to fail?

Applications are invariably going to fail, whether due to resource exhaustion, unavailability of critical services or any number of other conditions. This question is designed to prompt you to describe the ways in which you anticipate your application will fail so that reviewers can spot both risks that you've already identified, as well as risks that you might not have thought of just yet.

Scenario 01

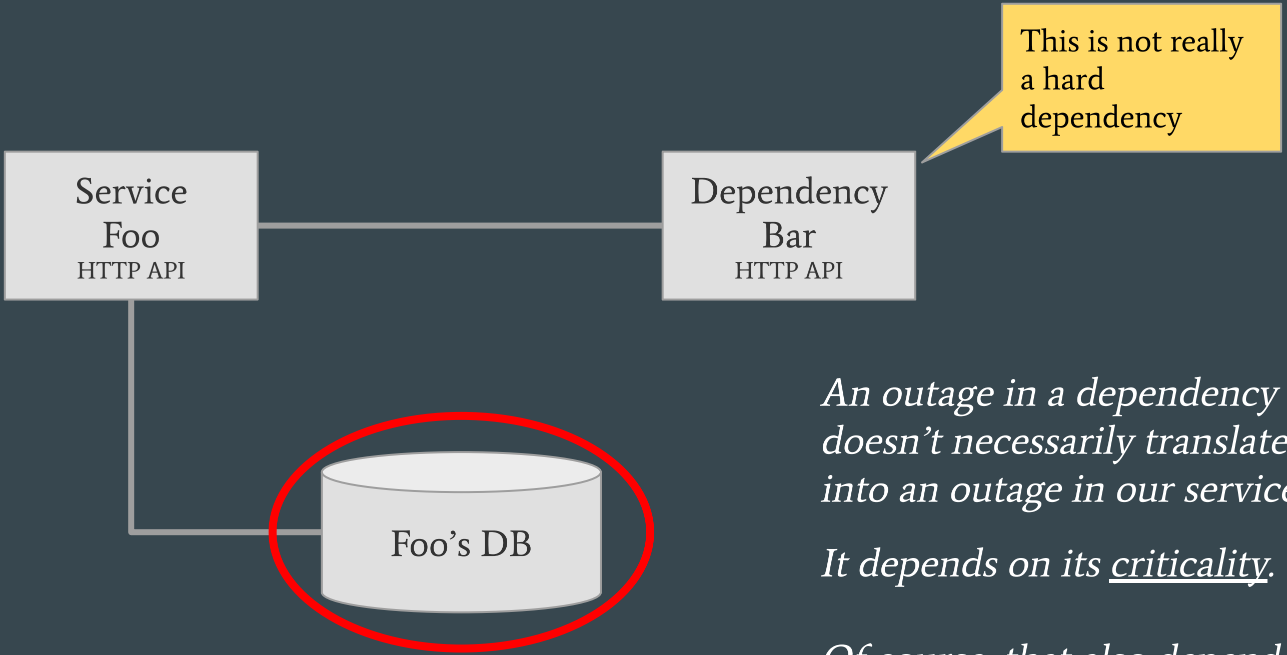

Service
Foo
HTTP API
Dependency
Bar
HTTP API
Foo’s DB


Service
Foo
HTTP API
Dependency
Bar
HTTP API
Foo's DB



An outage in a dependency doesn't necessarily translate into an outage in our service.

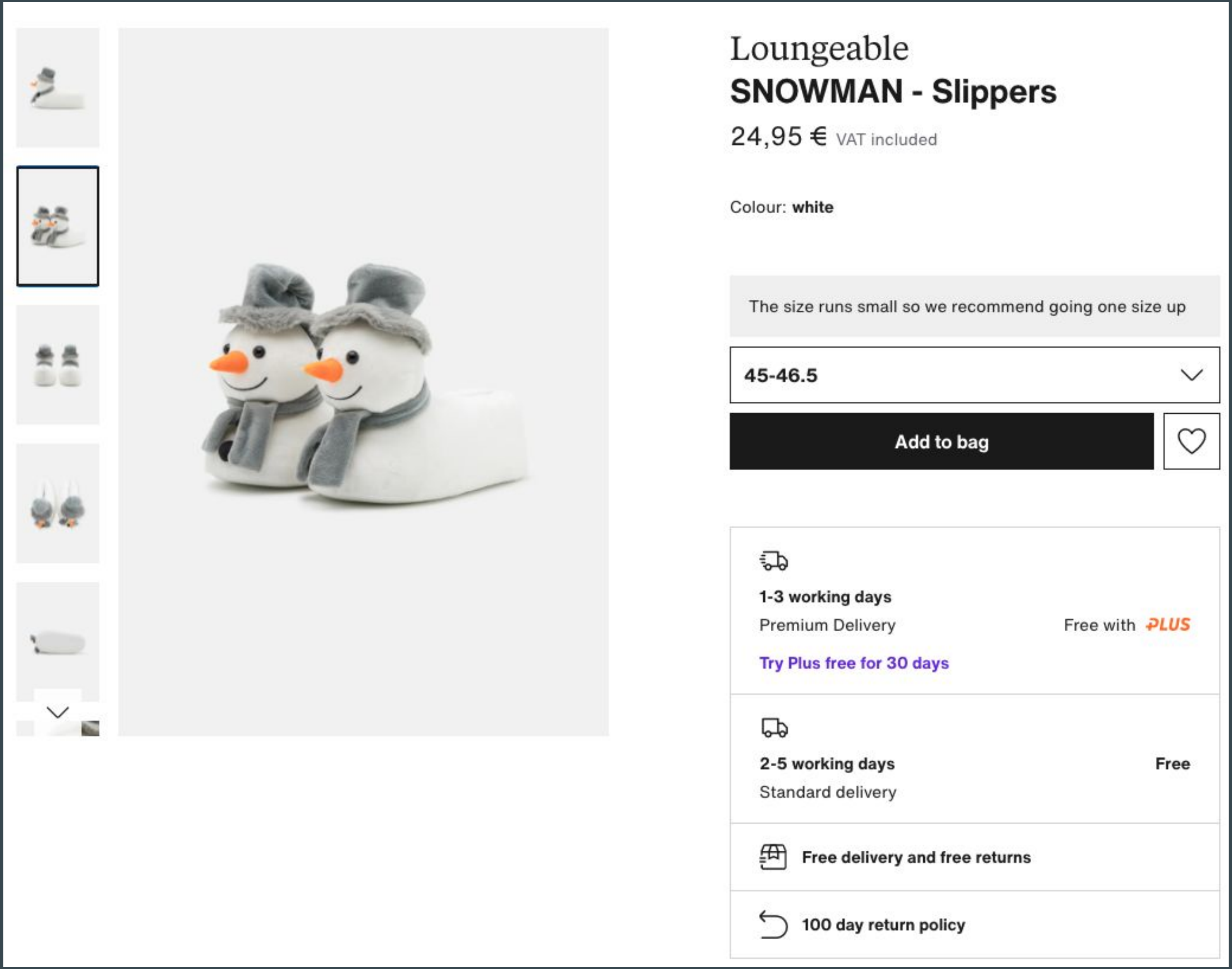
It depends on its <u>criticality</u>.



An outage in a dependency doesn't necessarily translate into an outage in our service.

It depends on its criticality.

Of course, that also depends on how we handle failures from that dependency.



Loungeable

SNOWMAN - Slippers

24,95 € VAT included

Colour: **white**

The size runs small so we recommend going one size up

45-46.5

Add to bag

1-3 working days

Premium Delivery — Free with PLUS

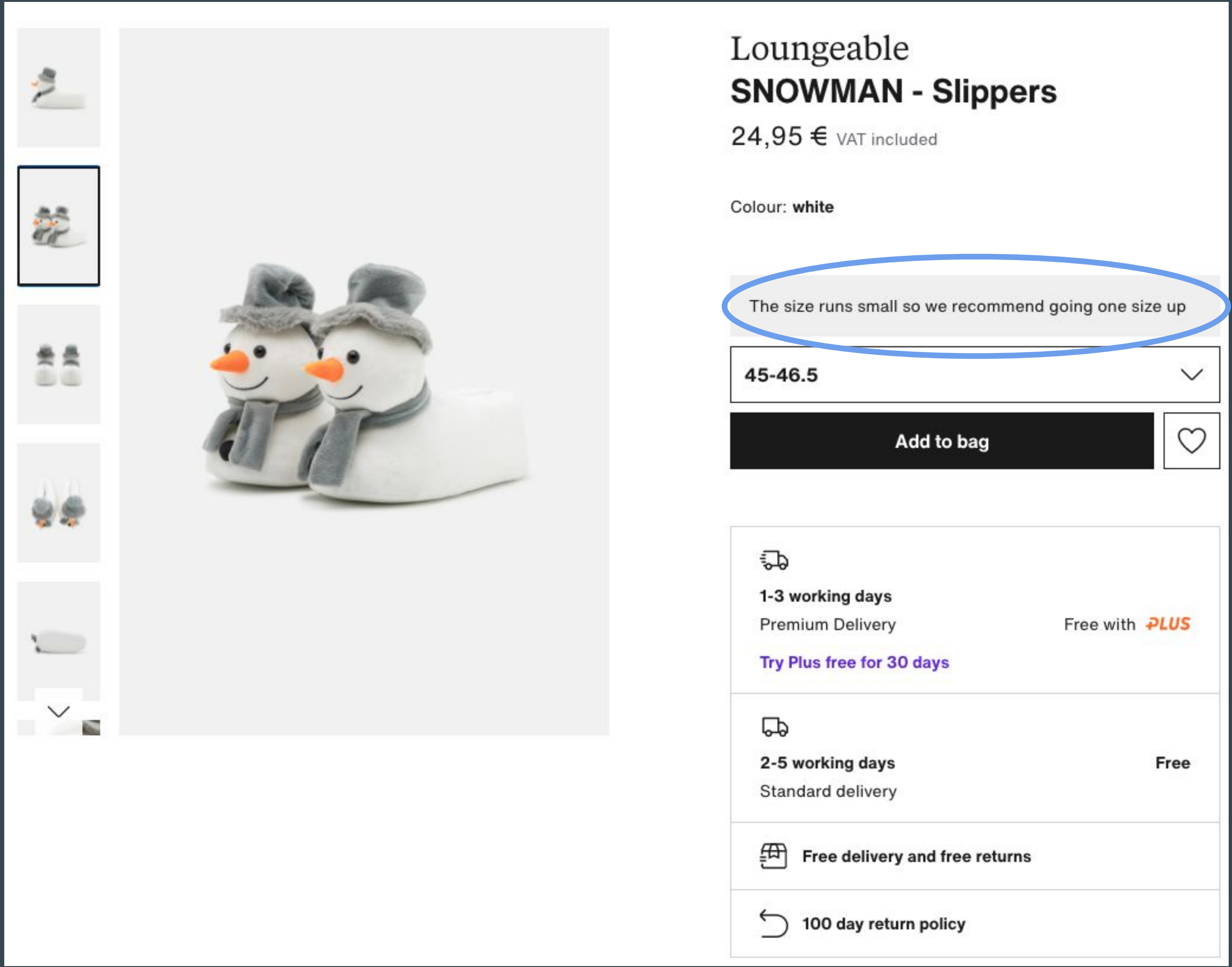
Try Plus free for 30 days

2-5 working days — **Free**

Standard delivery

Free delivery and free returns

100 day return policy



Loungeable

SNOWMAN - Slippers

24,95 € VAT included

Colour: **white**

The size runs small so we recommend going one size up

45-46.5

Add to bag

1-3 working days

Premium Delivery — Free with PLUS

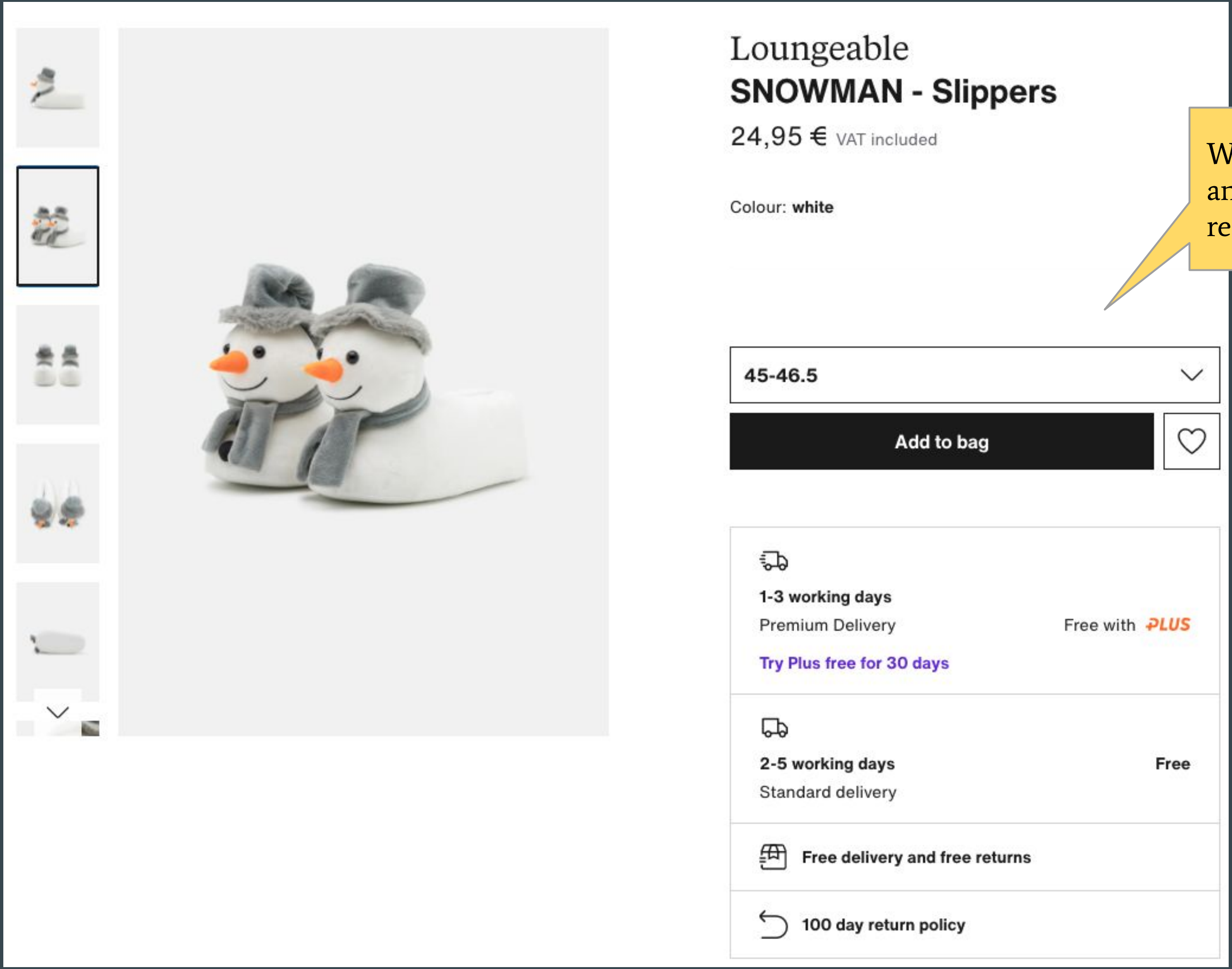
Try Plus free for 30 days

2-5 working days — **Free**

Standard delivery

Free delivery and free returns

100 day return policy



Loungeable
SNOWMAN - Slippers
24,95 € VAT included

Colour: **white**

45-46.5

1-3 working days
Premium Delivery — Free with PLUS
Try Plus free for 30 days

2-5 working days — **Free**
Standard delivery

Free delivery and free returns

100 day return policy

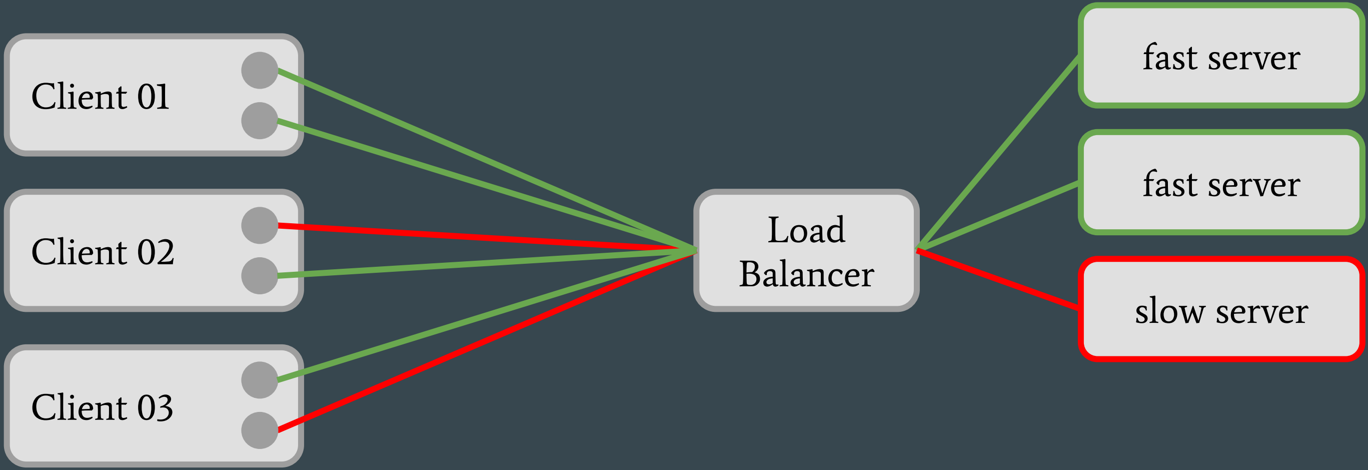
We used as a **fallback** an empty size recommendation.

Reliability patterns

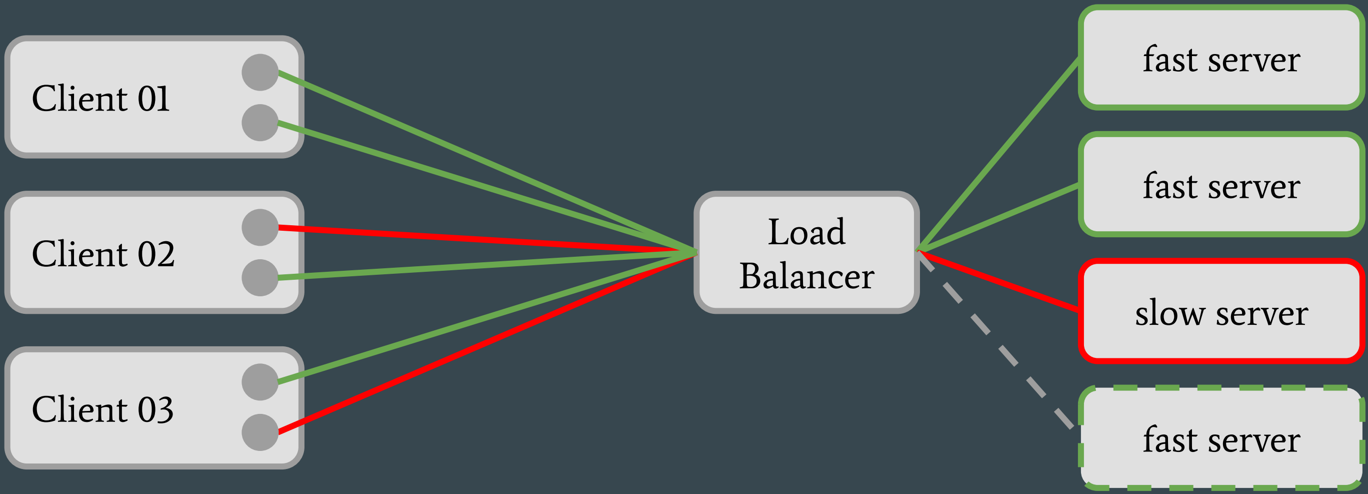
There are a few patterns that we can use in our code to be more resilient to failures in our dependencies:

- Timeouts
- Retries (with exponential backoff and jitter)
- Circuit breakers
- Fallbacks

Timeout


Client 01
Client 02
Client 03
Load Balancer
fast server
fast server
slow server

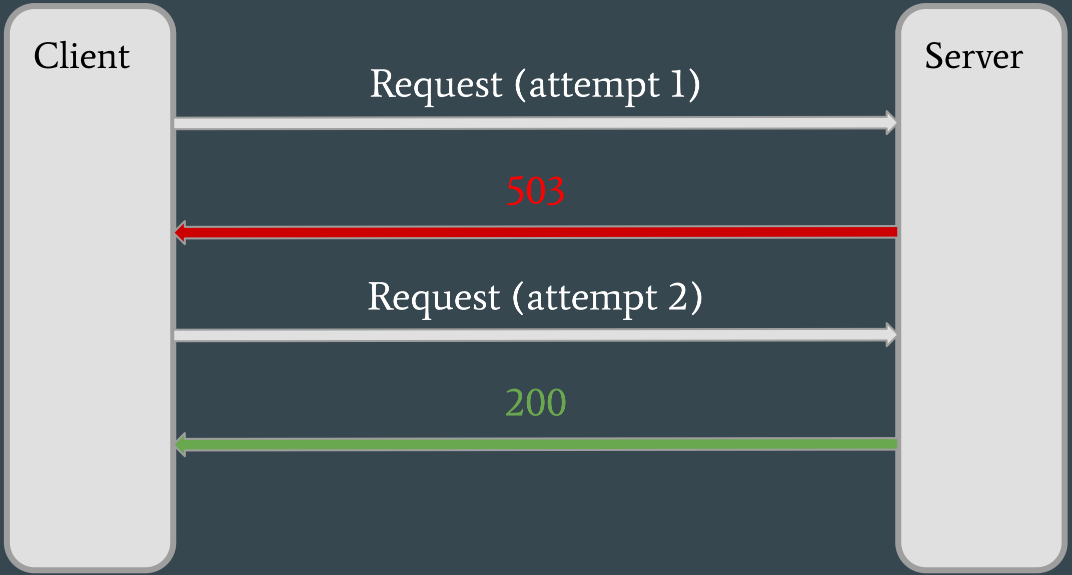
Timeout



Can also be set on the server side (terminating a request after some time to save the server’s resources), or when establishing a connection between server and client.

Retries

“If at first you don’t succeed, try try again”



Retries

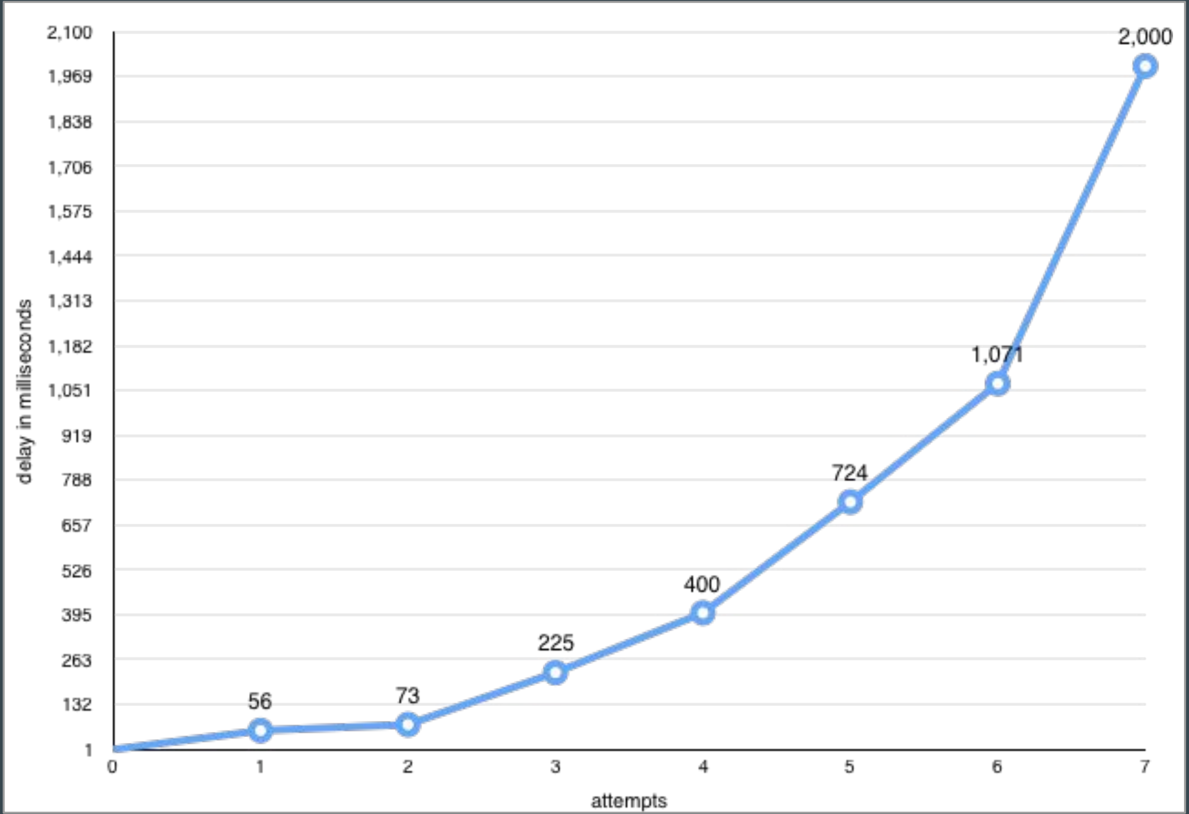
Retries are only supposed to be done in the case of transient errors (server overload, network errors; not for 4XX responses).

But not all transient errors are that ‘transient’.

Retries could actually make it worse, leading to a thundering herd problem.

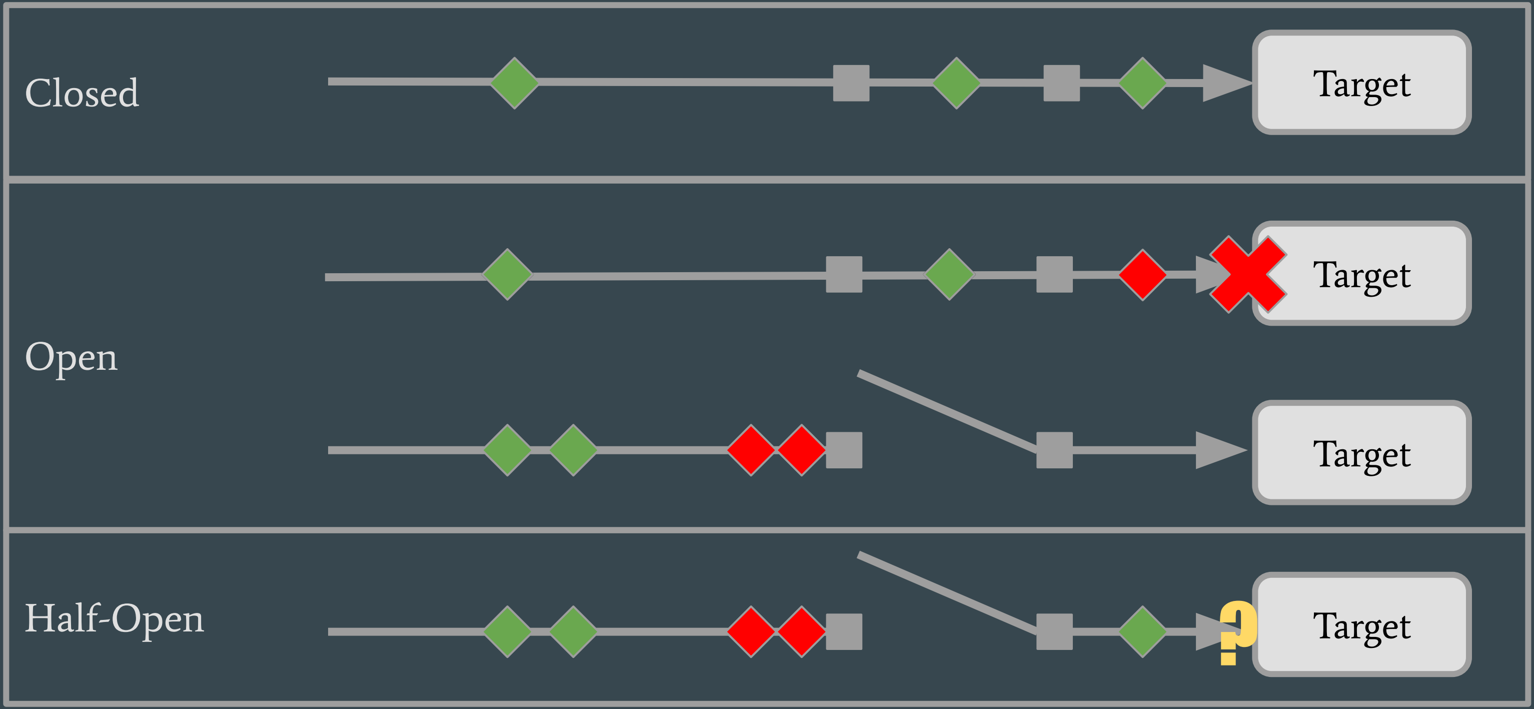
Also, beware of idempotency!

Retries with truncated exponential backoff and jitter



Circuit Breaker

Primarily designed to **help dependencies recover** from transient errors.



Fallback

“When all else fails...”

```
circuitBreaker.call((url) -> {
      for (int i = 1; i <= numRetries; i++) {
            try {
                  return restTemplate.getForObject(url, SizeRecommendation.class);
            } catch (RestClientException e) {
                  LOG.error("failed to get size recommendation", e);
                  if (i >= numRetries || isNonTransientFault(e)) {
                        throw e;
                  }
                  sleep(computeWaitTime(i, MAX_WAIT_TIME));
            }
      }
}).fallback(() -> "") // empty recommendation
```

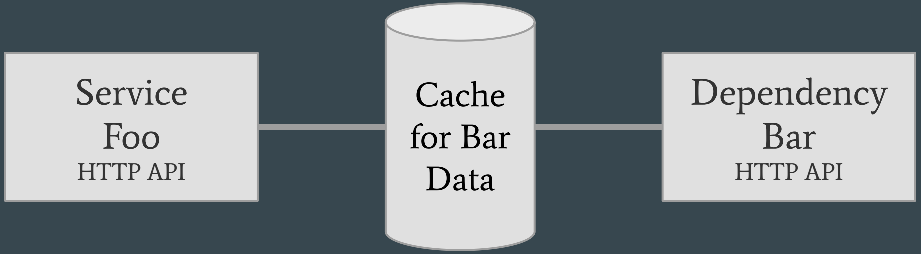
Implementing Reliability Patterns

You're likely to find a few libraries that implement one or more of these patterns in your language of choice.

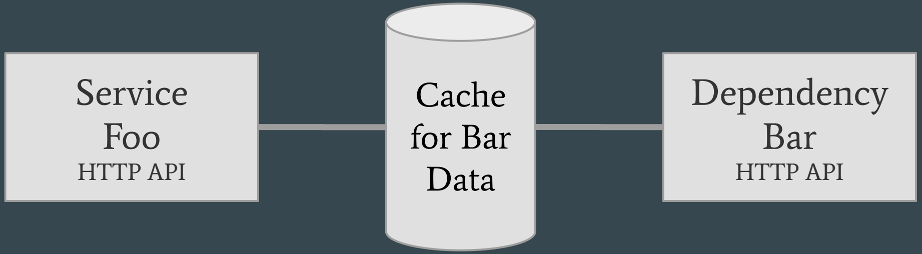
But do not underestimate the **importance** of properly configuring those patterns.

- Some libraries have as default infinite timeouts
- Are you sure you can retry all your requests?

Scenario 10

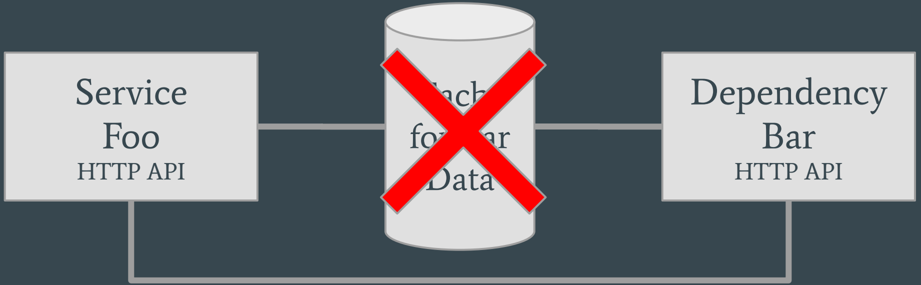


The Cache stores data from Bar for up to 12h. In the event of an outage in Bar, we can still serve data from the cache.



The Cache stores data from Bar for up to 12h. In the event of an outage in Bar, we can still serve data from the cache.

But what if it’s the Cache that breaks?



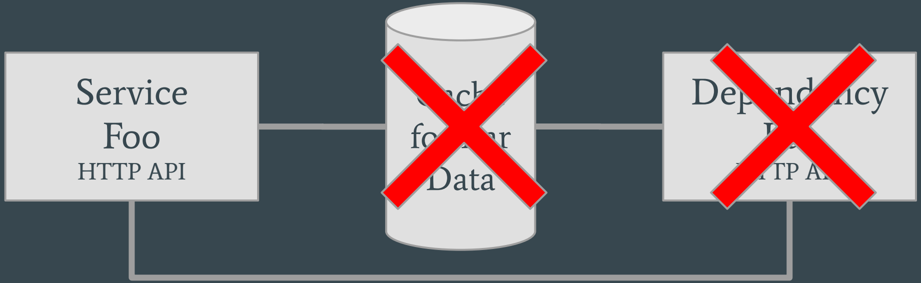
"That's fine. If the cache breaks I'll just send requests to Bar"

Foo will only be able to recover normal operations after the cache is back to its normal state. Until then it's possible that Foo will fall into an **overloaded** state.

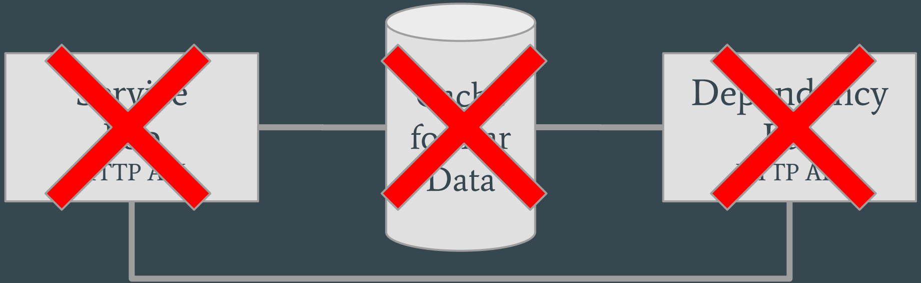
→ Can your service survive that, or will it break under pressure?

Why do you have the cache?

- "To serve requests faster" → Will going to Bar breach your latency SLO?
- "To reduce load on my dependency" → Will Bar be able to handle the extra traffic?

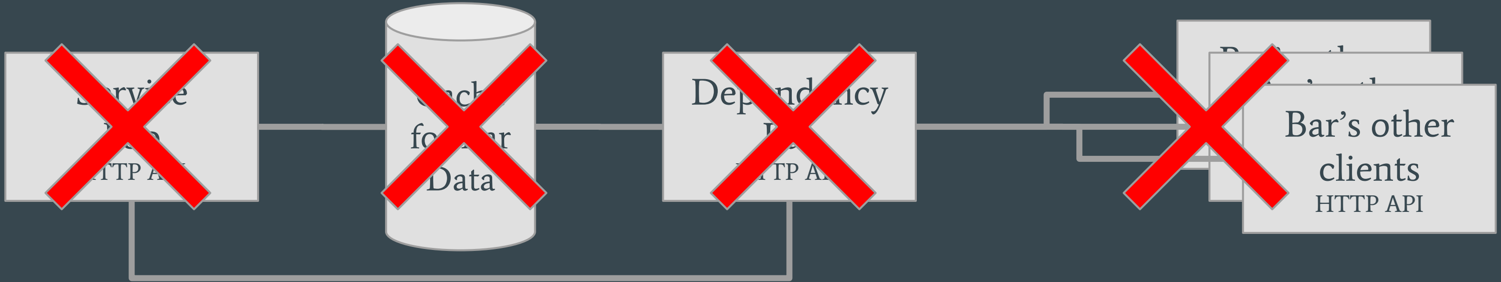


Let’s say Bar also breaks under pressure.



Let’s say Bar also breaks under pressure.

Your service will also start failing.

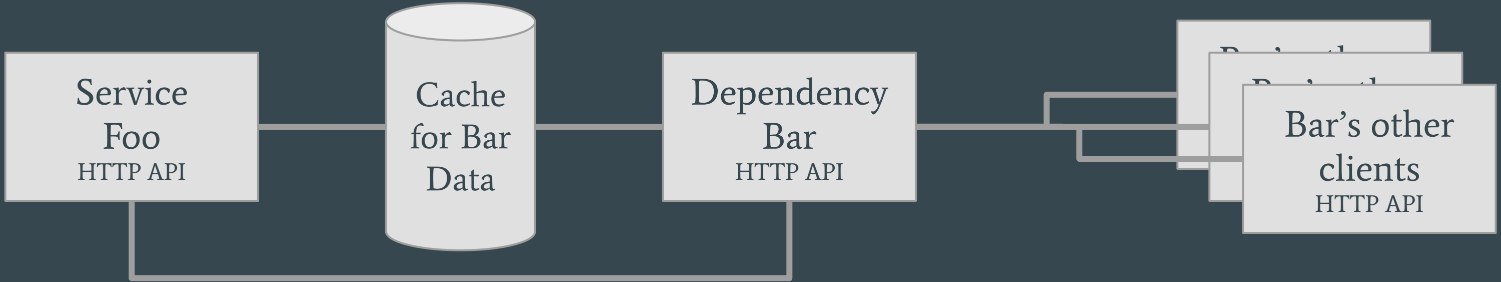


Let's say Bar also breaks under pressure.

Your service will also start failing.

And it should be safe to assume that the same will happen to its other clients.

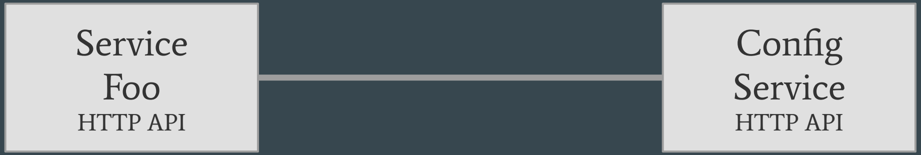
All of this because a cache failed. This tells us how critical the cache is.

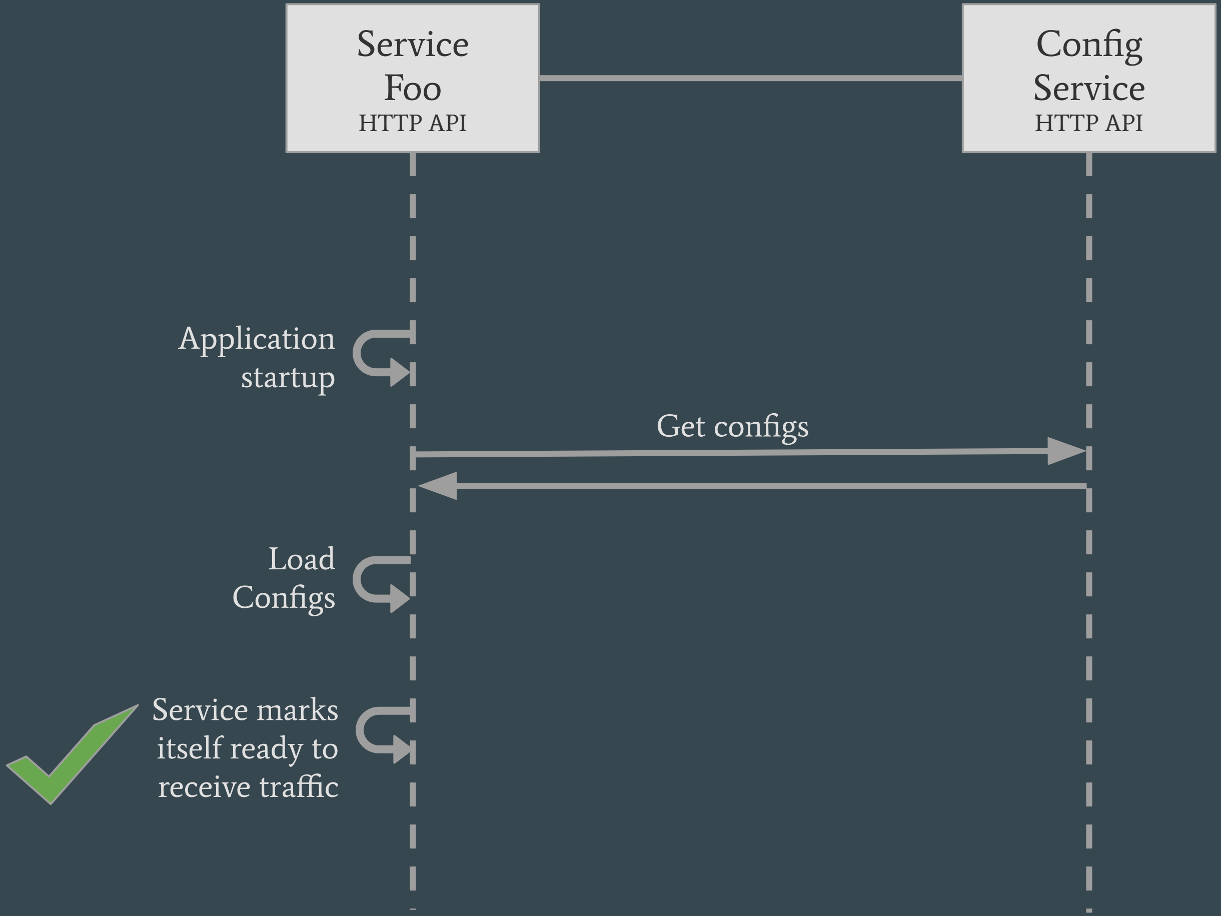


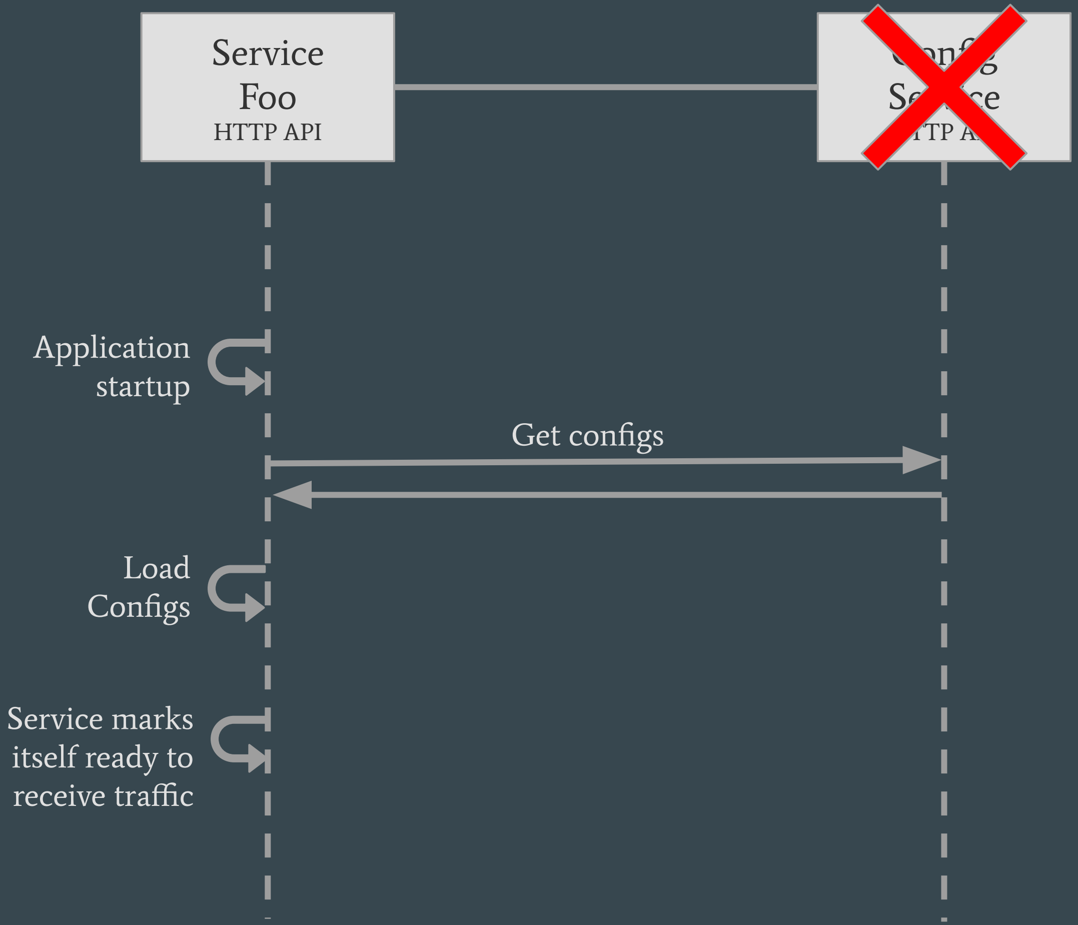
The Cache led to a cascading failure of services. But there are a few ways that Bar could have protected itself from that:

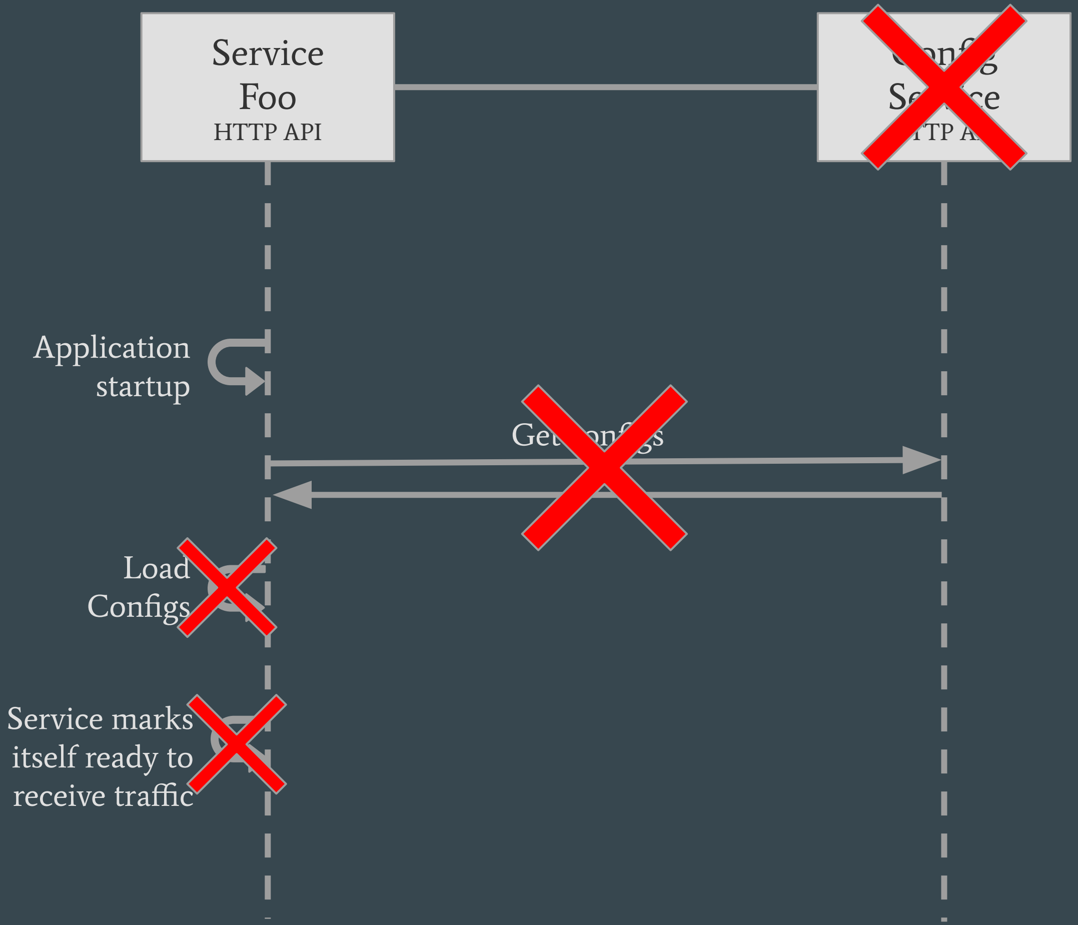
- Rate limiting - Prevents overload by limiting the requests a service can process. Could be global or per client.
- Bulkhead pattern - Isolates resources by a given criteria. By doing so, failures in one segment are isolated from other segments which can continue working normally
- Load shedding - Similar, to rate limiting, but can be more elaborate. Dropping requests for specific tasks, or for specific clients, while allowing others.

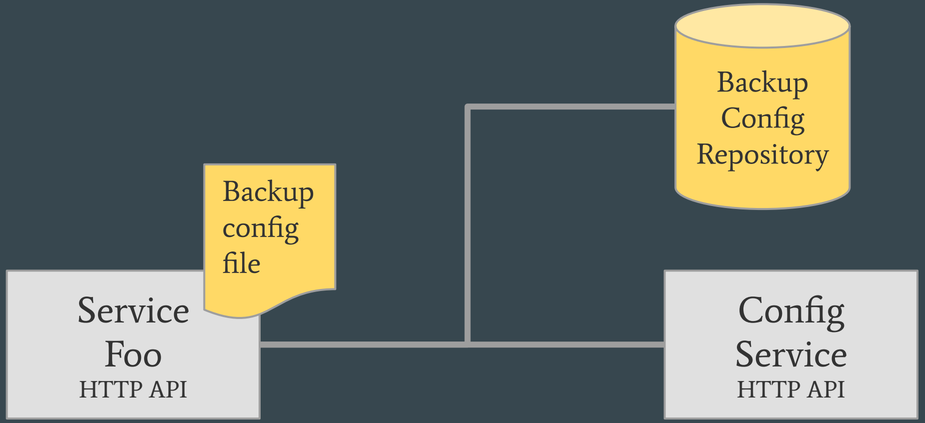
Scenario 11

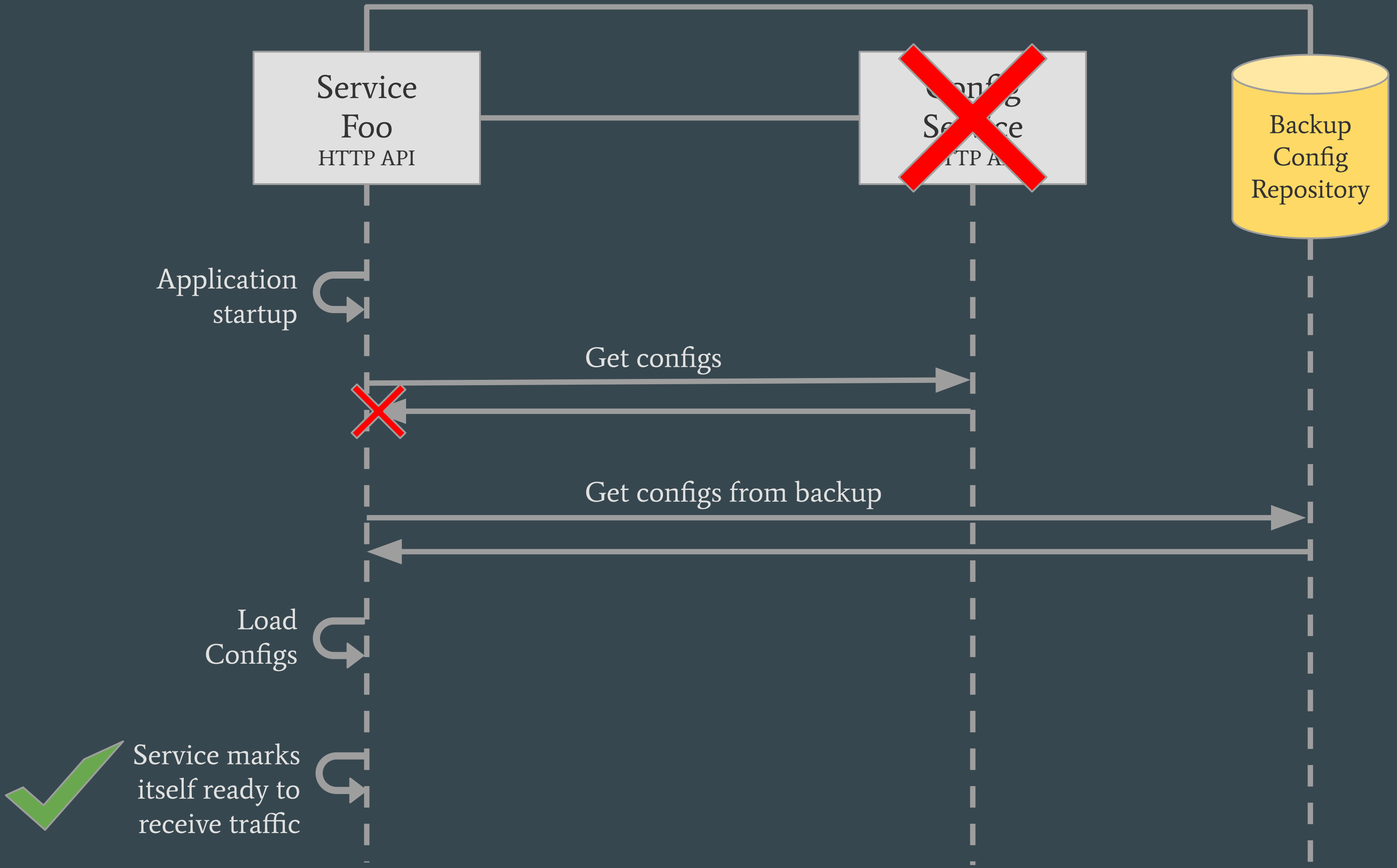

Service Foo
HTTP API
Config Service
HTTP API


Service Foo
HTTP API
Config Service
HTTP API
Application startup
Get configs
Load Configs
Service marks itself ready to receive traffic


Service Foo
HTTP API
Config Service
HTTP API
Application startup
Get configs
Load Configs
Service marks itself ready to receive traffic


Service
Foo
HTTP API
Config
Service
HTTP API
Application
startup
Get Configs
Load
Configs
Service marks
itself ready to
receive traffic


Backup
Config
Repository
Backup
config
file
Service
Foo
HTTP API
Config
Service
HTTP API


Service Foo
HTTP API
Config Service
HTTP API
Backup Config Repository
Application startup
Get configs
Get configs from backup
Load Configs
Service marks itself ready to receive traffic

A quick disclaimer

Every solution we saw adds **complexity**. Some, more than others.

Building resilience also means making the system more complex.

Understanding whether that added complexity is worthwhile boils down to two dimensions:

- **Impact** - “How many users will this affect?”
- **Criticality** - “How bad will it affect them?”

Always be sure to know **why** you are making the tradeoff of building reliability into your systems.

Learning from failure

Learning from failure

You can try really hard to build the most reliable system, but sometimes it’s just not your day.

Whenever something does break, be sure to **make the best** out of it!

Failures are **learning** opportunities, and a good way to get the most learnings is by conducting a **Post Mortem** on an outage in order to learn:

- What happened
- Why did it happen
- How did we respond to it
- What can we do to make sure that doesn’t happen again

Quick recap

Failures **will** happen!

- Design your system to be resilient
- Write your code to be resilient
- When something does break, take it as a learning opportunity

Q&A