MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Distributed Application Architecture Patterns

Master's Thesis

BC. JURAJ FIALA

Brno, Autumn 2024

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Distributed Application Architecture Patterns

Master's Thesis

BC. JURAJ FIALA

Advisor: RNDr. Martin Kuba, Ph.D.

Department of Computer Systems and Communications

Brno, Autumn 2024

MUNI
FI

Declaration

Hereby, I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during the elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, I used the following AI tools:

- GitHub Copilot for predictive writing
- Phind and ChatGPT for research, brainstorming and review
- Grammarly for proofreading
- Semantic Scholar for literature review

I declare that I used these tools in accordance with the principles of academic integrity. I reviewed the content and take full responsibility for it.

Bc. Juraj Fiala

Advisor: RNDr. Martin Kuba, Ph.D.

Acknowledgements

I would like to thank my advisor, doc. Ing. RNDr. Barbora Bühnová, Ph.D., for her remarkable insight and guidance, without which this thesis would never have been possible, and RNDr. Martin Kuba, Ph.D., for helping me take it over the finish line after her sudden passing. I would also like to thank my wife, Terézia, for her unwavering support and patience during long evenings and weekends, and my family and friends for their neverending encouragement.

Abstract

Software patterns and their visualisations are powerful learning tools. However, the existing literature on distributed application architecture patterns is often fragmented and incomplete, presenting an unnecessarily steep learning curve. This thesis proposes a methodology for identifying these patterns based on the current literature. It applies it to develop a structured online catalogue of the most prevalent patterns in the field, their applications and disadvantages, featuring accessible and consistent visualisations and illustrative examples.

Keywords

software architecture, distributed systems, patterns, design patterns, architectural patterns, microservices, cloud, containers, scalability, reliability, resilience, performance, security, consistency, distributed transactions, migration, compatibility

Contents

Introduction 1

1 Methodology 2
- 1.1 Scope of this work . . . 2
  - 1.1.1 Distributed application patterns . . . 2
  - 1.1.2 Application architecture patterns . . . 2
  - 1.1.3 Design patterns . . . 3
- 1.2 Existing resources . . . 3
  - 1.2.1 Classics . . . 3
  - 1.2.2 Service Oriented Architecture . . . 4
  - 1.2.3 Cloud computing . . . 4
  - 1.2.4 Microservices . . . 4
  - 1.2.5 Distributed systems . . . 5
  - 1.2.6 Wikipedia . . . 5
- 1.3 Selection process . . . 5
  - 1.3.1 Categorisation . . . 6
- 1.4 Pattern construction . . . 7

2 Design and implementation 9
- 2.1 Inspiration . . . 9
- 2.2 Visual style . . . 10
  - 2.2.1 Typography . . . 10
  - 2.2.2 Colour palette . . . 11
- 2.3 Technologies . . . 12
  - 2.3.1 Diagrams . . . 12
  - 2.3.2 Website . . . 12
- 2.4 Visual language . . . 13
- 2.5 Design process . . . 14

3 Overview 15
- 3.1 Prerequisites . . . 15
  - 3.1.1 Fallacies of distributed computing . . . 16
  - 3.1.2 CAP theorem . . . 16
- 3.2 Pattern categories . . . 17
- 3.3 Conventions used . . . 18
  - 3.3.1 Diagrams . . . 18
  - 3.3.2 Terminology . . . 19

3.4 Example application 19

4 Communication Patterns 20
4.1 Gateway Routing 21
Abstract service locations from the clients
4.1.7 API Gateway 23
4.2 Publisher–Subscriber (Pub–Sub, Topics) 24
Asynchronous one-to-many communication
4.2.7 Message Broker 26
4.3 Asynchronous Request–Reply 26
Asynchronous two-way communication

5 Decomposition Patterns 29
5.1 Sidecar 30
Language-agnostic, locally-running supporting components
5.2 Ambassador 32
Handle network communication on behalf of a service
5.3 Offload to Gateway 34
Move common functionality to a gateway to offload backend services
5.4 Backend for Frontend (BFF) 36
Reduce backend complexity by specialising for each frontend

6 Scalability Patterns 39
6.1 Competing Consumers & Load Balancer 40
Continuous parallel request processing
6.2 Partitioning (Sharding) 42
Scale out by separating tasks or data into logical partitions
6.2.7 Deployment Stamps 45
6.3 Scatter–Gather 45
Asynchronously distribute workloads and aggregate results
6.4 Externalised Configuration 47
Centralised configuration management

7 Resilience and Reliability Patterns 50
7.1 Bulkheads 51
Use logical partitions to isolate failures
7.2 Queue-Based Load Leveling 53
Use a messaging queue to manage and cope with peaks in demand
7.3 Retry 55
Do not fail because of transient errors
7.4 Health Monitoring 57

Proactively check and react to service failures

7.5 Rate Limiting 60
Control the rate of incoming requests to prevent overload or starvation

7.6 Leader and Followers 62
Decentrally appoint a replaceable group leader

8 Performance and Latency Patterns 65

8.1 Circuit Breaker 66
Isolate failure with controlled recovery

8.2 Colocate 69
Improve network reliability by decreasing the distance between services with high coupling

8.3 Aggregating Gateway 70
Aggregate multiple service requests into a single client response

8.4 Claim Check 72
Reduce message sizes by storing large payloads externally

8.5 Command and Query Responsibility Segregation (CQRS) . 74
Isolate data store reads and writes to prevent contention

9 Security Patterns 78

9.1 Identity provider & Federated Identity 78
Centralise authentication to reduce the attack surface

9.2 Gatekeeper (Service Firewall) 81
Protect services by validating and sanitising incoming requests in a limited environment

10 Consistency Patterns 83

10.1 Transaction-Based Processor 84
Use transactions to encapsulate atomic processes

10.2 Transactional Outbox 85
Atomically send a message and update the database

10.3 Saga 87
Sacrifice isolation for increased availability during a distributed transaction

10.4 Choreography-Based Sagas 90
Distribute responsibility for a saga across participating services

10.5 Orchestration-Based Sagas 92
Centralise responsibility for a saga in a single service

10.6 Event Sourcing 93
Store data as a sequence of changes to preserve history

11 Migration and Compatibility Patterns 96
11.1 Anti-Corruption Layer (ACL) 96
Mediate between modern and legacy systems
11.2 Incremental Replacement (Strangler Fig) 98
Replace a legacy system step-by-step
11.3 Messaging Bridge 101
Connect two services with incompatible messaging middleware

Conclusion 103

Bibliography 104

A Digital attachments 125

Introduction

Over the past thirty years, patterns have become an undisputable part of software development and architecture. They capture knowledge about common problems and their solutions, help communicate, learn, and ultimately avoid reinventing the wheel. It is all the more surprising that when it comes to distributed systems, the literature becomes fragmented and incomplete, with a steep learning curve.

This work aims to fill that gap by creating a complete, contemporary and unopinionated catalogue of distributed application architecture patterns with an accompanying website, aimed primarily at making the patterns more accessible to newcomers to the field.

Chapter 1 examines the existing literature and defines a suitable scope for the patterns based on its deficiencies. It then proposes a methodology for identifying, classifying and documenting the patterns. Chapter 2 describes the process behind creating the diagrams and website and their visual language.

Chapter 3 then provides the theoretical background and additional information needed to understand and navigate the patterns.

The 33 identified patterns are then presented in the following chapters by their corresponding categories: communication (4), decomposition (5), scalability (6), resilience and reliability (7), performance and latency (8), security (9), consistency (10), and migration and compatibility (11).

The complete pattern catalogue is also available on the following website: `https://jurf.github.io/daap/`.

1 Methodology

This chapter describes the methodology used for constructing the catalogue in this work. Section 1.1 defines the scope of the work, § 1.2 reviews existing literature, § 1.3 describes the pattern selection process and name resolution, § 1.3.1 details their categorisation and § 1.4 outlines the chosen structure of the individual patterns.

1.1 Scope of this work

Defining the scope coincided with the literature review to prevent unnecessary overlap with existing works. This work considers the following criteria for choosing patterns.

1.1.1 Distributed application patterns

Patterns have to *exclusively pertain to distributed systems.* Each pattern has to involve at least two discrete components. This work does not strive to rebrand existing monolithic system patterns unless they exhibit unique characteristics when applied to distributed systems, such as **Event Sourcing** (see § 10.6) or **CQRS** (see § 8.5), and not patterns such as **Cache-Aside**, which is a technique universal to any system, or **Index Table**, which is a database design pattern.

1.1.2 Application architecture patterns

Patterns have to *pertain to application architecture,* i.e. not algorithms (such as specific leader election algorithms), data management principles, or specific technologies (such as cloud services, databases or message brokers). They should be future-proof and without a vendor bias.

Defining software architecture is hard [1, 2, p. 3, 3, p. 527] – this work settles on the heuristic that the patterns have to have a *visualisable structure* to be considered architectural.

As such, this work does not aim to cover messaging patterns in general, which were already covered by Hohpe et al. [4] and stood the test of time well, data replication patterns, which were extensively covered by Joshi [5], or the basics of information security, such as asymmetric encryption or certificates.

1.1.3 Design patterns

The patterns have to be *design patterns*, i.e. not strategies, tactics, overall architectural styles (such as SOA or microservices), methodologies or anti-patterns. Architectural styles were already extensively covered by Richards et al. [2] and Klimešová [6].

1.2 Existing resources

The scope defined in § 1.1 creates an interesting gap that no existing source seems to comprehensively cover.

This section provides what the author hopes to be a comprehensive review of existing literature on distributed systems patterns. Only major works containing constructed pattern sets or catalogues were considered to ensure relevancy and consistency in selection.

Methods employed include online searches, querying academic databases, and reviewing the works' bibliographies. Unfortunately, many non-academic works fail to provide a comprehensive list of references, so almost every pattern required additional research to find the original source.

1.2.1 Classics

Martin Fowler's *Patterns of Enterprise Application Architecture* [7] is a natural place to start, but it pertains mostly to layered monolithic systems (the chapter on distribution patterns only contains two patterns) and, importantly – as Fowler notes [8] – completely omits asynchronous messaging.

This led to the creation of Gregor Hohpe and Bobby Woolf's *Enterprise Integration Patterns* (2003) [4]. However, this, in turn, only covers messaging patterns and not the architecture as a whole.

The *Pattern-Oriented Software Architecture* (POSA) series defined a multitude of patterns during its run. Frank Buschmann et al.'s *Volume 2: Patterns for Concurrent and Networked Objects* (2000) [9] and *Volume 4: A Pattern Language for Distributed Computing* (2007) are most relevant to distributed systems, with the latter being the last complete book on this topic. This work considers all patterns later confirmed by *Volume 5* (2007) [10], which was, unfortunately, the last in the series.

Michael Nygard's influential *Release It!* (2007) [11] focuses on stability patterns, a few of which are architectural. This work uses the updated 2018 second edition [12].

1.2.2 Service Oriented Architecture

SOA, although now declined in popularity [2, p. 242], was an important milestone in distributed computing, and it brought Arnon Rotem-Gal-Oz's *SOA Patterns* (2012) [13]. Despite the ageing technology references and vocabulary, it still holds up remarkably well, with quite a large overlap with later works, but often uses different names for the same patterns.

1.2.3 Cloud computing

Due to the rise of cloud computing in recent years, several works have been published documenting patterns used for architecting cloud applications. The first was Bill Wilder's *Cloud Architecture Patterns* (2012) [14], documenting a handful of patterns in depth. Fehling et al.'s *Cloud Computing Patterns* (2014) [15] was one of the inspirations for this work's structure and visual style and has a whole chapter dedicated to cloud application architecture patterns. It also has an accompanying website[1].

The major cloud providers have also published their own pattern catalogues. The first was Alex Homer et al.'s *Cloud Design Patterns* (2014) [16], which later evolved into the open-source *Azure Architecture Center* series on *Cloud Design Patterns* [17]. It provides the most comprehensive list and often tops the search results for the given pattern. However, it suffers from inconsistency both in the selection of patterns[2] and in structure, and is prone to wrapping Azure products as patterns. Amazon Web Services also hosts *Cloud design patterns, architectures, and implementations*, a catalogue by Anitha Deenadayalan [18], which covers fewer patterns but more consistently.

However, while many cloud architecture patterns are applicable to any kind of distributed system, many require a cloud environment to be fully utilised.

1.2.4 Microservices

The advent of microservices brought Sam Newman's *Building Microservices* (2015) [19] and the expanded 2021 second edition [3], which offer a comprehensive guide but buries the patterns in the text. Chris Richardson's *Microservices Patterns* (2018) [20] fills this gap with a complete catalogue but lacks structure or overview diagrams. This is offset somewhat by the accompanying website[3] [21], but at the time of writing, it is still under construction, and several patterns only have a skeletal structure.

1. https://www.cloudcomputingpatterns.org/
2. E.g. it includes the Index Table – a database pattern – in the list
3. https://microservices.io/

1.2.5 Distributed systems

In the last few years, several works have been published that focus on distributed systems in general. Brendan Burns' *Designing Distributed Systems* (2018) [22] is closest to the scope of this work (with some focus also on containers), but similarly to Richardson, opts for a deep dive instead of a catalogue. While Mark Richards and Neal Ford's *Fundamentals of Software Architecture* (2020) [2] covers a few well-known patterns, it takes a "new approach" and only mentions them in passing. Unmesh Joshi's *Patterns of Distributed Systems* (2023) [5] is the most recent and most in-depth work, but focuses more on the technical aspects of constructing distributed systems and infrastructure, with a focus on data replication instead of the overall architecture.

There is also a work-in-progress sequel to EIP, *Conversation Patterns*, by Hohpe [23], with a last update in 2017. Since it is not finished, is not considered for selection of patterns, but it already includes useful patterns that are referenced when appropriate.

1.2.6 Wikipedia

Wikipedia has a wide coverage of common software patterns and, due to its freedom of access, is often linked to, especially with older patterns, such as the *Gang of Four* patterns [24], where it is quite comprehensive [25]. However, when it comes to distributed systems, this coverage is often sporadic and lacklustre. This is a shame, as Wikipedia's encyclopedic nature and focus on secondary sources would make it a valuable resource for this work. Still, due to its open access and the educational nature of this work, it is referenced as an additional source wherever applicable.

1.3 Selection process

This work gathers patterns from the sources defined in § 1.2 and selects those that meet the criteria defined in § 1.1. As patterns have to be "rooted in practice" [7, p. 10], it requires contemporary secondary sources to validate the patterns' ongoing relevance and applicability, as it does not strive to simply rephrase patterns.

Such a cutoff is inherently subjective, but necessary due to the change in landscape observable in the literature. This work settles on defining "contemporary" as anything that was released after containers and cloud – two technologies that shaped the landscape the most – became prevalent.

If a pattern is omitted in a later edition of a book, this work also omits it.

Some authors define pattern boundaries differently. This work tries to handle these situations by using the highest fidelity possible. On the other hand, if different authors define different patterns that result in the same structure, this work discusses their differences under one name instead.

These kinds of conflicts require name resolution. This work balances the following pillars.

- **Familiarity.** The name should match the original as close as possible. The reason is twofold: to facilitate knowledge carryover and to enable easy cross-referencing. If a different name is chosen, the original name is mentioned in parentheses.
- **Descriptiveness and intent.** Each name should prime the reader for the pattern's goals and means as comprehensively as possible. It should not require additional context or knowledge to be understandable [26, p. 26].
- **Uniqueness.** The pattern name should not clash with other existing patterns used in different contexts to prevent confusion.

Unfortunately, these pillars are often in conflict and require a subjective decision. This is documented alongside the pattern.

1.3.1 Categorisation

Most of the literature reviewed in § 1.2 categorises patterns by what they are, but this requires the reader to analyse the whole catalogue to find patterns to solve a particular problem. Instead, this work categorises patterns by intent so as to give the reader a clear motivation for each pattern and provide a clear path for readers aiming to solve a specific problem.

Eight major categories were identified in the patterns selected for this work; these are detailed in § 3.2. Each pattern is categorised by its primary intent, but it may belong to multiple categories secondarily. This is detailed at the start of each chapter. Thus, instead of having to provide a guide on how to reach the desired solutions, this work aims to bake this information directly into its structure.

Other categorisations were considered but created issues, such as the following. Creating a *messaging* category would be useful to quickly identify which patterns require messaging brokers, but this would mean that patterns in §§ 4.3, 6.1 and 10.5 would need to be split or would cause confusion, as messaging is not necessarily part of them (even if it often is). Creating

a category for communicating with persistent data stores creates a similar issue, such as with § 10.1. On the other hand, traditional *structural* or *behavioural* categories from [24] would mostly only overlap due to the implication of the scope defined in § 1.1.

Ultimately, they were discarded due to their incompatibility and limited usefulness. In the end, patterns should never be applied unless they solve a specific problem [24, p. 41], and this work aims to facilitate this.

1.4 Pattern construction

The following structure was chosen for the individual patterns. While it is inspired by the structures used in the existing literature § 1.2, they are inconsistent.

Due to the educational nature of this work, this structure is designed for readers new to the topic. It is based on the author's experience from trying to understand these works and aims to solve the issues encountered, such as missing summaries, motivation for the pattern being buried in the text or overlapping section concerns.

Name and summary The name of the pattern based on the process in § 1.3 and as short as possible summary of the pattern to prime the reader for the content.

Argumentation A summary of all sources and specific patterns this pattern is based on to show why this pattern is relevant, why it is included, and how it relates to existing patterns.

Context What problems this pattern solves and what the prerequisites are for its application.

Solution An overview of how the pattern works, how to apply it and a visualisation of the pattern based on the design process in § 2.5

Potential issues Since each pattern has its trade-offs, these are defined directly after the solution to highlight the potential problems and challenges that might arise from applying the pattern.

Example An example of the application to give the reader a more graspable understanding of the pattern, based on the system defined in § 3.4.

Related patterns What other patterns are related to this one, and why. This section is used to explain differences between similar patterns and to provide a path for further reading.

Further reading Any additional sources that might be useful to better understand the pattern.

2 Design and implementation

This chapter details the process and rationale behind the visual style (§ 2.2), the diagram design process (§ 2.5), and the technological choices used to construct it and the accompanying website (§ 2.3).

A common downside of the literature detailed in § 1.2 is the use of uninteresting, sometimes downright bland visual styles. This is a shame, as these patterns are inherently visual constructs, and an engaging visual style can have a significant impact on the reader's understanding and engagement [27].

2.1 Inspiration

One exception, however, is [15], which served as an inspiration for the visual style of this work. Out of all of the works reviewed, it is the only one with a visual style crafted to fit, with consistent iconography and (albeit monochrome) colour scheme, even going as far as creating pictograms for each pattern.

It has its downsides, however. While understandable for print, the monochrome colour scheme shows its limits on a screen, where the sacrifice to readability feels particularly out-of-place, especially when paired with the muted blue used as an accent on the website, pushing the look to almost dreary.

Another point is the choice of typeface. The diagrams employ *Open Sans*[1], a well-made open-source font, but not known to be terribly exciting, which is further diminished by the fact that, at the time of writing, it is the second most popular typeface on Google Fonts.

Curiously, it is paired with *Lato*[2] on the website, also a well-regarded open-source font, but these typefaces have such little contrast between them that it almost seems something is off at first glance[3].

1. https://fonts.google.com/specimen/Open+Sans
2. https://www.latofonts.com/
3. Ironically, the better pairing used in the book is with *Times New Roman*

2.2 Visual style

This section discusses the choices made for the visual style of the diagrams and website, namely the typography in § 2.2.1 and the colour palette in § 2.2.2.

2.2.1 Typography

One challenge to overcome is that many conventional choices for sans typefaces seem dull when used in a diagram, as they are usually optimised for body text. For this reason, the diagrams in this work use *Space Grotesk*[4], a wonderfully funky open-source typeface that continually surprises when read. It becomes a bit overbearing when used in body text, but it works remarkably well for short spurts of text, such as names in diagrams.

Thus, the body text uses a more grounded, serif typeface, *Source Serif Pro*[5]. It pairs well (see fig. 2.1), is readable (even on screen), provides ample contrast, and includes beautiful italics.

Figure 2.1: Font selection

Lastly, for code snippets and URLs, *Space Mono* and *Source Code Pro* – two monospace counterparts to the typefaces mentioned above – were considered but proved too wide for use in regular body text, especially for URLs.

4. https://floriankarsten.github.io/space-grotesk/
5. https://adobe-fonts.github.io/source-serif/

Instead, this work uses Iosevka[6], a generated and parametrisable typeface, due to its very narrow structure. Specifically, it uses its 9th stylistic set, designed to mimic Source Code Pro idiosyncrasies, which works well with Source Serif Pro.

2.2.2 Colour palette

The colour palette stems from the practical needs of the diagrams. It is built upon a simple contrasting black and white base and adds two tones for increasing and decreasing the level of focus – an orange–yellow hue for highlights adds an inviting warmth to the diagrams, and a light, unobtrusive grey for background objects finishes the visual hierarchy.

The palette also includes a light red for conveying failures or errors, and finally, a dark, cold green adds a sense of comprehensiveness to the palette and is useful as a secondary colour for foreground elements.

The palette roughly follows the popular 60–30–10 rule [28], providing a balanced and airy feel (see fig. 2.2). It offers a dark mode with slight modifications, such as decreased contrast for body text.

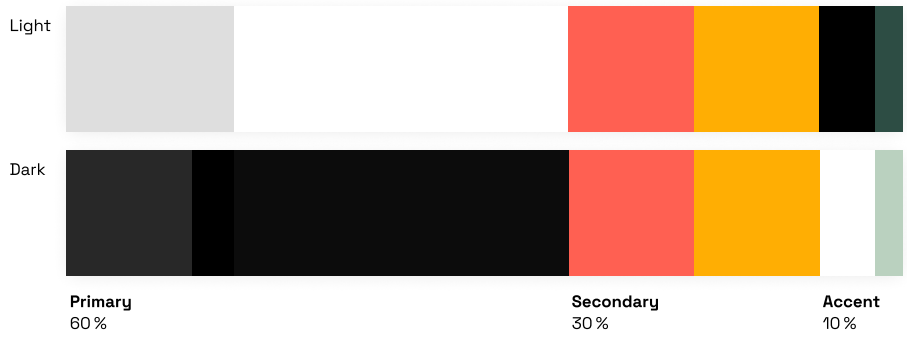

Figure 2.2: Colour palette

All colour combinations used for text have at least a 7:1 contrast ratio, achieving an enhanced WCAG rating (AAA) [29]. The palette performs well for different types of colour blindness, except for blue-yellow colour blindness, which is compensated for by consistent shape–colour pairings and supporting text (see § 2.4).

6. https://typeof.net/Iosevka/

2.3 Technologies

This section details the technologies used to create the diagrams (see § 2.3.1) and the website (see § 2.3.2).

2.3.1 Diagrams

The diagrams are constructed using PlantUML[7], a text-based diagramming tool that allows for easy version control and global styling. Joshi [5] already proved its validity for this use case. It presents some challenges, as its lack of granularity for controlling layout and limited style options are most visible with big, prominent designs, such as those in this work.

The biggest sore point is activity diagrams, which do not allow any control over the layout of the elements, so even simple diagrams can become cluttered and hard to read. While PlantUML does have tools to achieve the layout needed, to access them, one has to switch to a different diagram type, which makes start and end nodes inaccessible. One workaround is to design it as a state diagram with a `hide empty description` statement, such as fig. 10.3, which produces visually identical output, albeit requiring more manual work.

Another common issue is alignment. While PlantUML generally places elements on a grid, it is relatively easy to be left with angled, instead of perpendicular, lines, especially with nested elements. One possible solution is to strategically apply `norank` to arrows, which relaxes some constraints on the layout and may help straighten out the rest, such as in the aforementioned fig. 10.3. Switching to a `left to right direction` can help with making horizontal forks balanced, such as in fig. 6.4.

However, eventually, the last option is to give in and change the design to fit the tool or accept the limitations, such as with fig. 8.5. Some examples are the inability to style certain elements, such as some types of padding, arrowheads, or line thickness being inconsistently applied. Other examples are documented in § 2.4.

Other tools exist, such as Mermaid[8], but none seem to provide the same level of features and control as PlantUML.

2.3.2 Website

To account for dual-generation of this work – one into a PDF and one into a website – the text is written in Markdown with YAML front matter for custom

7. https://plantuml.com/
8. https://mermaid.js.org/

metadata, such as subtitles. The LaTeX Markdown package by Novotný [30] is used to translate them for LaTeX with custom renderers to read the front matter and generate the subtitles accordingly.

Rendering HTML is a bit more tricky. One option is to use Pandoc with the `chunkedhtml` output [31, p. 147], but that only has support for document-level metadata. To account for that, the front matter is extracted with Python into a JSON file before processing with Pandoc, for later use with Eleventy [9], a static site generator tool.

Styling is done with Tailwind CSS[10] and it is hosted on GitHub Pages[11]. The website has support for mobile devices and dark mode.

2.4 Visual language

The visual language design is largely based on UML but with some modifications to make it more simple, intuitive and visually appealing.

Natural language is used instead of keywords, visual cues are preferred over text, and multiple instances are explicitly shown instead of denoting cardinality to make the diagrams as accessible and simple as possible.

Other aspects of the visual language are dictated by the limits of PlantUML (see § 2.3), which, while having some control over the style and layout, is fairly limited. This ultimately worked well for the overall direction of the diagram design but did present some challenges.

The original design featured a jagged line used for errors, which would improve the connotations of failure and also the recognisability with blue-yellow colour blindness. However, such a style is unsupported by PlantUML. Using diagonal corners or dashed lines did not produce the desired effect, so the final design settled on sharp edges to convey errors.

Another unfortunate limitation is the lack of hexagons, which some newer works use to denote individual services [20, 3] (in reference to the *hexagonal architecture* by Cockburn [32]). This would allow for a very clear visual distinction between services and other entities, and PlantUML has added support for hexagons in later releases. However, they are not present in all diagram types, such as sequence diagrams, and even in other diagrams, they look unsightly oblong, with no control over their appearance. Thus, the diagrams have to resort to consistent naming to mitigate this.

9. https://www.11ty.dev/
10. https://tailwindcss.com/
11. https://jurf.github.io/daap/

2.5 Design process

Creating ideograms for the patterns, such as those employed in 2.1 was considered, but the issue is that by nature, they can only convey a limited amount of information to those unfamiliar with the underlying idea. However, having a simple, understandable representation of the pattern could be a very useful learning tool.

Therefore, this work instead uses a process to create pictograms called *eidetic reduction* [33, pp. 39-42]. Eidetic reduction is a process defined by Husserl in 1913 [34], which involves iteratively simplifying a concept, keeping only the elements essential to its representation, i.e. which constitute its *essence*. This is a time-consuming process that requires a lot of experimentation but results in much simpler diagrams than those used in the literature reviewed in § 1.2 with only a limited sacrifice to informativeness.

3 Overview

This chapter presents an overview of the patterns and any additional information needed to fully understand them and the problems they solve. This is discussed in § 3.1. Section 3.2 presents an overview of the pattern categories, § 3.3 describes the conventions used in this work, and finally § 3.4 introduces the example application used to showcase the patterns.

3.1 Prerequisites

ACID transactions A common acronym for the four properties of database transactions: *Atomicity*, *Consistency*, *Isolation*, and *Durability* [35, 36].

BASE An alternative acronym to ACID for *eventually consistent* systems: *Basically Available*, *Soft state*, and *Eventual consistency* [37, 2, p. 132].

CAP theorem See § 3.1.2.

CRUD A common acronym for the four basic operations of persistent storage: *Create*, *Read*, *Update*, and *Delete* [38].

Event-driven architecture An umbrella term for systems where components communicate by emitting and reacting to events. Fowler separates it into **Event Notification**, **Event-Carried State Transfer**, **Event Sourcing** (see § 10.6), and **CQRS** (see § 8.5) to improve reasoning about it [39]

Fallacies of distributed computing See § 3.1.1.

Idempotency A property of operations that can be applied multiple times without changing the result beyond the initial application [15, p. 197, 4, p. 469].

Message broker, message-oriented middleware See § 4.2.7.

Poison message A message that failed to be delivered enough times [40]. It can be moved to a **dead-letter queue** to wait for further inspection [4, p. 123].

Strict/eventual consistency The two possible consistency models in distributed systems: strict is a CP system, and eventual is an AP system (see § 3.1.2) [15, p. 127].

3.1.1 Fallacies of distributed computing

Writing distributed systems presents several challenges that are not present in standard monolithic applications. These challenges are often underestimated, which led to the formulation of the following *fallacies of distributed computing*, first coined by Deutsch and other colleagues from Sun Microsystems in 1994 [2, p. 124, 41].

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

Richards et al. [2, p. 131] also present additional considerations when creating distributed systems.

- Distributed logging makes tracking down problems more difficult
- Distributed transactions are not ACID
- Contract maintenance and versioning is more difficult due to the additional required coordination between teams and departments

3.1.2 CAP theorem

The CAP theorem was formulated by Brewer [42] in 1999 and later proven by Gilbert et al. [43] in 2002. It states that in a distributed system, it is impossible to guarantee all three of the following properties simultaneously:

1. *Consistency* – each node gives the same answer
2. *Availability* – each request receives a response
3. *Partition tolerance* – the system continues to operate despite network failures

Thus, when building a distributed system, one can only choose two of these properties. AP systems (sacrificing consistency, i.e. *eventually consistent* systems) are easier to build and scale, and CP systems (sacrificing availability) are much harder. Fortunately, these trade-offs do not need to be made on a system level; they can be much more nuanced, such as on a component or operation level. [3, pp. 410-412]

3.2 Pattern categories

This work is structured around the following categories of patterns, starting with two *building block* categories:

1. *Communication patterns* (chapter 4) shows options for communicating between components and the outer world
2. *Decomposition patterns* (chapter 5) shows common options on how to deal with shared functionality

These patterns are then used to achieve the following *quality attributes*:

3. *Scalability patterns* (chapter 6) shows how to scale and manage scalable systems
4. *Resilience and reliability patterns* (chapter 7) addresses the need for handling inevitable network, software or hardware failures
5. *Performance and latency patterns* (chapter 8) shows how to solve common performance or latency issues arrising in scaled or distributed systems
6. *Security patterns* (chapter 9) shows security practices that can be supported by the architecture of the system
7. *Consistency patterns* (chapter 10) details how to manage transactions when communicating between services

Finally:

8. *Migration and compatibility patterns* (chapter 11) shows patterns useful for managing legacy or external systems

Patterns were distributed by their primary intent but may belong to multiple categories secondarily. This is detailed at the start of each chapter.

3.3 Conventions used

This work uses the following typographical conventions.

- **Bold Title Case** is used to identify (the first use of) a pattern name, even beyond the scope of this work
- **normal bold** is used to highlight important keywords, namely in contexts or potential issues of a pattern
- *Italic text* is used to introduce new terms or concepts
- `Monospace` is used for code snippets or URLs

It also uses a consistent visual language, defined in § 2.4, and terminology, defined in § 3.3.2.

3.3.1 Diagrams

The diagrams use a UML-inspired [44] visual language, shown in fig. 3.1, with the following interpretations:

- Arrow directionality implies only focus; communication can still flow in the opposite direction
- *Virtual* diagram components are either logical or optional, depending on the context
- A *node* is whatever computational unit is relevant, such as a server or a virtual machine
- An *error* in the place of a *service* implies a *failed service*. This is always also denoted in the text, such as in 7.1.

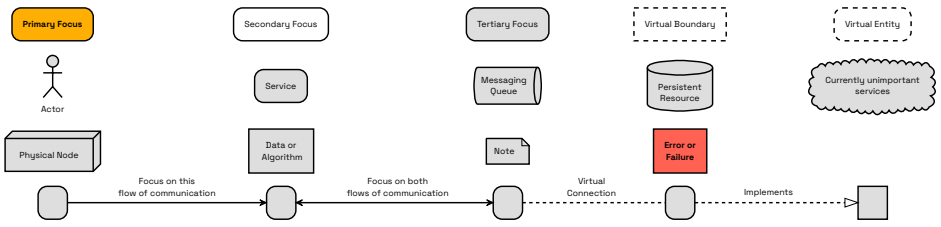

Figure 3.1: Visual language

3.3.2 Terminology

Client/Service A relationship between an abstract "user" of a service that does not necessarily have to be part of the same system and an autonomously running software component that can be accessed over a network.

Producer/Consumer A relationship between two services with a focus on the direction of data or message flow.

Request/Query A message from a client to a service that expects a response.

Node A single computational unit such as a server or a virtual machine.

Task A single computational operation that can be performed by a service.

Processor A component with a focus on processing data or messages.

3.4 Example application

This work uses a fictional e-commerce platform, *ExampleEshop*, to illustrate the patterns in a real-world context. The platform's name is purposefully generic, not to require any additional context to understand its purpose. It also does not use any specific technology stack or architectural style; additional details are always provided along with the pattern example.

For a better mental picture, *ExampleEshop* has roughly the following properties. Some are arbitrary but were chosen for consistency, but also to provide a simple enough domain so as not to require additional explanation.

1. It is large enough that it needs a distributed application to handle the required load
2. It employs multiple development teams
3. It does not allow third-party sellers, only selling its own products, but it has a broad selection of items
4. It operates on a multi-national, but not necessarily a global scale

4 Communication Patterns

This chapter presents the fundamental building blocks regarding communication in distributed systems.

1. The **Gateway Routing** pattern in § 4.1 abstracts the location of the underlying services from any clients, presenting the system as if it was a single service
2. The **Publisher–Subscriber** pattern introduces a common way to decouple and scale communication between services in § 4.2 and message brokers, a common way to implement this communication style, in § 4.2.7
3. The **Asynchronous Request–Reply** pattern then fills in a gap introduced by employing messaging or asynchronicity in general, that is, how to handle responses, in § 4.3

As distributed systems necessitate communication, many other patterns could be included in this chapter but are presented elsewhere due to their focus on other aspects of the system, such as:

- **Ambassador** (see § 5.2) handles communication on behalf of a service
- **Queue-Based Load Leveling** (see § 7.2) focuses on how messaging can improve reliability
- **Competing Consumers** (see § 6.1) and **Scatter–Gather** (see § 6.3) extend messaging to bring scalability
- **Messaging Bridge** (see § 11.3) shows how to connect two systems using incompatible messaging
- **Transactional Processor** (see § 10.1), **Transactional Outbox** (see § 10.2), and **Event Sourcing** (see § 10.6) show how to communicate transactionally
- **Choreography** (see § 10.4) and **Orchestration** (see § 10.5) show how to coordinate processes in services
- **Colocate** (see § 8.2) uses physics to speed up communication and make it more reliable

- Patterns in ch. 7 can be necessary to handle communication failures

Overall, this is just a subset of the topic and was covered in depth by Hohpe et al. [4] or Richardson [20, p. 85].

4.1 Gateway Routing

Abstract service locations from the clients

This pattern is based on **Gateway Routing** by Microsoft [45], **Virtual Endpoint** by Rotem-Gal-Oz [13, p. 64], and the **API routing** patterns by Deenadayalan [46]. Hohpe et al. define this pattern more generically with the **Message Router** and **Content-Based Router** in the context of a **Pipes and Filters** architecture [4, pp. 91, 211, 47, 48].

This pattern is sometimes considered a part of the **API Gateway** pattern. For notes on why this work breaks it down, see § 4.1.7.

When creating a client-facing distributed system, one of the first apparent problems is how a client will interact with it since there is no longer a single endpoint. The easiest solution is to create a common routing service – a **Gateway Router**.

4.1.1 Context

Apply this pattern if there is a need to decouple service configuration from clients and at least one of the following conditions holds.

1. The client needs to communicate with **multiple backend services**, and their configurations change frequently, or it is not feasible to update the clients
2. There is one backend service, but it has **multiple running instances** or **multiple versions**, perhaps for progressive rollout or A/B testing
3. Traffic should be redirected to other working instances in case of **failure** or **overload**

4.1.2 Solution

Introduce a singular gateway component (see fig. 4.1). Clients interact with the system only through this component, containing all the necessary rules to route the requests. This can be based on several criteria, such as the

hostname, URL, port or header. The gateway then forwards the request to the appropriate service. This way, the clients never need to know the actual service locations.

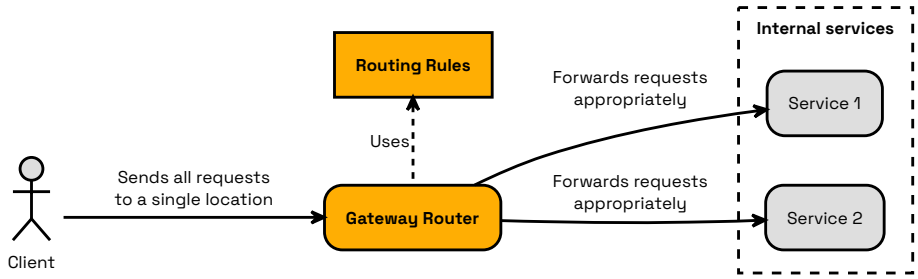

Figure 4.1: Gateway Routing

4.1.3 Potential issues

The biggest downside of this pattern is that it introduces a **single point of failure** – see chapter 7 on common techniques to reduce resiliency in similar situations. Another important aspect is the increase in **latency** due to an added jump in the requests and the introduction of a possible **bottleneck** to the system.

From a higher point of view, introducing this pattern makes it a very appealing point to add more features. This can accentuate the problems outlined above. [3, p. 162]

4.1.4 Example

ExampleEshop (see § 3.4) uses different APIs (see § 5.4.4) for its different frontend. In order to avoid having to update all its apps after changes in deployments, it uses a single entry point that routes the requests to the correct backend based on the path. Thanks to the amount of control this gives them, they can also use it to test new versions of the backend services, run experiments or re-route traffic in case of failures.

4.1.5 Related patterns

- From another point of view, this pattern can be seen as an application of **Partitioning** (see § 6.2)

- Instead of having a universal gateway, which might introduce a risk of handling too much, consider breaking it down into multiple **Backends for Frontends** (see § 5.4) to better split the concerns

- **Aggregating Gateway** (see § 8.3) builds upon the router and consolidates multiple services calls into one to increase client performance

- Use **Offload to Gateway** (see § 5.3) to offload common tasks to the gateway to simplify services

- Introduce load balancing into the gateway with **Competing Consumers** (see § 6.1)

4.1.6 Further reading

- This functionality is often implemented in a **Reverse Proxy**. Nygard [12, p. 179] discusses their capabilities in the context of load-balancing.

4.1.7 API Gateway

Gateway routing is sometimes considered part of an **API Gateway** pattern. However, that pattern, by definition, is a set of multiple functionalities, and that coupling can lead to it being sometimes considered almost an anti-pattern [3, p. 162]. However, these discrete functionalities do not need to be implemented in one system. They can even be necessary in some use cases, so this work adopts the naming scheme propagated by Microsoft [17], which splits the API Gateway into three discrete patterns – **Gateway Routing**, **Aggregating Gateway** (8.3) and **Offload to Gateway** (5.3) [45, 49, 50].

The following sources are still very useful for insight into the individual functionalities of an API Gateway, although they focus more on them being deployed as a whole.

- Richardson [20, p. 260, 51] has a mostly positive take, arguing that they are an essential part of most systems and can be very useful

- Newman [3, p. 162], on the other hand, is much more sceptical of its use in, especially third-party solutions, but he admits it can have a use under the right conditions, i.e. when the team is small enough, and it does not handle aggregation and filtering

- The API management *Gateway* component on Wikipedia [52]

4.2 Publisher–Subscriber (Pub–Sub, Topics)

Asynchronous one-to-many communication

This pattern is based on the **Publish-Subscribe Channel** by Hohpe et al. [4, p. 113, 53], **Publisher-Subscriber** by Buschmann et al. [54, p. 234] and Microsoft [55] and **Publish-subscribe** by Deenadayalan [56].

The name for this pattern is sometimes interchangeably used for communication through message brokers in general; see § 4.2.7.

4.2.1 Context

There is a need for **asynchronous communication** between a **publishing service** and one or more **consuming services** (subscribers). Other supporting factors include

- The need for **loosely coupled** or **fluctuating** components in the system
- The need for **message buffering** or **guaranteed delivery**

4.2.2 Solution

Introduce a message broker as an intermediary between publishers and subscribers (see fig. 4.2). This broker manages "topics," which consist of one input queue for the publisher and multiple output queues, one for each subscriber. The publisher sends messages to the input queue, which the broker processes and propagates to all the output queues.

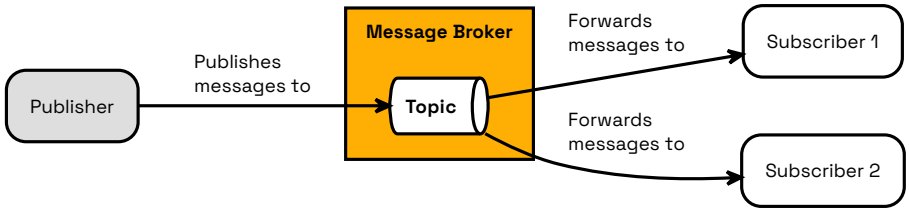

Figure 4.2: Publisher–Subscriber

The broker does not need to manage only one queue; once set up, it can handle multiple topics, and each subscriber can choose which topics to subscribe to.

Existing message brokers typically also provide guaranteed delivery, but the implementation details are challenging and vary [3, p. 135] and, as such, are outside the scope of this work. Other patterns described here may assume that the broker provides this feature.

4.2.3 Potential issues

Using a broker means introducing another component into the system, leading to **increased complexity** and possibly increased **latency**. Also, due to its nature, it can become a **single point of failure** and thus a possible **bottleneck**.

Having multiple subscribers can lead to **ordering issues** if the order of messages is important, and, as with any network communication, **message duplication** is a possibility and needs to be handled.

Furthermore, implementing a custom message broker may not be feasible due to limited development resources, which may mean **trusting a third-party service**.

4.2.4 Example

ExampleEshop (see § 3.4) uses a read-only database (see § 8.5.4) to improve their performance when serving product pages. However, they have found that even this is not enough to handle the surge in traffic during large events. To address this, they decided to replicate their read-only database. To ensure that the data is consistent across all replicas, they subscribe to the same topic to receive updates.

4.2.5 Related patterns

This pattern directly facilitates **Scatter–Gather**, which deals with destructuring workloads into smaller tasks and reassembling the results (see § 6.3). It is also possible to use a single messaging queue for communication instead of a topic, which still includes the benefits of message buffering.

1. Using it in one-to-one communication results in **Queue-based Load Levelling** (see § 7.2)
2. Using it for one-to-many communication results in load-balancing **Competing Consumers** (see § 6.1)

4.2.6 Further reading

- The “Gang of Four” **Observer** pattern [24, p. 273] this pattern builds upon with asynchronous messaging

- Wikipedia [57]
- The *Inbox and Outbox* pattern can also be used for guaranteed delivery [58]

4.2.7 Message Broker

The categorisation, naming, and cross-section of messaging patterns vary. Most centre around messaging middleware and queues [4, p. 76, 20, p. 85, 15, p. 136, 14, p. 27], whilst others assume that terms such as "pub/sub" and "competing consumers" are common knowledge and do not need explaining [3, p. 445, 12, p. 117, 20, p. 92]. This work errs instead on caution and fidelity, presenting the most common communication patterns but focusing primarily on the context in which they are applied: see §§ 4.2, 4.3, 6.1, 6.3, and 7.2.

- Newman on topics [3, p. 135] and the **Event-Driven Communication** pattern [3, p. 108]
- Richardson's general **Messaging pattern** in *Microservice Patterns* [20, p. 87, 59]
- **Message-Oriented Middleware** by Fehling et al. [15, p. 67]
- Burns [22, p. 129] calls it *publisher/subscriber* infrastructure

4.3 Asynchronous Request–Reply

Asynchronous two-way communication

This pattern is based on **Request–Reply** by Hohpe et al. [4, p. 147, 60], **Asynchronous Request–Reply** by Microsoft [61], **Asynchronous Request–Response** by Hohpe [62], **Request-Response Communication** by Newman [3, p. 104], and partly on the **Messaging** pattern by Richardson [20, p. 87, 59] and the **Decoupling Middleware** pattern by Nygard [12, p. 117].

The patterns have varying implementations but share the same structure and purpose. An overview is given below.

See also **Message Broker** (see § 4.2.7).

4.3.1 Context

Synchronous messaging has become a **bottleneck**, but using only **Publisher–Subscriber** (see § 4.2) or **Queue-Based Load Levelling** (see § 7.2) is not an option, as either

1. The client expects a response
2. One of the communicating parties cannot use a messaging broker due to technical limitations (e.g. a web browser), or when long-running connections are not feasible

4.3.2 Solution

There are three potential ways to implement this pattern.

1. Use a **message broker** (see § 7.2) and configure two queues: one for the request and one for the response. The sender sends a message to the request queue and waits for a response on the response queue.
2. Use **synchronous communication** to send the request, then use **polling** to check for a response on a pre-determined endpoint. This option is less efficient as there will be more communication overall, but it might be easier to implement with existing infrastructure or tools. The receiver can improve efficiency by replying with an estimate on the request completion [61].
3. Use **synchronous communication** and a **callback** mechanism. The sender sends a request and a callback address. The receiver processes the request and sends the response to the callback address. [62]

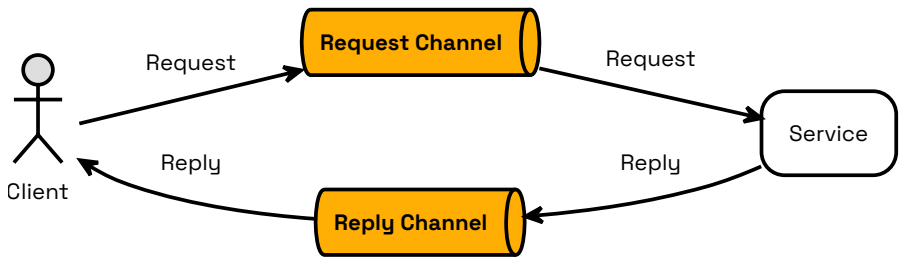

Figure 4.3: Asynchronous Request–Reply

4.3.3 Potential issues

Compared to synchronous communication, this pattern brings **increased latency** and **complexity**. If implemented using synchronous calls, **Retry** logic (see § 7.3) might need to be employed to handle transient failures.

Using synchronous communication is faster, simpler and more efficient if the requirements allow it.

Using this pattern to communicate with multiple parties may result in requests being processed **out of order**. This can be mitigated using a **Correlation Identifier** [4, p. 154, 63].

See also § 4.2.3.

4.3.4 Example

Creating an order in *ExampleEshop* (see § 3.4) is a complex operation implemented as a distributed transaction (see § 10.3.4), which can take longer than the timeout of a synchronous call. However, they want to provide feedback to the user as soon as possible. Still, these updates do not have to be long-term, as analytics showed that users stop refreshing the page once they receive confirmation that their payment is through and an estimated delivery time.

Thus, for a simple implementation, they opted to poll the server periodically to check the order status in the initial phases when many changes are happening. Once the order is confirmed, the frontend stops polling and informs the user that they will receive further updates via email.

4.3.5 Related patterns

- While not necessary, **Queue-Based Load Leveling** (see § 7.2) can be used to improve handling peak demand
- **Retry** (see § 7.3) logic might be necessary if using synchronous communication

4.3.6 Further reading

- *Request-response* on Wikipedia [64]

5 Decomposition Patterns

This chapter discusses how to code duplication and cross-cutting concerns in a distributed system. It presents patterns that can be used to implement many others in chapters 6–11. It starts with two single-node patterns by Burns [22].

1. **Sidecar** in § 5.1 is a pattern that allows to offload cross-cutting concerns to a locally running component, such as a container or a process, in a programming language–agnostic way – which has since become a general term for any locally running supporting component
2. Similarly, **Ambassador** in § 5.2 is a pattern that allows to handle network communication on behalf of a service, thus allowing it to be extended with any functionality needed

It then continues with two common approaches to handle client-facing cross-cutting concerns:

1. **Offload to Gateway** in § 5.3 is a simple pattern that puts shared functionality into a gateway, which works well for smaller applications but does not scale
2. **Backend for Frontend (BFF)** in § 5.4 is a (more costly) solution that overcomes this issue and has been shown to work well in practice

In a sense, other patterns could also be considered a part of this category:

- **Bulkheads** (see § 7.1) define general principles on how to decompose, structure or isolate a system
- **Identity Provider** (see § 9.1) and **Externalised Configuration** (see § 6.4) both extract functionality into common services
- **Gatekeeper** (see § 9.2) forces decomposition to isolate security incidents and **Aggregating Gateway** (see § 8.3) to improve performance of unreliable clients
- **Anti-Corruption Layer** (see § 11.1) extracts functionality to preserve semantics between different systems
- **Messaging Bridge** (see § 11.3) is a shareable component used for connecting two services with incompatible messaging middleware

Notable omissions in this category due to the methodology described in § 1 include:

- The **Adapter** pattern by Burns [22, p. 31], a complement to sidecars and ambassadors, a term that did not seem to catch on, in contrast to the other two. In a sense, it can be seen as a *sidecar anti-corruption layer* and is presented in this work as such in § 11.1
- The **Service Mesh** pattern by Richardson [20, p. 380], a concept mostly used in the context of microservices

5.1 Sidecar

Language-agnostic, locally-running supporting components

This pattern is based on **Sidecar** defined by Burns [65, later 22, p. 11]. It was originally defined specifically for implementation with containers but has since become a general term for any process or container running alongside a main service; see Richardson [20, p. 410, 66] and Microsoft [67].

5.1.1 Context

Multiple services are created with different, **incompatible technologies** and need to **share some functionality**. Alternatively, the services might need to offload some functionality so that it does not affect them, but it is still advantageous to have it close by.

5.1.2 Solution

Implement a small, locally running **Sidecar** service that provides the necessary functionality (see fig. 5.1). This service communicates with the main service over the network and can be deployed with multiple other services. Since the sidecar is running on the same node, it has access to all of the same local resources as the main service.

This relationship does not necessarily have to be one-to-one[1]. Depending on the need, multiple services can attach to one sidecar, and one service can have multiple sidecars attached to it.

1. This is where the metaphor falls apart slightly

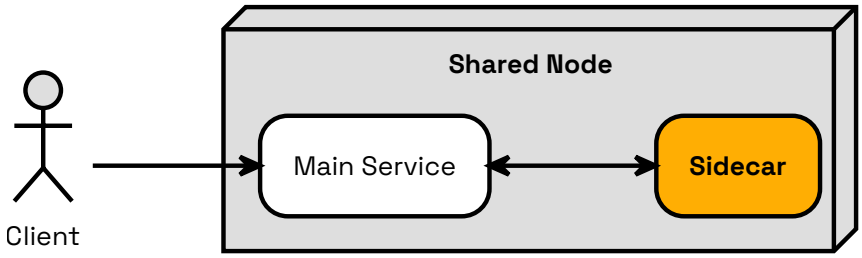

Figure 5.1: Sidecar

5.1.3 Potential issues

Compared to a shared library, a sidecar will still add unnecessary **overhead** and **latency**, even though it is running nearby. Since it lives separately from the main service, it also adds **additional complexity** to the system. Furthermore, if the sidecar fails, it can affect the main service, and debugging across service boundaries can be more challenging.

Depending on the type of functionality the sidecar provides, it can also lead to **tight coupling** between the service and the sidecar, potentially leading to the sidecar becoming a **development bottleneck**.

Since a sidecar runs on the same node as the main service, it can be **difficult to scale**. If the sidecar needs to handle disproportionately more work than the main service, consider offloading the functionality to a separate service instead.

5.1.4 Example

ExampleEshop (see § 3.4) needs to process product images, but it also allows users to post images in their product reviews. These functionalities are handled by different services using different technologies. These need to be converted into different formats and sizes, such as thumbnails or lower-resolution images for mobile devices. To reduce code duplication, the image processing is handled by a sidecar service deployed alongside both services. This has the added benefit of allowing more fine-grained control over the resource usage of the image processing and isolating any potential failures from the main services.

5.1.5 Related patterns

This pattern expands upon the idea of the **Colocate** (see § 8.2) pattern. The following patterns are commonly deployed as sidecars to improve performance.

1. **Ambassador** (see § 5.2), for placing helper services between clients and main services (see § <>)
2. **Anti-Corruption Layer** (see § 11.1), for abstracting different domains

Health Monitoring (see § 7.4) can be beneficial to implement as a sidecar due to its close access.

5.2 Ambassador

Handle network communication on behalf of a service

This pattern is based on **Ambassador** defined by Burns [65, later 22, p. 21] and Microsoft [68], and **Client Proxy** by Buschmann et al. [54, p. 240]. It can also serve as a **Channel Adapter** by Hohpe et al. [4, p. 128, 69].

5.2.1 Context

This pattern handles the same problem described in § 5.3.1 – there is a need to offload commonly implemented functionality from services. Alternatively, a legacy service cannot be altered but may still need some functionality added to it.

5.2.2 Solution

Implement a small service that acts as an intermediary between the main service and the network (see fig. 5.2). This service can be deployed alongside the main service and enrich its networking capabilities without having to modify the existing service. It can also be reused across multiple services.

5.2.3 Potential issues

Compared to a shared library, using an ambassador will result in **increased latency** due to the extra network hop, **added system complexity**, and an **additional point of failure**. If an ambassador is shared across multiple services, it might become a **bottleneck**.

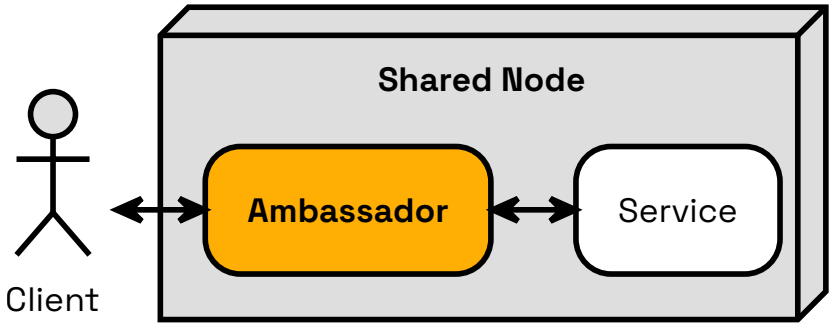

Figure 5.2: Ambassador

Consider the impact of offloaded features on the underlying service. For example, automatically employing **Retry** logic (see § 7.3) might cause problems if the service endpoint is not idempotent.

5.2.4 Example

ExampleEshop (see § 3.4) wants to try out a new version of its recommendation service with an updated model and evaluate its performance, but it is a risky operation, so it needs to be rolled out gradually. This is a temporary experiment, so they do not want to change the main service. However, as the recommendations are generated in real-time as users browse the site, it is critical for the experiment not to affect the latency of the application.

To achieve this, they deploy an ambassador service that intercepts requests to the recommendation service and forwards a percentage of them to the new version. Once the experiment is over, the ambassador service is removed without needing to change the main service. [22, p. 27]

5.2.5 Related patterns

- An ambassador is commonly deployed as a **Sidecar** (see § 5.1) to improve performance

- Patterns implementable in a gateway include **Rate Limiting** (see § 7.5) , **Circuit Breaker** (see § 8.1) , **Retry** (see § 7.3) , **Partitioning** (see § 6.2) and **Health Monitoring** (see § 7.4)

- Compared to **Offload to Gateway** (see § 5.3) , this pattern has the added benefit of being able to be configured per service and be deployed as a sidecar for reduced latency. However, unlike it, it cannot be easily extended with aggregation (see § 8.3)
- While an **Anti-Corruption Layer** (see § 11.1) lives in the same space as an ambassador, they differ in purpose. An ambassador can be reused across services; an ACL is, by definition, specific to a pair of services and exists to isolate domain-specific logic
- The **Gatekeeper** (see § 9.2) pattern has a similar function but is deployed in a restricted environment for added security

5.3 Offload to Gateway

Move common functionality to a gateway to offload backend services

This pattern is based on **Gateway Offloading** by [50]. It is part of the three patterns sometimes called an **API Gateway**, as detailed in § 4.1.7. Of the three, this one is most prone to its pitfalls but can nevertheless be helpful in some scenarios.

The original name seems to imply that it is the gateway *itself* being offloaded. Therefore, this work uses the term **Offload to Gateway** instead, which should better denote the direction of the offloading whilst preserving recognisability.

5.3.1 Context

When decomposing a system into multiple services, common problems may arise, such as caching, logging, monitoring, rate limiting, protocol translation, authentication or authorisation. Repeating re-implementation of these functionalities violates the basic *Don't Repeat Yourself* (DRY) [70, p. 30] principle. It thus can be needlessly expensive and – perhaps more importantly, when it comes to security – error-prone.

Whilst usable in many scenarios, this pattern may be best utilised with individual small teams due to the issues described below.

5.3.2 Solution

Introduce a gateway component that handles these cross-cutting concerns, simplifying the underlying services (see fig. 5.3).

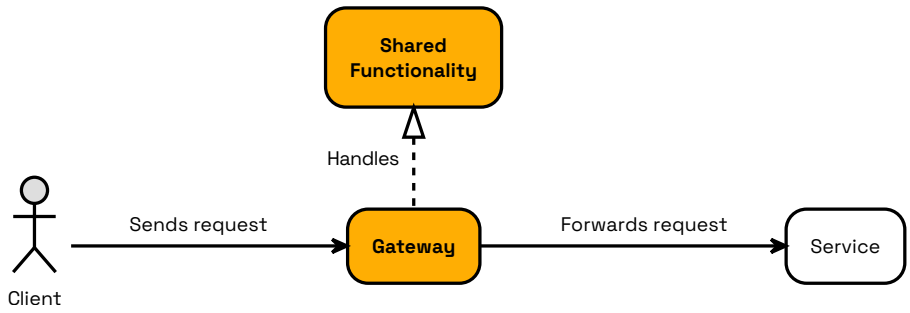

Figure 5.3: Offload to Gateway

5.3.3 Potential issues

Similarly to **Gateway Routing** (see § 4.1), this pattern can introduce a **single point of failure**, a **bottleneck** and **latency**.

However, this pattern also has the potential to exacerbate these issues due to the amount of functionality it can accumulate. These can lead to further problems, such as introducing **tighter coupling** between the services and the gateway and becoming a **development bottleneck**. [3, p. 162, 20, p. 267]

If using this pattern to implement authentication or authorisation, see § 9.1 on potential security issues inherent to such a design.

5.3.4 Example

ExampleEshop (see § 3.4) already uses a gateway to route requests (see § 4.1.4) to correct services and need to employ rate limiting (see § 7.5.4) to prevent abuse. As this needs to be applied to all user-facing services and this functionality does not contain any business logic, they decided to go the simple route and implement it in the gateway itself.

5.3.5 Related patterns

- This pattern builds upon **Gateway Routing** (see § 4.1), as it must be a single entry point

- Patterns implementable in a gateway include **Rate Limiting** (see § 7.5), **Competing Consumers** (see § 6.1), **Circuit Breaker** (see § 8.1), **Retry** (see § 7.3) and **Health Monitoring** (see § 7.4). However, see the warnings in the previous section

- One possible mitigation of these issues is to separate the gateway into multiple **Backends for Frontends** (see § 5.4), reducing its overall scope and overlap. Another solution can be to use small services deployed alongside applications to offload some concerns, such as an **Ambassador** (see § 5.2), which can handle network-related shared functionality, or a **Sidecar** (see § 5.1), which can handle isolated concerns
- Simplify the client instead by using **Aggregating Gateway** (see § 8.3)
- A gateway can communicate with an **Identity Provider** (see § 9.1) to authenticate clients

5.3.6 Further reading

- This pattern is often implemented with a **Reverse Proxy**. For several examples, see Burns [22, p. 53] and [12, p. 179]

5.4 Backend for Frontend (BFF)

Reduce backend complexity by specialising for each frontend

This pattern is based on **Back-end for Front-end (BFF)** created by Calçado [71], and later by Newman [72, 3, p. 480], Richardson [20, p. 265, 51] and Microsoft [73].

5.4.1 Context

There are several types of clients with the system, and they require **server-side aggregation**. While the clients are typically individual frontends, this pattern can apply in other scenarios, such as providing an API to a third party [3, p. 487].

5.4.2 Solution

Implement a separate backend for each client. These backends process data from shared services and handle all the necessary aggregation for the client. This provides a natural separation of concerns, helping mitigate the core issue of Offload to Gateway.

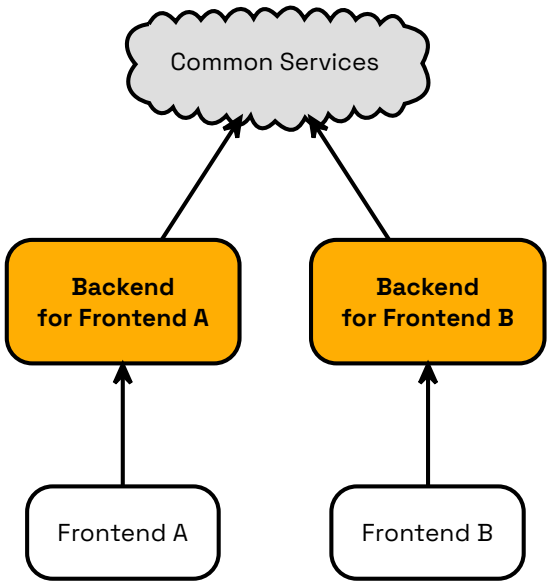

Figure 5.4: Backend for Frontend (BFF)

5.4.3 Potential issues

BFFs can introduce **duplicated code**. There are several potential ways to deal with this problem.

1. Extract the code into a shared library, thereby introducing coupling between the BFFs
2. Merge the BFFs into a single service, thereby devolving back into **Offload to Gateway**
3. Use the knowledge of where the duplication arises to introduce new shared services
4. Tolerate the duplication as it may be worth the trade-off for the better separation of concerns this pattern provides [3, p. 484]

Of course, as always, when introducing any new service, there is the risk of introducing **increased complexity** and **development overhead** to the system.

5.4.4 Example

ExampleEshop (see § 3.4) provides a web interface and two mobile applications for different platforms. Each backend is a separate BFF conforming

to the different needs and functionality of the frontends. The mobile app backends might provide a coarser-grained API to reduce the number of requests, leverage that phone screens are smaller and can't display as much information as a desktop browser, or be structured differently due to the differing designs and user experiences. [72]

5.4.5 Related patterns

- Even with instances with one client, using **Aggregating Gateway** (see § 8.3) on itself can still be helpful to simplify logic on the frontend and increase performance on poor networks
- Use the **Incremental Replacement (Strangler Fig)** (see § 11.2) pattern to migrate to BFFs
- Use a **Identity Provides** (see § 9.1) to authenticate requests
- Use an **External Configuration Store** (see § 6.4) to offset the complexity of managing multiple BFFs

6 Scalability Patterns

This chapter details patterns that help to scale and manage scalable systems – one of the fundamental driving forces behind distributed systems.

1. **Competing Consumers & Load Balancer** in § 6.1 have differing implementations, but the same underlying structure and intent – to distribute work among multiple identical consumers
2. **Partitioning** in § 6.2, also known as *sharding*, is a similar pattern, but with predetermined responsibilities for each participant, be it data or tasks
3. **Scatter–Gather** in § 6.3 presents a common way to work with different forms of partitioning
4. **Externalised Configuration** in § 6.4 deals with management of scaled systems

Other patterns that could be considered a part of this category include:

- **Publisher–Subscriber** (see § 4.2) can be used to scale broadcasts
- **Choreography** (see § 10.4) which helps scalability by losing dependence on a central coordinating, potentially bottlenecking, service
- **Identity Provider** (see § 9.1) improves handling of secrets in scaled systems
- **Sidecar** (see § 5.1) for introducing shared functionality without a bottleneck
- **Ambassador** (see § 5.2) for plugging scaling services into existing systems
- **Colocate** (see § 8.2) to apply physical restrictions to scaling to improve scaling efficiency
- **Rate Limiting** (see § 7.5) to protect scaled systems from individual abusive clients
- **Leader and Followers** (see § 7.6) for coordination in scaled systems
- **Event Sourcing** (see § 10.6) for easier replication in scaled systems

Notable omissions in this category due to the methodology described in § 1 include:

- **Pipes and Filters** by Hohpe et al. [4, p. 85] and Microsoft [74], due to its classification as an architectural style by Richards et al. [2,

p. 143]. Furthermore, describing it in the context of distributed systems does not add much and presents notions already discussed in **Choreography** (see § 10.4)

- **Map Reduce** by Fehling et al. [15, p. 106] and Wilder [14, p. 59], as it usually depends on specific products, and has an identical structure to scatter–gather
- **Geode (Geographic Nodes)** by Microsoft [75], which does not appear outside of Azure and seems to just be a combination of load balancing and eventual consistency wrapped around specific products

6.1 Competing Consumers & Load Balancer

Continuous parallel request processing

This pattern is based on two patterns which share the same structure and purpose.

1. **Competing Consumers** by Hohpe et al. [4, p. 446, 76], later by Microsoft [77]. It is commonly considered as part of the functionality of message brokers [20, p. 92, 3, p. 135]
2. **Load Balancer** by Hohpe [78] and the **Service Instance** pattern by Rotem-Gal-Oz [13, p. 62], which are commonly implemented by *reverse proxies* [12, p. 178]

See also **Message Broker** (see § 4.2.7).

6.1.1 Context

The only condition to be able to utilise this pattern is to have independent tasks and identical processors. There are a number of requirements that can fuel this decision.

1. The need for **increased throughput** or **decreased latency**, even under high load
2. The need for **resiliency** – if one consumer fails, the task can be picked up by another
3. The task is composed of several subtasks which are **highly dependent** on each other

6.1.2 Solution

Either use a **Messaging Queue** (see § 7.2) to distribute tasks to multiple identical consumers (see fig. 6.1), which "compete" for the tasks (whichever is faster takes the task), or use a [**Gateway Router**]#sec:gateway-routing) to distribute the tasks with a load balancing mechanism.

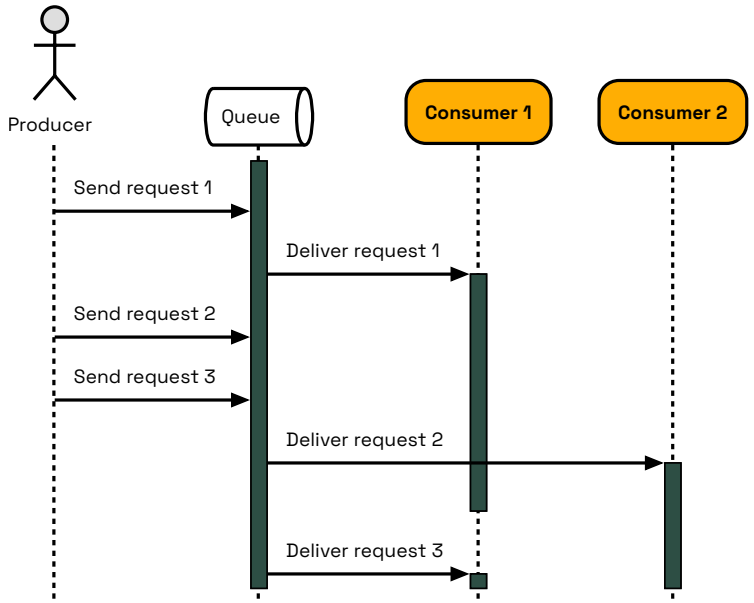

Figure 6.1: Competing Consumers

Since the processors are independent, their number can easily be scaled up or down, depending on the need.

6.1.3 Potential issues

Since the processors can have different speeds and latencies, messages might be processed in the **incorrect order**. There are two specific circumstances where incorrect message ordering can be mitigated.

1. If processing the task does not have any side effects, attach a **Correlation Identifier** [4, p. 154, 63] such as a timestamp or a sequence number and use **Scatter–Gather** (see § 6.3) to ensure the order of messages in the output.

2. If the messages only need to be ordered within a specific category, use **Partitioning** (see § 6.2) and a single consumer per partition

The limit of scaling is not infinite; the router may become a **bottleneck** if the consumers are faster. On the other hand, if the consumers are not fast enough, the router might be **overwhelmed** and may have to start dropping messages.

If resiliency measures are implemented, the system might need to be further guarded against **poison messages** or **duplicated processing**.

See also § 4.2.3.

6.1.4 Example

ExampleEshop (see § 3.4) uses a saga orchestrator (see § 10.3.4) to handle the distributed transaction it needs to implement orders. However, the system has trouble keeping up with the backlog of orders during large events, even if using threading or asynchronous processing. Since they are already using message queues (see § 7.2.4) to increase the reliability of the saga, they decide to employ competing consumers to process the orders in parallel.

6.1.5 Related patterns

- If the producers need a response, see **Asynchronous Request–Reply** (§ 4.3)
- To improve performance, consider using a **Claim Check** to offload the message queue if the messages are heavy (see § 8.4).

To improve reliability and resilience, this pattern can be extended with

- **Retry** for handling transient failures (see § 7.3)
- **Circuit Breaker** to maintain the system's responsiveness in their event (see § 8.1)

6.2 Partitioning (Sharding)

Scale out by separating tasks or data into logical partitions

This pattern is based on two distinct but architecturally similar groups of patterns.

1. Partitioning data is typically discussed in the context of databases as *sharding* [3, p. 426] – Wilder's **Database Sharding** [14, p. 67], Burns' **Sharded Services** [22, p. 59] and **Sharding** by Microsoft [79]

2. Partitioning tasks goes by different names, usually as a solution to the ordering issue of **Competing Consumers** (see § 6.1) – **Sequential Convoy** by Microsoft [80] or Richardson's *sharded channels* [20, p. 94]

6.2.1 Context

The data store or a specific task has become the **bottleneck**, or they **surpass the capacity** of a single machine. Still, it is possible to distribute them uniformly using some partitioning logic.

6.2.2 Solution

Instead of creating one shared data store or service, create multiple instances and divide the data or tasks between them (see fig. 6.2).

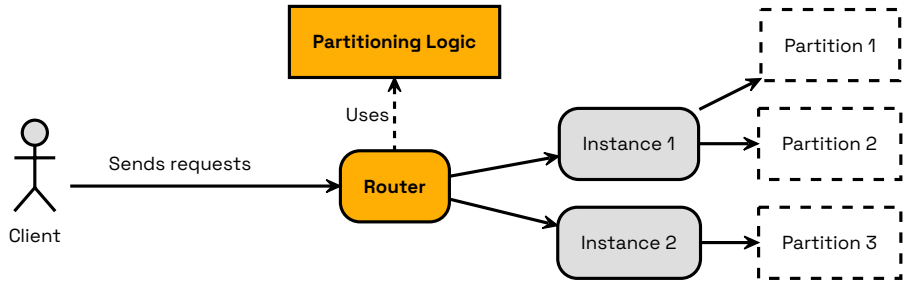

Figure 6.2: Partitioning (Sharding)

Since each partition is managed by exactly one instance, the order of operations is guaranteed within a partition, in contrast to **Competing Consumers** (see § 6.1).

If partitioning data, choose a fixed size of partitions to avoid repartitioning the data if the number of data stores changes [5, p. 219].

6.2.3 Potential issues

Data partitioning logic design can be **complex** and is fundamental to the system's performance. Choosing an incorrect scheme might require **expensive repartitioning**. [3, p. 429]

Since one partition is managed by exactly one instance, additional resiliency measures are needed to handle **failures**.

6.2.4 Example

ExampleEshop (see § 3.4) receives an influx of reviews during major events. These reviews must be scanned for inappropriate content or spam before they are displayed and then compressed, which can take some time if the review contains images or videos. To handle the increased load, the system uses **Competing Consumers** (see § 6.1). However, after introducing the ability to update reviews, they noticed a rise in failures.

This was due to users finding mistakes in their text shortly after posting a review containing a video. These edits only contained the edited text, which was processed much faster by a different consumer and failed to save the edit as the review did not exist yet.

ExampleEshop could fix this by disabling edits before they are screened, but this could result in users forgetting to edit their reviews later, decreasing their overall quality and trustworthiness. Additional logic could be introduced to handle this, but this would have to be managed by the queue. Degrading to one consumer wasn't an option, as the influx of reviews could easily overwhelm it. Instead, they decided to partition the review processors by item. Each item was assigned randomly to a single processor. This way, the system's ability to scale was not altered significantly, but each review event was guaranteed to be processed in order.

6.2.5 Related patterns

- Even though the partitions do not increase the system's overall resiliency per se, this can still be used to one's advantage to prevent cascading failures. Introducing logical partitions for this purpose is known as the **Bulkhead** pattern (see § 7.1)
- An **Ambassador** can used to implement sharding logic on the client side (see § 5.2) [22, p. 22]

6.2.6 Further reading

- Newman on data partitioning in [3, p. 426]
- Joshi's **Fixed Partitions** pattern if the amount of participating data stores can be variable [5, p. 219] and **Key-Range** partitions to optimise for queries with ranges [5, p. 243]
- *Shard (database architecture)* on Wikipedia [81]. Also see [82] for a similar concept in storage systems

6.2.7 Deployment Stamps

An interesting sub-case of partitioning is partitioning users with Microsoft's **Deployment Stamps** [83]. If the use cases of the system allow for creating partitions of users that do not have to share data, creating a distributed system might not be necessary at all (see fig. 6.2.7).

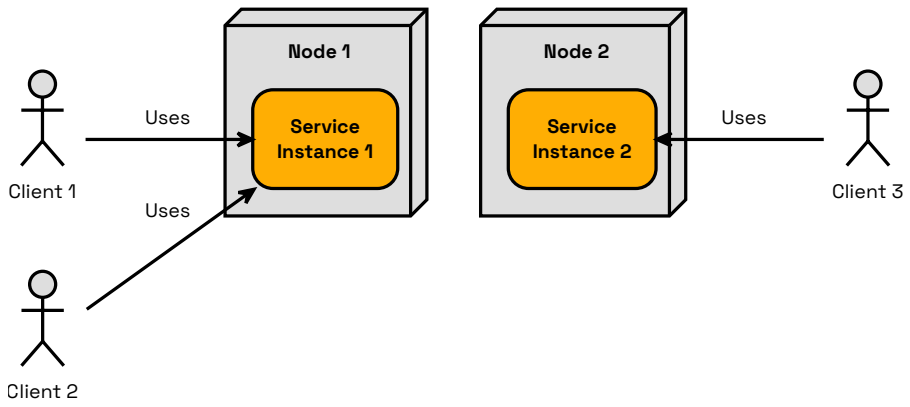

Figure 6.3: Deployment Stamps

Partitions corresponding to geographic regions inherently facilitate **low latency** (until the users travel).

6.3 Scatter–Gather

Asynchronously distribute workloads and aggregate results

This pattern is based on **Scatter-Gather** by Hohpe et al. [4, p. 267, 84] and Deenadayalan [85], and **Scatter/Gather** by Burns [22, p. 73].

See also **Message Broker** (see § 4.2.7).

6.3.1 Context

At least one of the following conditions needs to hold.

1. Many external services need to be queried
2. The problem is *embarassingly parallel*[1]

1. "Embarrassingly parallel" is a term used to describe problems that can be trivially divided into smaller tasks that can be solved independently of each other, which is the ideal scenario for parallelism [86].

3. The operation needs to be run on a whole partitioned data store [22, p. 73]

4. An operation needs to happen with the lowest possible latency

6.3.2 Solution

This pattern has two possible implementations, depending on the use case.

1. The publisher sends out the **same task** using **Publisher–Subscriber** (see § 4.2) to multiple **different processors**

2. The publisher independently sends out a **different task** to multiple **identical processors**

Each processor then processes its task and sends its result back to a central aggregator, which combines the results (see fig. 6.4).

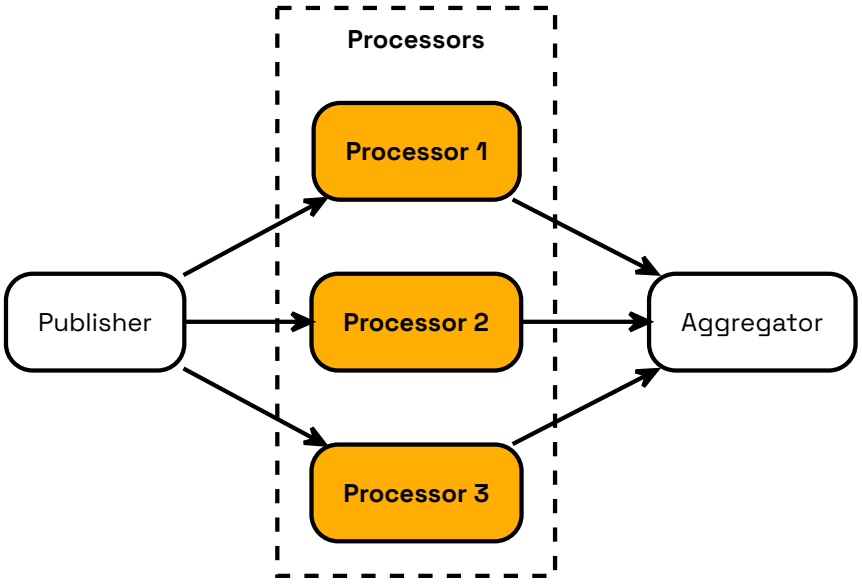

Figure 6.4: Scatter–Gather

Unless this operation is a part of an overall workflow, the aggregator does not need to be a different entity than the publisher.

6.3.3 Potential issues

Depending on the use case, each processor might be a **bottleneck**, and its failure might require retries that might delay the whole operation. Not noticing an error in a processor might lead to **incomplete results** or the **failure of the entire operation**.

The parallelism of a single operation may have a lower limit than its theoretical maximum due to **network overhead** (see § 3.1.1).

See also § 4.2.3.

6.3.4 Example

To improve search capacity, *ExampleEshop* (see § 3.4) partitions its search index into multiple data stores. When a user searches for a product, the search service sends the query to all shards in parallel using **Publisher–Subscriber** (see § 4.2). The search service then aggregates the received results and returns them to the user.

6.3.5 Related patterns

- This pattern combines **Publisher–Subscriber** (see § 4.2) for sending out tasks and **Queue-Based Load Levelling** (see § 7.2) for collecting the results
- It is a natural fit for use with data using **Partitioning** (see § 6.2) when implementing operations over the whole data store
- Use **Queue-based Load Levelling** (see § 7.2) to decouple the processors from the aggregator

6.4 Externalised Configuration

Centralised configuration management

This pattern is based on **Externalized configuration** by Richardson [20, p. 361, 87], **Managed Configuration** by Fehling et al. [15, p. 247, 88], and **External Configuration Store** by Microsoft [89]

The **Control Bus** pattern by Hohpe et al. [4, p. 477, 90] is a similar concept, using messages to control other services

6.4.1 Context

When creating many deployments in a system, it might become challenging to configure or update the configuration of each instance properly. This causes the system to become **inflexible** and increases the probability of **user error**, which can lead to **failures** or **security issues**.

6.4.2 Solution

Centrally manage the configurations of each service. The configuration is either requested by the services themselves when they start or is requested by the infrastructure when creating each service (see fig. 6.5).

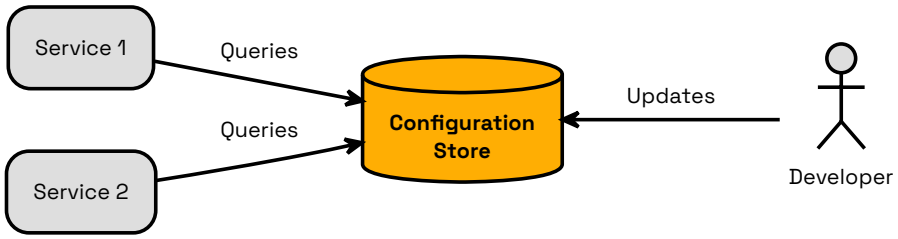

Figure 6.5: Externalized Configuration

This is sometimes called *pull-based externalised configuration* or *push-based externalised configuration* [20, p. 362-363].

6.4.3 Potential issues

The configuration might become a **single point of failure**. If the services request the configuration themselves, changing the configuration store implementation or location might require **changes in all services**.

If the number of services is very high, the services change frequently, or the configuration needs to be updated online, the configuration store could become a **bottleneck**.

6.4.4 Example

ExampleEshop (see § 3.4) employs partitioned data stores for its product catalogue (see § 8.5.4) or its search index (see § 6.2.4). This requires managing many instances, and any changes in configuration, such as after upgrading the database version, would require updating each instance manually. To address this, the system uses a central configuration store. The infrastructure is

configured to pull the configuration from this store on startup, allowing easy updates and ensuring that all instances are running the same configuration.

6.4.5 Related patterns

The **Identity Provider** (see § 9.1) uses the same concept, but is used for accommodating security rather than scalability.

6.4.6 Further reading

- The very similar **Edge Workload Configuration** pattern by Microsoft [91], focusing more on IoT[2] use cases

2. Internet of things

7 Resilience and Reliability Patterns

This chapter presents patterns that combat network unreliability and service instability, both of which are inevitable (see § 3.1.1) in distributed systems. As such, it is the largest chapter in this work.

1. The **Bulkheads** pattern in § 7.1 limits the impact of a failing service on the rest of the system
2. The **Queue-Based Load Levelling** pattern in § 7.2 evens out the load between services
3. The **Retry** pattern in § 7.3 guards against transient failures
4. The **Health Monitoring** pattern in § 7.4 ensures the system is running as expected
5. The **Rate Limiting** pattern in § 7.5 protects services from misuse
6. The **Leader and Followers** pattern in § 7.6 coordinates services in a cluster without a natural leader

Other patterns that could be considered part of this category include [92]:

- Message brokers (see § 4.2.7) can improve communication reliability
- **Circuit Breaker** (see § 8.1) improves overall service reliability during failures of its dependencies
- **Ambassador** (see § 5.2) and **Offload to Gateway** (see § 5.3) present opportunities to implement these patterns
- **Competing Consumers & Load Balancer** (see § 6.1) provide natural backups for failed services
- **Claim Check** (see § 8.4) can help recover message contents in the event of a disaster and **Partitioning** (see § 6.2) or **Identity Provider** (see § 9.1) can limit the amount of information lost
- Chapter 10 discusses patterns to ensure consistency during failures
- **Backend for Frontend (BFF)** (see § 5.4) can be used to personalize reliability for the different requirements of clients
- **Gateway Aggregation** (see § 8.3) can improve reliability when a client is slow or on an unreliable network
- **Strangler Fig** (see § 11.2) improves reliability during transitional periods

Notable omissions in this category due to the methodology described in § 1 include:

- **Cache-Aside** and **Throttling** by Microsoft [93, 94], although common techniques, do not add anything new to the discussion in the context of distributed systems and are not recognised as patterns by other authors
- **Compensating Transaction** by Microsoft [95] is discussed as part of **Sagas** (see § 10.3)

7.1 Bulkheads

Use logical partitions to isolate failures

This pattern is based on **Bulkheads** by Nygard [12, p. 98], later by Newman [3, p. 400] and Microsoft [96].

"Bulkheads" are a concept borrowed from shipbuilding, where, among other things, they are used to divide the inner space into rooms and to prevent a breach in one part of the ship from leaking into the whole vessel [97]. Translating this concept to software architecture means introducing some kind of logical partitions into the system to prevent failures from affecting other parts of the system.

In essence, this is the **Partitioning** pattern (see § 6.2) applied for a different purpose – to increase resiliency instead of accommodating scalability.

7.1.1 Context

There are multiple interconnected parts of an application. A failure in a single part has the potential to cause failures in others, and it is desirable to keep some parts of the application running in such an event.

7.1.2 Solution

Create logical partitions in a system aimed at containing failures (see fig. 7.1). There are several levels where bulkheads can be implemented.

In an individual service, this can mean using **different threads, processes or pools**[1] for different tasks and independently manage their resource limits and priorities to prevent starvation.

1. Pools in this context mean any kind of pooled resource, e.g. worker pools [98]

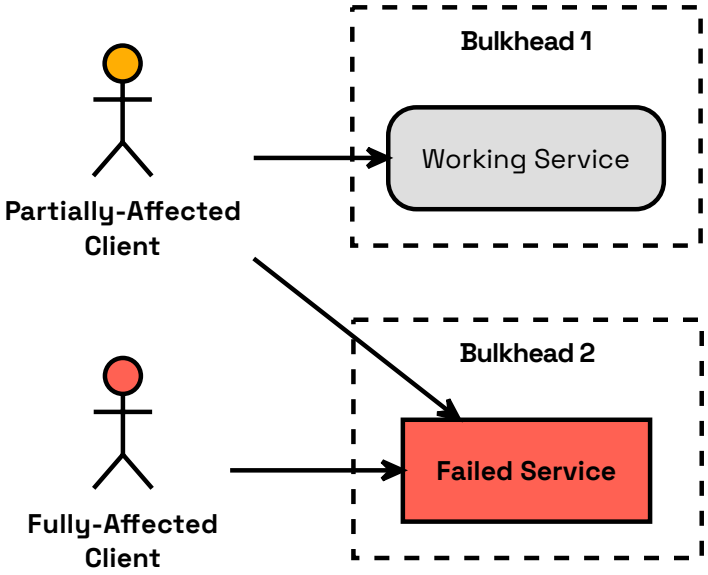

Figure 7.1: Bulkheads

On the component level, this can be done by **avoiding sharing resources** between services (or between clients), i.e. removing single points of failure. This means if there is a failure in one of the resources – or one of its clients overloads it – it will not affect all its clients.

On the deployment level, this can mean running each bulkhead on **different nodes**, isolating hardware failures, or using virtualisation.

At the highest level, this means designing a system with bulkheads in mind, **identifying mission-critical services** and planning for potential failures so that as much of the system as possible continues operating independently – or with limited functionality.

7.1.3 Potential issues

Employing bulkheads more leads to **more resource usage** and **increased complexity**. It may also be difficult to introduce them into a tightly-coupled system.

7.1.4 Example

ExampleEshop (see § 3.4) has defined mission-critical functions that need to be kept running at all costs, as outages would result in significant financial losses. These include basic product page functionality, basic search

and order processing. To achieve this, they are deployed separately from other services and have their dedicated resources. They communicate with dependencies using **Ciruit Breakers** (see § 8.1). If other features such as the recommendation service, full text search or product reviews are unavailable, they continue to function, either by using older, cached data or by providing a degraded experience.

7.1.5 Related patterns

- **Retry** (see § 7.3) might prevent another bulkhead from failing if the failure is only transient
- **Circuit Breakers** (see § 8.1) can be used to automatically "seal" a bulkhead, speeding up fault handling and allowing the affected bulkhead to recover

7.2 Queue-Based Load Leveling

Use a messaging queue to manage and cope with peaks in demand

This pattern is based on **Queue-Based Load Leveling** defined by Microsoft [99], **Queue-Centric Workflow** by Wilder [14, p. 27], **Loose Coupling** by Fehling et al. [15, p. 156] and **Decoupled Invocation** by Rotem-Gal-Oz [13, p. 47]. However, it is just a re-interpretation of the **Point-to-Point Channel** pattern by Hohpe et al. [4, p. 111, 100], focusing on benefits they consider part of messaging in general [4, p. 16].

Nygard [12, p. 73] presents *publish/subscribe messaging* and *message queues* as an alternative to improve the scalability of point-to-point communication but does not elaborate on their differences[2]. Burns [22, p. 117] considers handling bursts an inherent part of the **Work Queue** pattern but instead focuses on applying it for batch processing.

Richardson mostly inherits terminology from [4] and mentions *message buffering* as a benefit of broker-based **Messaging** [20, p. 93]. So does Newman [3, p. 107], but instead embraces the *topic* and *queue* terminology used by brokers [3, p. 135].

This work uses the term **Queue-Based Load Leveling** as it best captures the pattern's intent, as per guidelines defined in § 1.3.

See also **Message Broker** (see § 4.2.7).

2. And later even calls it a *pub/sub bus* [12, p. 211]

7.2.1 Context

The system experiences **variable load** and does not manage to keep up when demand peaks, leading to **dropped requests**.

7.2.2 Solution

Introduce a **message queue** into the service's communication channels (see fig. 7.2). This queue can be used to **buffer requests** and allow the service to process them at its own pace.

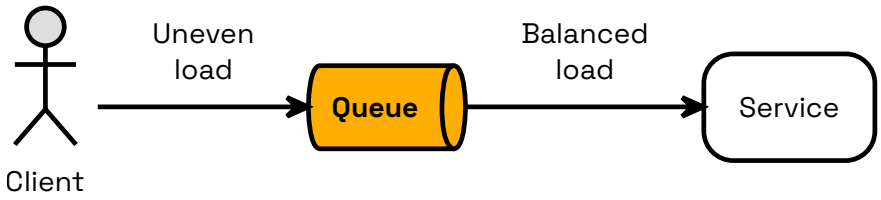

Figure 7.2: Queue-Based Load Levelling

Since the service now does not need to be powerful enough to handle the peak load, its computational resources can be **scaled down**, reducing the cost of the system. Furthermore, the services now have **decreased coupling** as they do not necessarily need to know about each other.

7.2.3 Potential issues

See the potential issues in § 4.2, namely **increased latency**, **additional complexity**, and an additional **point of failure**, when compared to direct communication. Furthermore, using a queue means losing replies without additional measures (see § 4.3).

See also § 4.2.3.

7.2.4 Example

The distributed transaction used for handling orders in *ExampleEshop* (see § 3.4) handles business failures with the **Saga** (see § 10.3) pattern, but this does not handle technical failures. To address this, the system deploys a message broker to handle the communication between services. This broker persists messages until they are successfully processed and communicates with the services transactionally (see § 10.1).

7.2.5 Related patterns

- **Competing Consumers** can be used at the other end of the queue to increase throughput (see § 6.1)
- The queue can implement **Rate Limiting** to limit the rate of requests to protect an underlying service (see § 7.5)
- Can be extended to two-way communication using the **Request–Reply** pattern (see § 4.3)

7.2.6 Further Reading

- The *Inbox and Outbox* pattern can also be used for guaranteed delivery [58]

7.3 Retry

Do not fail because of transient errors

This pattern is based on **Retry** by Microsoft [101], **Retry with backoff** by Deenadayalan [102], **Request-Response with Retry** by Hohpe [103], **At-Least-Once delivery** by Fehling et al. [15, p. 144, 104] and **Request/Reaction** by Rotem-Gal-Oz [13, p. 114], and partly on the **Busy Signal** pattern by Wilder [14, p. 83].

7.3.1 Context

A service connects to an external resource, and this request is either idempotent or has some duplicate handling in place. A failure would negatively affect the currently running operation more than any added latency. The response either signalled the failure is transient explicitly (i.e. with a busy signal [14, p. 83]) or there is a probability it is.

7.3.2 Solution

When a failure occurs, retry the request. However, there are several options on *when* to retry and for *how long*.

If latency is an issue, e.g. an interactive situation, and this type of failure is known to be transient or to resolve quickly, retry **immediately**, with a **fixed delay** or with a **linearly increasing delay**, depending on the balance needed.

If this is a long-running operation, an **exponentially increasing delay** (also known as *exponential backoff* [105]) would be more suitable to avoid collisions or overloading the service. [14, p. 88, 101]

To prevent collisions, consider adding **jitter** to the delay [106].

The service might also estimate how long to wait, e.g. if it employs a **Rate-Limiter** (see § 7.5).

Retry until the failure is recognised as non-transient, or a timeout or a maximum number of retries is reached (see fig. 7.3).

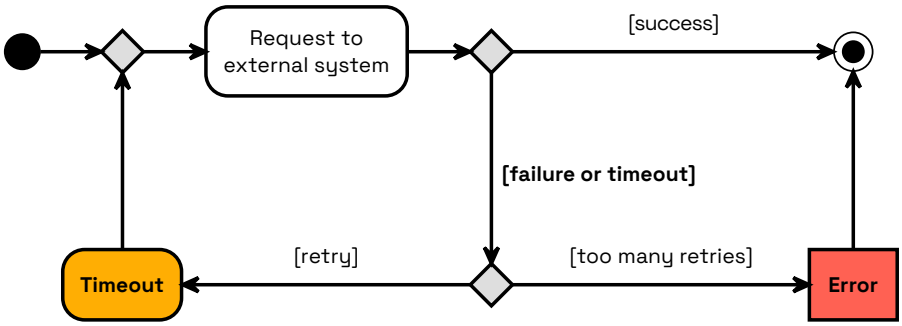

Figure 7.3: Retry

7.3.3 Potential issues

If a service fails due to overload, retrying the request might **exacerbate the problem**. If the retries are long-running, employing an inappropriately-designed backoff algorithm might lead to **collisions** [106] (also known as the *thundering herd* problem [107]).

Depending on the implementation, this pattern might introduce **delays** or **increased load** (especially if the problem is mistakenly categorised as transient). It may also result in **duplicate requests** if the network caused the issues and the idempotency was not implemented correctly. If the calling service has its own consumers, it might be better to let it react to the failure instead [12, p. 93].

7.3.4 Example

ExampleEshop (see § 3.4) sends out transactional emails or newsletters to its users. While this process is not idempotent, sending an email twice is not a critical issue – it is better than not sending it at all, especially if it contains important information such as an invoice or a password reset link.

To address this, the system uses a retry mechanism that resends the email if it fails to be delivered due to network problems or a temporary issue in the recipient's email provider.

7.3.5 Related patterns

- Use a **Circuit Breaker** for avoiding retries if a failure is long-lasting (see § 8.1)
- **Queue-Based Load Leveling** typically implements retry logic (see § 7.2)
- Retries can be implemented in a reusable **Ambassador** (see § 5.2)

7.3.6 Further Reading

- Nygard's thoughts on retries [12, p. 93]
- Newman on timeouts [3, p. 397] and retries [3, p. 399]
- *Exponential backoff* on Wikipedia [105]

7.4 Health Monitoring

Proactively check and react to service failures

Health monitoring is a broad topic covered by many different but related patterns. This pattern is based on the following: **Health Endpoint Monitoring** by Microsoft [108], **Watchdog** by Fehling et al. [15, p. 260, 109], **Service Watchdog** by Rotem-Gal-Oz [13, p. 67], **Health Check API** by Richardson [20, p. 366, 110], and **HeartBeat** by Joshi [5, p. 93, 111].

It is also part of the **Control Bus** and similar to the **Test Message** patterns by Hohpe et al. [4, pp. 477, 498, 112, 113].

7.4.1 Context

The system needs to adhere to **availability or reliability requirements**. Failures in the system need to be **detected** and either **automatically resolved** or **reported**.

7.4.2 Solution

Proactively monitor the health of services (see fig. 7.4). The services either implement a **health check endpoint** or **send heartbeats** to a central service. The fidelity of the health checks can vary from simple *up/down* checks to more complex checks that verify the service's resource usage, response time, internal state or dependencies. This allows to cover a wider range of potential issues than just whether the service is running or not.

In case of a failure or timeout, the system can **automatically respond** by restarting the service, redirecting traffic to a healthy instance, or alerting an operator.

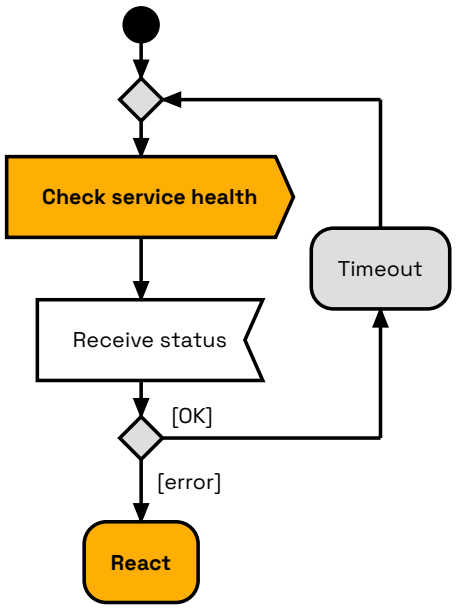

Figure 7.4: Health Monitoring

7.4.3 Potential issues

Imperfectly designed or tuned health checks can lead to **false positives**, **false negatives** or **performance issues**. A check that is too strict can cause unnecessary service restarts, traffic redirection, or performance degradation if it is required too frequently or requires expensive tests.

On the other hand, checks that are too lenient or test too little can lead to undetected issues, rendering the monitoring useless.

They can also introduce security risks by **exposing sensitive information** or creating a vector for **denial-of-service attacks**. This can be mitigated by endpoint authentication or obfuscation [108].

False negatives can also occur due to **failures in the monitoring system** itself. Measures to mitigate this should be taken to ensure correct operation, such as **redundancy** or **self-checks**.

7.4.4 Example

ExampleEshop (see § 3.4) uses health monitoring for all its running services. If the service has an API, it exposes an authenticated `/health` endpoint, which is polled regularly. This endpoint is configured to check the service's dependencies, such as the database or the message broker. For databases, it runs a test query. The system also monitors and logs the response times.

If the service fails, the monitoring system signals the infrastructure to restart it and the load balancer (see § 6.1.4) to not route traffic to it. If it fails repeatedly or is under a long-term high load, the system sends an alert to the operations team. For the parts critical to the operation (see § 7.1.4) of the eshop, it also employs more sophisticated checks that are run less frequently, such as for the search functionality, where it runs a pre-defined query and checks if the results are as expected, or for the order processing, where it places a test order and checks if it is successful.

7.4.5 Related patterns

- Health monitoring can be linked to a **Circuit Breaker** (see § 8.1) to proactively respond to service state instead of relying on requests to fail
- The **Leader and Followers** (see § 7.6) pattern can be used to autonomously detect and recover from failures if the system is dependent on a single coordinating instance

7.4.6 Further reading

- Nygard [12, p. 162] on transparency
- Newman [3, p. 305] on monitoring and observability
- Richardson on designing observable services [20, p. 364]
- Wilder on how to react to node failures in [14, p. 93]

7.5 Rate Limiting

Control the rate of incoming requests to prevent overload or starvation

This pattern is based on **Rate Limiting** by Microsoft [114]. Burns [22, p. 54] mentions this as a denial-of-service defence.

It is sometimes considered as part of an **API Gateway** (see § 4.1.7) [20, p. 262, 3, p. 163].

7.5.1 Context

This pattern can be implemented either server-side or client-side.

1. Server-side – several clients are using a single endpoint. Overuse from one client can lead to **decreased quality of service** or even **starvation** for others – and such a situation is not desired – or **increased costs** for the provider. This can scale from accidental overuse to misuse or even up to coordinated malicious efforts, such as DDoS[3] or Yo-yo[4] attacks.

2. Client-side – a client wants to preemptively **avoid errors** while communicating with an endpoint that is either rate-limited or has a limited capacity and wants to **reduce resource usage** or **improve communication with users**.

7.5.2 Solution

To protect an endpoint, implement a stateful component that keeps track of the number of requests from each client over a specific time period (see fig. 7.5). If a client exceeds their quota, it denies any future requests in the time window and signals this appropriately back to the client. It can also keep the client updated on their current usage so that they can adjust their behaviour accordingly, but it may also withhold that information to prevent coordinated attacks.

There are several approaches that can be used to implement rate limiting, such as simple approaches like *fixed* or *sliding window counters* or classic algorithms like *token* or *leaky buckets* [116]. The best approach will depend on the specific requirements (see [117, pp. 1504–1505] for a brief overview).

Requests over the limit can be either dropped or queued to be processed later. There can also be several tiers of access. Unauthenticated users should

3. Distributed Denial of Service
4. A special type of DDoS designed to over-provision cloud services [115]

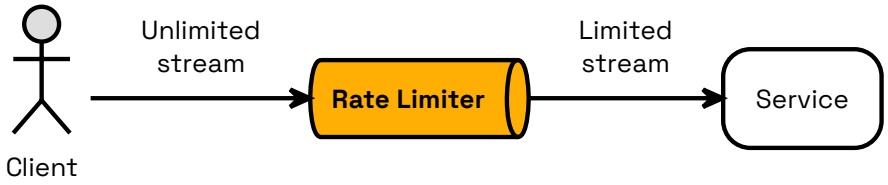

Figure 7.5: Rate Limiting

be rate-limited more strictly, as they can be harder to identify and are easier to scale with attacks [22, p. 54].

7.5.3 Potential issues

Adding a rate limiter **increases complexity** and may introduce **overhead**, **latency**, or **bursts**, depending on the load and implementation. Optimising for throughput may require **specialised hardware**. [117, p. 1505] Faulty implementations may become a **bottleneck** or be **bypassed** by attackers.

Configuration needs to be **carefully tuned** to balance throughput and fairness.

7.5.4 Example

Some of *ExampleEshop's* (see § 3.4) users (or perhaps competitors) employ bots to scrape the site for price changes. This puts a significant load on the system and degrades the performance for legitimate users. To prevent this, the system uses rate limiting to restrict the number of requests a single IP address can make in a given time frame. To reduce false positives, the system tracks the number of requests made by logged-in users separately from anonymous users.

7.5.5 Related patterns

As this pattern serves to protect and may need to be applied in several places, it might be desirable to use a common implementation with **Offload to Gateway** (see § 5.3) or an **Ambassador** (see § 5.2).

A client can use a **Queue** (see § 7.2) to store requests for processing, the estimation in the request replies to plan **Retries** (see § 7.3), or a **Circuit Breaker** (see § 8.1) to fail-fast until the limit is lifted.

7.5.6 Further Reading

- *Wikipedia* [118]
- Wilder [14, p. 87] on reacting to **Busy Signals**

7.6 Leader and Followers

Decentrally appoint a replaceable group leader

This pattern is based on **Leader and Followers** by Joshi [5, p. 85, 119], **Leader Election** by Microsoft [120], and **Ownership Election** by Burns [22, p. 93].

7.6.1 Context

A group of peer service instances with **no natural leader** need to coordinate their actions. However, the group needs to **be resilient** to the leader's failure, and the **leader is replaceable**.

7.6.2 Solution

Each service has one of the following states: *Leader*, *Follower*, or *Looking for Leader* (sometimes called *Candidate*). Each service starts a leader election on startup and does not accept requests until a leader is elected.

Once a leader is elected, it sends **HeartBeats** [see 5, p. 93] to the followers. If the followers do not receive a heartbeat within a specific time, they start a new leader election (see fig. 7.6).

There are a number of algorithms for leader election, such as *Zab* [121] or *Raft* [122].

Alternatively, if there is an implicit order in the group, such as the service age, this can be used to determine the leader without an election, known as the **Emergent Leader** pattern [5, p. 375].

7.6.3 Potential issues

The **added complexity** of leader election may not be necessary, depending on the **availability requirements** of the system.

The leader may become a **bottleneck** if responsible for too much work. If this cannot be mitigated otherwise, one option is to have the leader not participate in the work itself, only in its coordination.

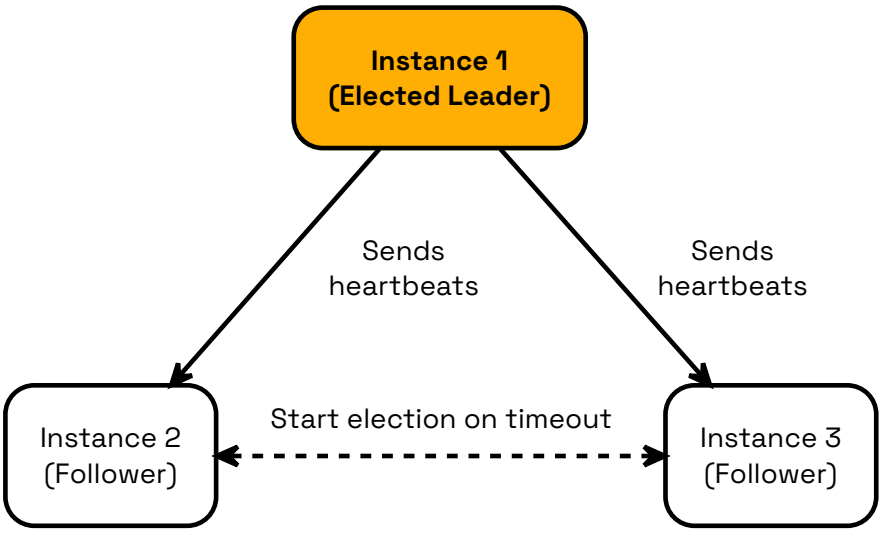

Figure 7.6: Leader and Followers

A fault in the algorithm implementation can lead to **split-brain**[5] scenarios, where multiple leaders are elected.

7.6.4 Example

ExampleEshop (see § 3.4) uses health monitoring to check the status of its services. The system uses multiple monitors to accommodate for possible failures and handle the number of service instances they operate. However, they need to coordinate on how to handle failures so as not to result in conflicts, such as multiple restarts. Furthermore, there is no natural leader, as all monitors are equal. To address this, the monitors use leader election to select a single monitor to react to failures. If this monitor fails, other monitors will elect a new leader.

7.6.5 Related patterns

- **Health Monitoring** for more details on heartbeats (see § 7.4)

5. Rotem-Gal-Oz [13], [p. 66]

7.6.6 Further reading

- Joshi's **HeartBeat** pattern for leader failure detection [5, p. 93, 111], and the **Emergent Leader** pattern [5, p. 375] for avoiding the need for an election protocol
- Nygard [12, p. 305] on using leader elections for achieving *loose clustering*
- Wikipedia [123]

8 Performance and Latency Patterns

This chapter presents patterns that do not necessarily facilitate scalability but solve some specific performance or latency issues introduced by the distribution.

1. The **Circuit Breaker** pattern in § 8.1 improves latency in the event of a failing service so that a service can respond appropriately without having to wait for timeouts
2. The **Colocate** pattern in § 8.2 names a simple fact: the closer two services are, the faster and more reliably they can communicate, and this needs to be considered when designing a distributed system
3. The **Aggregating Gateway** pattern in § 8.3 improves the performance of clients on unreliable or slow networks by doing the heavy lifting for them
4. The **Claim Check** pattern in § 8.4 offloads communication by storing large payloads in a shared location so that they do not need to be handled by message brokers or services that do not need them
5. Under specific circumstances, the **Command and Query Responsibility Segregation (CQRS)** pattern in § 8.5 can improve performance by separating the read and write sides of a service

Other patterns that could be considered part of this category include [124]:

- **Asynchronous Request–Reply** (see § 4.3) can improve concurrency and scheduling of the services by allowing them to respond to requests at their own pace
- **Backend for Frontend** (see § 5.4) enables performance optimisation for specific use cases
- **Choreography** (see § 10.4) can improve performance by removing the bottleneck of a central coordinator
- **Competing Consumers** (see § 6.1) can improve throughput and load distribution
- **Event Sourcing** (see § 10.6) can improve throughput by decreasing contention when writing and better performance of replication
- **Identity Provider** (see § 9.1) removes the need for individual services to handle authentication

- **Health Monitoring** (see § 7.4) can provide resource usage data to load balancers to maximise performance
- **Partitioning** (see § 6.2) can decrease contention for resources and reduce data set sizes

Notable omissions in this category due to the methodology described in § 1 include:

- **Cache-Aside**, **Index Table**, **Materialised View** by Microsoft [93, 125, 126], as they are not specific to distributed systems
- **Compute Resource Consolidation** and **Static Content Hosting** by Microsoft [127, 128] and **Valet Key** by Wilder and Microsoft [14, p. 115, 129] as they are only useful in cloud environments
- **Content Distribution Network** by Fehling et al. [15, p. 300] and the **CDN** pattern by Wilder [14, p. 125], as they are tied to specific products with global infrastructure

8.1 Circuit Breaker

Isolate failure with controlled recovery

This pattern is based on **Circuit Breaker** by Nygard [12, p. 95] and further elaborations by Newman [3, p. 401], Richardson [20, p. 77, 130], Microsoft [131] and Deenadayalan [132].

A circuit breaker is an electronic safety device that protects circuits from overcurrent [133]. This pattern stretches this metaphor slightly, but this work keeps this name as it gives a good intuition for its function and might potentially help with communication with people already familiar with the concept, although the terminology implied by the metaphor can be (unfortunately) misleading for software developers.

8.1.1 Context

A service is communicating with an external resource. In the event of failure, it is more beneficial to fail immediately until it is resolved than to try to get a response eventually.

8.1.2 Solution

Introduce a stateful proxy with the following states (see fig. 8.1).

1. **Closed** – this is the default state. The proxy passes along requests but keeps track of failures.
2. **Open**[1] – enough failures have occurred to *trip* the breaker, "opening" the circuit, immediately returning an error on any requests.
3. **Half-open**[2] – a cooldown timer has run out, and the breaker starts routing requests through again. However, on the first failure, it trips back open and resets the timer.

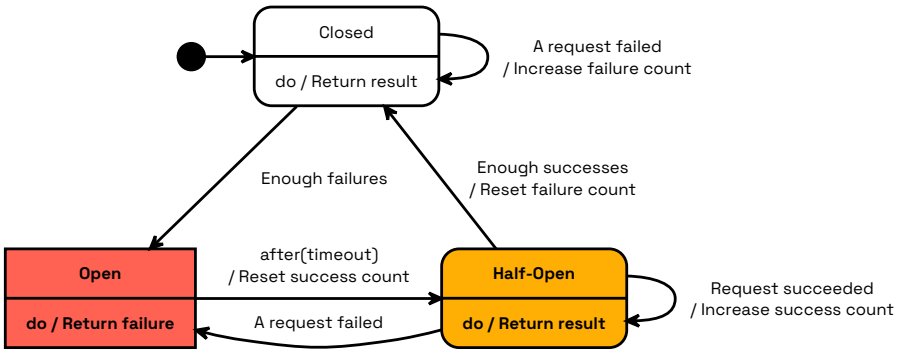

Figure 8.1: Circuit Breaker

Manual operation of the breaker might also be desirable in certain use cases.

8.1.3 Potential issues

While the **additional overhead** circuit breakers introduce is minimal, they can still impact high throughput services and introduce **additional complexity**. The breaker also needs to be **tuned** correctly and specifically to each service, as, if it is not, it might trip just due to transient failures **causing**

1. Coming from software development, this terminology can be very confusing. This work considered using a different naming because an "open" proxy has the completely opposite connotations that it should. However, this would, unfortunately, completely break the metaphor and add confusion to discussions involving people already familiar with this naming scheme.
2. Again, the naming convention is a bit misleading. The breaker is practically closed but will trip open much more easily.

failures, remain in an open state far longer than the underlying service is down, **prolonging downtime**, or switch between open and half-open stats too frequently, **reducing reliability**.

Circuit breakers should only be used for remote resources, as they may add too much overhead for local operations [131].

8.1.4 Example

ExampleEshop (see § 3.4) uses circuit breakers between its bulkheads (see § 7.1). If a bulkhead fails, the circuit breaker trips and isolates the failure. This allows the rest of the system to fall back to a degraded mode, such as using older, cached data or providing a degraded experience, without having to wait for timeouts, which can be critical for business operations[3].

8.1.5 Related patterns

This pattern has an interesting resonance with **Retry** (see § 7.3). If retry logic is placed after a circuit breaker, it may delay the failure too long, causing many more requests to come through and fail before the breaker trips. Retry logic can still be useful before the circuit breaker, e.g. to protect from transient network failures, but an incorrect configuration combination can cause the circuit breaker to trip too often.

As a rule of thumb, the circuit breaker is better in interactive situations, where failing fast might be more important than getting a response. Retrying, on the other hand, is better for long-running operations, where the service might recover in time.

Instead of retrying, **Competing Consumers** can be used to re-route requests to a healthy backup service (see § 6.1). Or, as a compromise, instead of trying to push the request through the breaker, the breaker can actively check the state of the underlying service akin to **Health Monitoring** (see § 7.4) and close itself once the service is healthy again. Staying with the same pattern, the circuit breaker is also an important source of information which can be used for alarms and analysis.

The external service might employ **Rate Limiting** (see § 7.5), which is important to acknowledge when designing the breaker.

If the system employs **Bulkheads** (see § 7.1), the circuit breaker can be used to close off a failing bulkhead and proactively enter a restricted mode of operation.

---

3. Dieulot compiled an interesting list of sources of how even small changes in response times lead to measurable changes in user behaviour [134]

If the implementation needs to be shared among multiple services of incompatible technologies, the circuit breaker can be implemented as part of an **Ambassador** (see § 5.2) or **Offload to Gateway** (see § 5.3).

8.1.6 Further reading

- Wikipedia [135]

8.2 Colocate

Improve network reliability by decreasing the distance between services with high coupling

This pattern was explicitly defined only by Wilder [14, p. 109], but is directly leveraged by other authors in other patterns (see §§ 8.3, 5.1, 5.2 and 11.1).

8.2.1 Context

Heavy traffic between two parties leads to additional network latency and unreliability, which has a significant **performance impact**.

8.2.2 Solution

Decrease the network distance between parties with high coupling (see fig. 8.2). This might include deploying services to the same node, moving services to the same LAN or data centre, or moving services geographically closer to their clients.

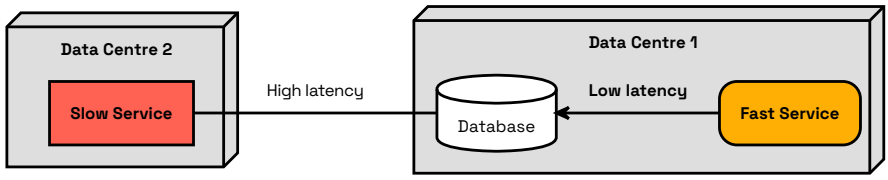

Figure 8.2: Colocate

8.2.3 Potential issues

Moving services together might make the system less resilient to failures, as they might share the same physical resources (see § 7.1).

8.2.4 Example

Searching in *ExampleEshop* (see § 3.4) is a critical part of the system, as it is the primary way users find products. The search index is partitioned across multiple data stores (see § 6.2.4) to handle the load. However, this means that the amount of network traffic needed to search the index is much higher than if the index was stored locally and that the speed of the slowest response dictates the speed of the search. Furthermore, with more partitions, the chance of a network request failing increases. To combat this, they moved the partitioning logic into a separate component that they deployed close to the data stores. This way, most of the network requests are fast and more reliable, which improves the overall search performance and reliability.

8.2.5 Related patterns

This idea of placing services close to each other is a common theme in several other patterns.

1. A **Sidecar** builds upon this pattern to provide additional, language-agnostic functionality to a service by deploying it alongside the main application (see § 5.1)
2. Similarly, an **Ambassador** can be used to handle network communication on behalf of a service, offloading common connectivity tasks (see § 5.2)
3. An **Anti-Corruption Layer**, although it does not necessarily need to be colocated, will still benefit from close proximity to at least one of the involved parties (see § 11.1)

8.3 Aggregating Gateway

Aggregate multiple service requests into a single client response

This pattern is based on Newman's **Central Aggregating Gateway** [3, p. 475], Microsoft's **Gateway Aggregation** [49] and Richardson's **API Composition**, specifically its implementation in a service [20, p. 221].

The last of the roles of a central gateway (defined in § 4.1) is to coordinate and aggregate multiple service responses, but in contrast to **Offload to Gateway** (§ 5.3), this acts for the benefit of the clients. While this pattern is a distributed equivalent of the well-known facade pattern, it has different corollaries and is worth mentioning separately.

8.3.1 Context

Clients need data from multiple services, and there is a need for at least one of the following.

1. **Reduce the complexity** of the client by offloading the aggregation or filtering logic to the gateway.
2. **Reduce network usage** with the client if the amount of data is larger than the aggregate and the gateway, especially if the gateway is near the services.
3. **Decrease latency** and **improve reliability** if the data has to be transmitted over an unreliable or high-latency network, such as over mobile networks, as each new request adds to the latency and increases the likelihood of failure.
4. **Decrease coupling** between the client and the services

8.3.2 Solution

Extend a **Gateway Router** (see § 4.1) with a **Facade** (see [24, p. 175]). Instead of a client sending multiple requests to different services to ask for the resources it needs to construct the result, it asks directly for the result (see fig. 8.3).

To also effectively combat network latency and reliability issues, the gateway needs to have a better link to the services than the client.

8.3.3 Potential issues

Overuse of this pattern has similar effects as the overuse of **Offload to Gateway** (see § 5.3), namely becoming a **development and performance bottleneck** and a **single point of failure**, and handling **too many concerns**.

8.3.4 Example

See the example in **Backend for Frontend** (see § 5.4.4).

8.3.5 Related patterns

- If only no aggregation is needed, **Gateway Routing** can still be used to abstract service locations (see § 4.1)
- After an aggregator is placed, it might be beneficial to offload shared functionality to it using **Offload to Gateway** (see § 5.3) – however, see the note in § 4.1.7

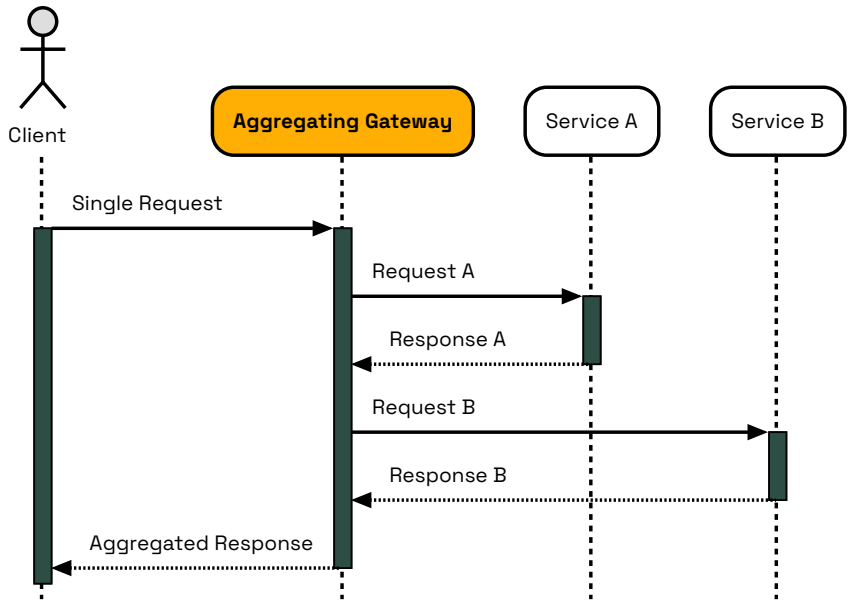

Figure 8.3: Aggregating Gateway

- If the need for server-side aggregation comes from differing needs of clients, **Backend for Frontend (BFF)** can be instead utilised to effectively combat the complexity of the aggregator (see § 5.4)

8.3.6 Further Reading

- Martin Fowler's **Remote Facade** [7, p. 388, 136] is a very similar pattern that applies the same logic but just to one service, aggregating multiple calls to one

8.4 Claim Check

Reduce message sizes by storing large payloads externally

This pattern is based on **Claim Check** by Hohpe et al. [4, p. 305, 137] and Microsoft [138].

8.4.1 Context

Two parties need to pass messages with data payloads through a network, messaging system (see § 4.2.7) or other services. None of these intermediaries need to access the payload, and at least one of the following holds.

- Its size causes **unecessary load** or **resource usage**, causes the intermediaries to become a **bottleneck**, or simply does not fit into the message
- The payload contains **sensitive information** that should not be exposed to the intermediaries

8.4.2 Solution

Add a data store that is accessible to both communicating parties. The producer stores the payload in the store with an associated unique key – the **claim-check token**. It then sends the message with the token instead of the payload. Upon receiving the message, the consumer uses the token to retrieve the payload from the store (see fig. 8.4).

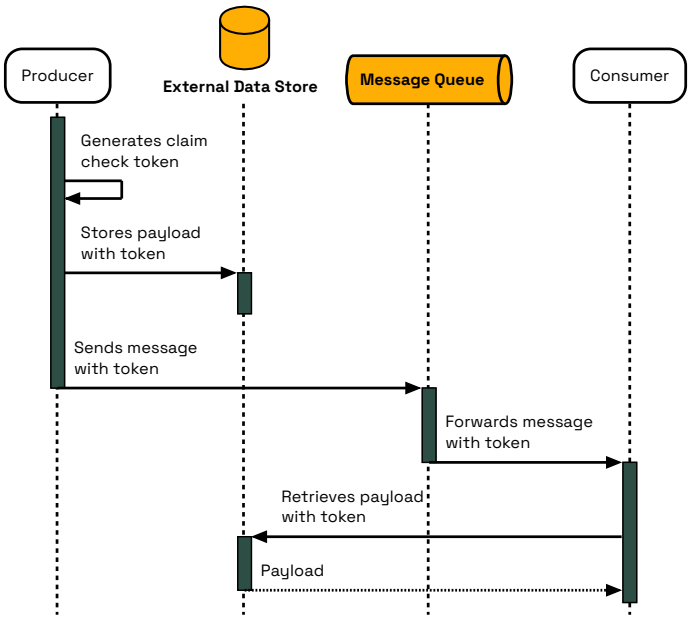

Figure 8.4: Claim Check

The system can also conditionally store the payload based on the size, content, or other criteria to minimize round-trips to the data store if it is not necessary.

8.4.3 Potential issues

Failures may make payloads pile up in the storage system. Strategies for **garbage collection** or **time-to-live** need to be defined. The storage system can also become a **bottleneck** or a **single point of failure** if additional measures are not taken.

This pattern may also require **tuning** to find the balance between throughput and latency and introduce **additional complexity** into the communication.

8.4.4 Example

ExampleEshop (see § 3.4) messaging queues to handle review processing (see § 6.2.4). However, the reviews can contain large files, such as images or videos, which overwhelm the messaging system during large events. To address this, the system saves the files to a separate data store before adding the message to the queue. The message only contains a reference to the file, which allows the system to handle much larger backlogs.

8.4.5 Related patterns

- Another way to use transactional storage is with a **Transactional Outbox** to ensure message delivery (see § 10.2)

8.5 Command and Query Responsibility Segregation (CQRS)

Isolate data store reads and writes to prevent contention

This pattern is based on **Command and Query Responsibility Segregation** by Richardson [20, p. 228, 139], and Microsoft [140]. It was originally defined by Dahan [141] and popularised by Young [142, 143, 144].

8.5.1 Context

1. Reading data is **inefficient**, either because it is aggregated from **many sources**, need **expensive joins** or **transformations**, possibly of many kinds
2. Reads and writes are **asymmetrical**, leading to a need for **separation of concerns** due to performance, security, or resilience (see § 7.1) requirements
3. Reads and writes are **contentious**

8.5.2 Solution

Split the data store and all associated operations into two partitions: **commands** for mutating the data and **queries** for reading the data. Upon processing commands, the command side publishes an event that is eventually consumed by the query side to update the read model (see fig. 8.5).

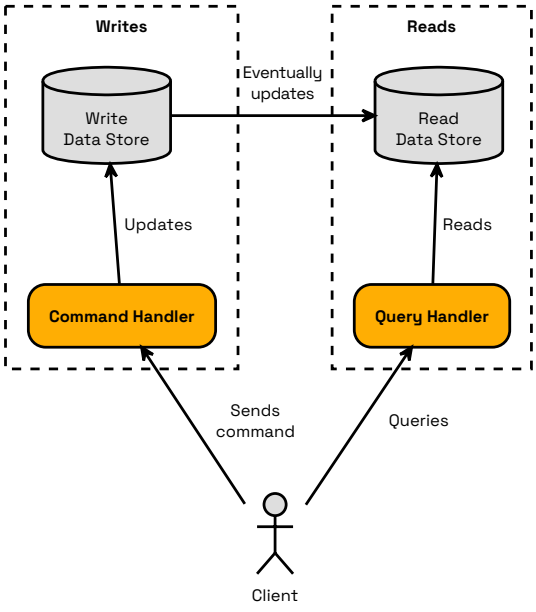

Figure 8.5: Command and Query Responsibility Segregation (CQRS)

The data stores can reside in the same database but can also be physically separated or use different technologies.

The original semantics of the commands focused on the intent of the operation, not the data [141, 142]. Richardson uses a looser interpretation,

even mentioning the command-side services handling all the CRUD[4] operations – including (simple) queries – focusing instead on separate models [20, p. 233]. Others offer this kind of solution as a separate pattern, such as Fowler's **Reporting Database** [145] or Microsoft's **Materialized View** [126].

8.5.3 Potential issues

This pattern is difficult to use well and can add **unneeded complexity**. Consider only using this pattern if the benefits outweigh the risks. For most systems, a CRUD-based approach is both sufficient and more natural. If such, shield the consumers from the underlying implementation. [3, p. 435, 143]

If eventual consistency is employed between the models, it can cause **consistency problems** if a single client updates data and then tries to query it immediately afterwards. Possible solutions include supplying a warning if the data is currently not up-to-date or having the client update a local copy of the data [20, p. 236].

8.5.4 Example

ExampleEshop (see § 3.4) implements its own product catalogue. However, the reads and writes to this catalogue differ, sometimes by orders of magnitude. Furthermore, reads require expensive joins and aggregations, as the product page displays many different kinds of data. Fortunately, there is no requirement for *strict consistency*, as a small delay in updating the product page is manageable.

To address this, the product catalogue uses a separate read database. This database has several prepopulated views of products for different use cases, such as the product page or the search page. Any aggregations, filters or transformations are pre-applied.

8.5.5 Related patterns

This pattern directly facilitates (and is often used with [3, p. 434]) **Event sourcing** (see § 10.6).

8.5.6 Further reading

- Fowler's thoughts on CQRS [143] and Event-Driven Architecture [39]
- Newman's thoughts on CQRS and Event Sourcing [3, p. 434]
- Wikipedia [144]

4. Create, Read, Update and Delete [38]

The read-only nature of the query can be used for several different purposes. Unfortunately, even though their purpose is similar, they differ enough in structure not to be easily merged into a single pattern.

1. Fowler's **Reporting Database** focuses only on the read-only replica to improve performance and scalability (something Richardson sees as a part of CQRS) [145]
2. Richardson's **Command-side replica** moves the read-only store to the client side to relieve stress on a providing service [146]
3. Microsoft's **Materialized View** [126] adds several command handlers, generalising the well-known database feature to distributed data stores

9 Security Patterns

Security is a broad topic and many patterns have been created to address it, such as in [147]. However, while most of them do pertain to distributed systems, few are focused on the architecture of the system itself. This chapter presents a few useful patterns for enhancing the security of a distributed system directly through its architecture.

1. The **Identity Provider & Federated Identity** patterns in § 9.1 shows two common ways to handle authentication and authorization in a distributed system
2. The **Gatekeeper** pattern in § 9.2 instead uses distribution to its advantage to isolate security incidents in a limited environment

Other patterns that could be considered part of this category include [148]:

- **Ambassador** (see § 5.2) and **Offload to Gateway** (see § 5.3) can be used to improve network security, especially in legacy systems
- **Bulkheads** (see § 7.1) can be used to limit the blast radius of a security incident
- **Claim Check** (see § 8.4) can be used to protect sensitive data in communication by hiding them in a secure data store

Notable omissions in this category due to the methodology described in § 1 include:

- **Quarantine** by Microsoft [149], as it is a process, not an architectural pattern
- **Valet Key** by Wilder and Microsoft [14, p. 115, 129] as it is only useful in cloud environments

9.1 Identity provider & Federated Identity

Centralise authentication to reduce the attack surface

This pattern is based on **Identity Provider** by Delessy et al. [150] and Rotem-Gal-Oz [13, p. 91], **Federated Identity** by Delessy et al. [150] and Microsoft [151], and **Access Token** by Richardson [20, p. 354, 152].

9.1.1 Context

Services need to verify the authorisation of requests. Implementing this in each service can lead to the following problems, each decreasing the system's security.

1. **Code duplication** increases the **attack surface**, as each service needs to implement the same logic
2. Users may need to manage **multiple identities**, which reduces usability and increases password fatigue[1]
3. **Added complexity** in user management and access control leads to more **potential security vulnerabilities**

9.1.2 Solution

Implement or employ an **Identity Provider** to centralise authentication and authorisation. Once a user is authenticated, the identity provider issues an access token that can be used to verify the user's identity or more granular permissions (see fig. 9.1). This token can be either:

1. **Opaque** – to verify the authenticity of the token, the service must query the identity provider to check the token's validity
2. **Transparent** – the token is cryptographically signed, allowing the service to verify the token's authenticity independently

The client can retrieve the token from the identity provider directly, or a gateway can facilitate communication (see § 5.3). The identity provider can transitively trust other providers, allowing for **single sign-on** across multiple services (also called a *federated identity*).

9.1.3 Potential issues

Using a single identity provider can lead to a **single point of failure**, both in terms of security and availability. This risk can be lowered by using federated identity providers, but this, in turn, results in the trade-off of **increasing the chance of failure** and having to **trust** multiple, possibly external, providers.

Implementing authorisation in a gateway naively without tokens can lead to the *Confused Deputy Problem*. A Confused Deputy is a specific type of

1. Password fatigue is a widely used term to describe the heavy cognitive load of managing multiple passwords resulting in relaxed security practices [153, 154, 155]

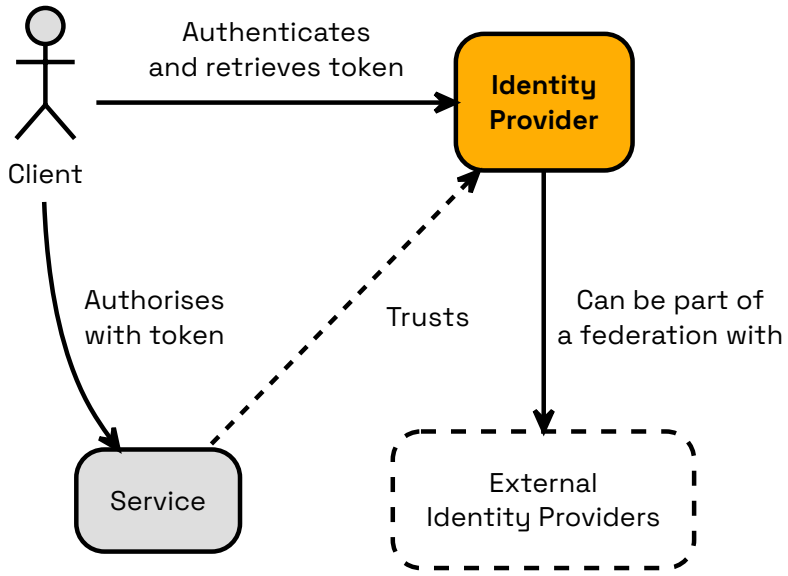

Figure 9.1: Identity Provider & Federated Identity

privilege escalation attack where an intermediary is tricked into performing actions on behalf of an attacker [156, 157]. This can happen as downstream services do not verify the authorisation of requests. Due to this implicit trust, an attacker would only need to trick a single service in a call chain to get access to unauthorised data. [3, p. 380]

9.1.4 Example

ExampleEshop (see § 3.4) offers an employee discount program. To access this program, employees need to log in using their company email address, but this account also needs to be tracked and terminated when the employee leaves the company. However, they do not want to store the whole employee identity database in the eshop, as this would introduce a security risk. Instead, they deploy two identity providers in a federation. The public and private parts of the system use their own providers, but the public one trusts the private one, which allows it access to the employee's name, email and affiliation.

9.1.5 Further reading

- Sam Newman on common single sign-on implementations in *Building Microservices* [3, p. 377]

- *Identity providers* [158] and *Federated identities* [159] on Wikipedia

9.2 Gatekeeper (Service Firewall)

Protect services by validating and sanitising incoming requests in a limited environment

This pattern is based **Service Firewall** by Rotem-Gal-Oz [13, p. 35] and **Gatekeeper** by Microsoft [160]. It can be seen as a variation of **Firewall Proxy** by Buschmann et al. [54].

This work adopts the newer **Gatekeeper** name to provide a distinction between traditional firewalls.

9.2.1 Context

A service or a resource needs to be **secured from malicious requests** from external clients, as a vulnerability could potentially give an attacker access to **sensitive internal resources** or **disrupt the service**.

9.2.2 Solution

This situation can be improved by deploying a gateway service to a limited, isolated environment. This service acts as an intermediary between the clients and the internal services, handling all incoming requests (see fig. 9.2).

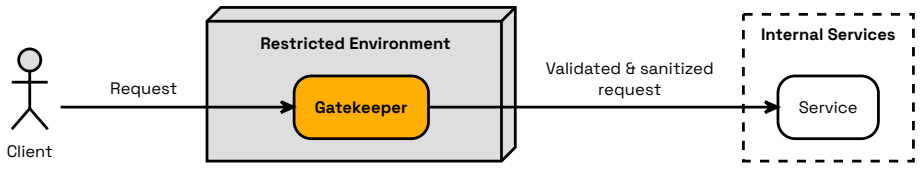

Figure 9.2: Gatekeeper (Service Firewall)

The gatekeeper validates and sanitises the requests, depending on the requirements. It does not perform any additional processing. When encountering a problematic request, it can be audited and logged for later analysis.

If the gatekeeper gets breached, the environment limits the *blast radius* of the attack, as the internal services and resources are shielded from direct access.

9.2.3 Potential issues

The gatekeeper introduces **additional complexity** and **latency** to the system, which can **decrease performance**, depending on the implemented functions. It can also act as a **single point of failure** or **bottleneck** if no additional measures are taken to ensure its availability.

9.2.4 Example

ExampleEshop (see § 3.4) allows users to create accounts or post reviews. This means accepting and storing user-generated content, which can lead to security risks. To address this, the system uses gatekeeper services to validate all content entering the system. For instance, it sanitises all text to defend against XSS attacks[2], or checks for known vulnerabilities in uploaded files.

It also includes a rate limiter to prevent abuse and a honeypot[3]. Since it exists in an isolated environment and sees only a limited number of (secured) endpoints, the gatekeeper acts as a first line of defence against attacks. This limited environment can be easily monitored and audited, so the attacker does not automatically get access to secrets or sensitive data in the event of a breach.

9.2.5 Related patterns

- This pattern can be seen as a specialisation of **Offload to Gateway** (see § 5.3) or **Ambassador** (see § 5.2)

2. Cross-site scripting (XSS) is a (slightly confusing) name for a type of security vulnerability that enables JavaScript injection, thereby being able to manipulate the content of a web page or access sensitive data [161]

3. A honeypot is a security mechanism designed to look like a vulnerable part of the system to attract attackers and log their actions [162]

10 Consistency Patterns

This chapter details patterns that help to ensure consistency in distributed systems.

1. The **Transaction-Based Processor** in § 10.1 expands well-known database transaction to message processing
2. The **Transactional Outbox** in § 10.2 ensures messages are sent if a database is updated
3. The **Saga** pattern in § 10.3 shows a way how to manage distributed transactions, a notoriously difficult problem. It has two possible implementations:
    1. **Choreography** in § 10.4 where services communicate directly
    2. **Orchestration** in § 10.5 where a central service manages the process
4. The **Event Sourcing** pattern in § 10.6, a well-known pattern that can also be used to simplify consistency in an event-driven architecture

Other patterns that could be considered a part of this category include:

- **Queue-Based Load Levelling** (see § 7.2) which can be used to achieve guaranteed delivery of messages
- **Publisher–Subscriber** (see § 4.2) to facilitate data replication when using event sourcing
- **Leader and Followers** (see § 7.6) which is commonly used in data replication [5, p. 85]
- **Command and Query Responsibility Segregation (CQRS)** (see § 8.5) which can be used to implement event sourcing, but also to introduce eventual consistency

Notable omissions in this category due to the methodology described in § 1 include:

- **Data Abstractor** by Fehling et al. [15, p. 194], which is not mentioned by any other authors
- **Command-side replica** by Richardson [146] for the same reason
- **Domain Event** by Richardson [20, p. 160] as it does not pertain to architecture

10.1 Transaction-Based Processor

Use transactions to encapsulate atomic processes

This pattern is based on the **Transaction-Based Processor** by Fehling et al. [15, p. 201, 163], which in turn generalises the **Transactional Client** by Hohpe et al. [4, p. 431, 164] and database transactions.

10.1.1 Context

Reading data from message middleware or updating data in a database transactionally is not enough, as it must depend on an associated process.

10.1.2 Solution

The transaction is extended to encompass the associated process.

1. The message is only deleted from the queue once the operation is successful

2. The changed data is only committed to the database if the operation is successful

This extends the ACID properties of the transaction to the associated process (see fig. 10.1).

10.1.3 Potential issues

This pattern requires support in the message middleware and the database system.

10.1.4 Example

The distributed transaction used for handling orders in *ExampleEshop* (see § 3.4) uses messaging queues (see § 7.2.4) to reliably communicate between services and handle backlogs of messages. However, a service could still fail while processing a message, e.g. the warehouse system could crash while handling a reservation, and this reservation would be lost on a restart. To account for this, the services communicate with the broker transactionally. When receiving a message, the broker does not remove the message from the queue; it only hides it and only removes it once it confirms that the message was processed successfully.

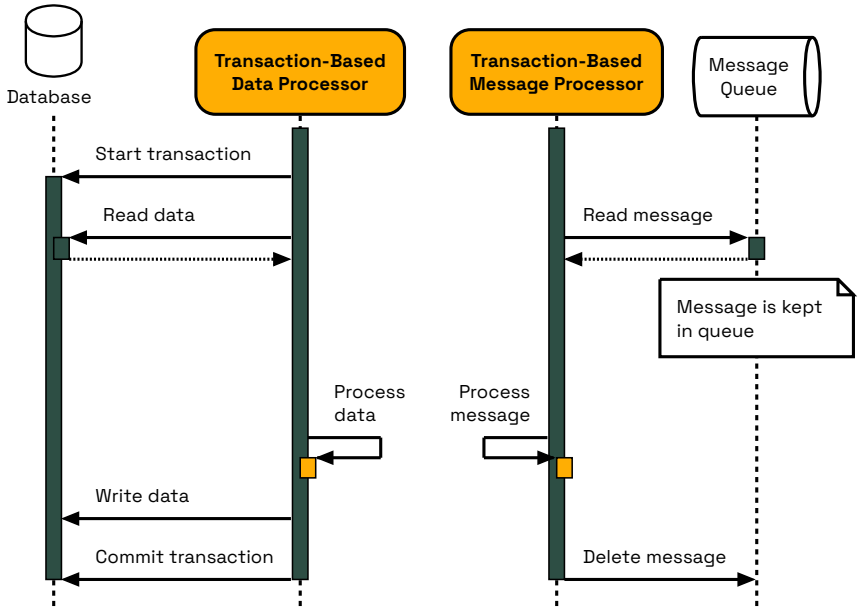

Figure 10.1: Transaction-Based Processor

10.1.5 Related patterns

The **Transactional Outbox** combines these two approaches, guaranteeing a message is sent if a database is updated (see § 10.2).

10.2 Transactional Outbox

Atomically send a message and update the database

This pattern is based on the **Transactional Outbox** by Richardson [20, p. 97, 165] and Deenadayalan [166].

10.2.1 Context

A message and a database operation need to be done as part of the same transaction – the message must be sent iff the operations succeed – and a *two-phase commit*[1] cannot be used, either because of incompatibility with the underlying technologies, payload differences, or availability concerns [20, p. 112, 169].

1. A two-phase commit is a protocol that ensures that all participants in a distributed transaction agree on whether to commit or abort the transaction [see 5, p. 257, 167, 168].

10.2.2 Solution

Store the message in the database as a part of the transaction in an *outbox table* (see fig. 10.2). An *outbox processor* reads messages from the outbox table and sends them to the message broker. The processor marks the messages as sent or deletes them to avoid reprocessing.

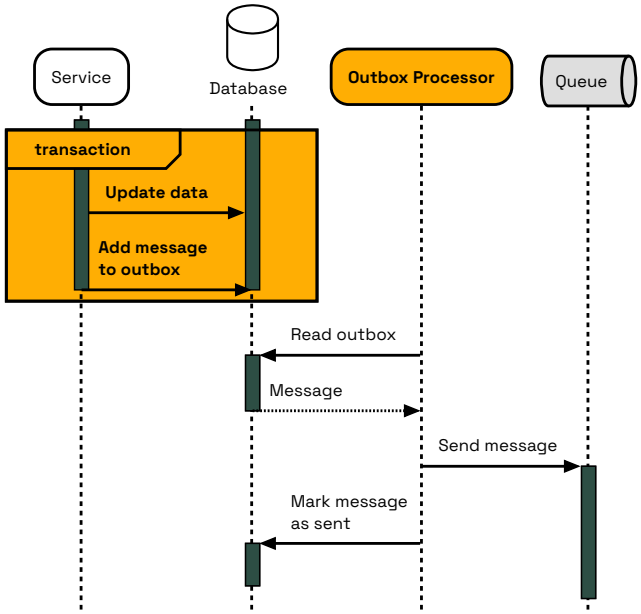

Figure 10.2: Transactional Outbox

Richardson highlights two methods for implementing the outbox processor:

1. The **Polling publisher** periodically polls the database, trading simplicity for increased load and traffic [20, p. 98, 170]
2. The **Transactional log tailing** is a more efficient method which reads the database's commit log and propagates that instead [20, p. 99, 171]

10.2.3 Potential issues

Depending on the implementation, this can cause **increased database load**, which can cause it to become a **bottleneck**. It also introduces **additional complexity** due to a need for an extra processor and **additional latency** in message delivery.

This pattern does not prevent **duplicate messages**; handling those requires additional measures.

10.2.4 Example

Example eshop (see § 3.4) needs to send a confirmation email when users register. If the confirmation email has not been delivered but the account has been created, the user might need to ask for a password reset. To avoid this, the service handling registration writes a message to an outbox as part of the user creation database transaction. The message is not removed from the outbox until the service receives a confirmation that the email has arrived or that the email address does not exist.

10.2.5 Related patterns

- If the transaction needs to span multiple services, use the **Saga pattern** (see § 10.3)
- If there is no update to the database, using **Queue-based Load Leveling** for guaranteed delivery might be more efficient (see § 7.2)

10.2.6 Further reading

- **Inbox and Outbox** pattern [58]

10.3 Saga

Sacrifice isolation for increased availability during a distributed transaction

This pattern is based on **Saga** by García-Molina et al. [172]. Although originally intended to implement long-living transactions (LLTs)[2], it has been interpreted for use in distributed systems in general, see Rotem-Gal-Oz [13, p. 129], Richardson [20, p. 114, 173], Newman [3, p. 182], Richards et al. [2, p. 261], Microsoft [174], and Deenadayalan [175].

10.3.1 Context

Multiple services need to participate in a distributed transaction of a business process, but other ACID algorithms, such as a two-phase commit, are

2. Transactions locking resources in a database for a long time, causing contention

not an option (see the context of § 10.1 for more details). Not having the transaction isolated is an acceptable trade-off.

10.3.2 Solution

Break the long-lived transaction into smaller short-lived transactions. In the case of a business failure, the system can apply either:

1. **Forward recovery**, by retrying the transaction from a particular *save-point*, if the failed operation was idempotent
2. **Backward recovery**, by running *compensating transactions* to undo the operations that were already performed
3. Combine both approaches based on the context (see fig. 10.3) [3, p. 189]

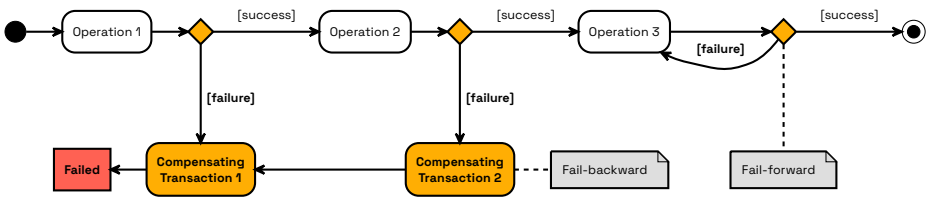

Figure 10.3: Saga

Executing steps with a high probability of failure can decrease the amount of compensating transactions needed in case of a failure [3, p. 188].

There are two possible ways to implement a saga: **Choreography**, an event-driven approach suitable primarily for simpler sagas (see § 10.4), or **Orchestration** (see § 10.5). It may also be possible to nest these two styles based on the requirements of the process [3, p. 194, 176].

10.3.3 Potential issues

If possible, always avoid distributed transactions, as it introduces a large amount of **additional complexity** to the system [3, p. 181, 2, pp. 132, 260, 20, p. 114].

Not all business operations can be fully rolled back. The compensating transactions thus have to be *semantic rollbacks*, i.e. roll back enough in the context of the saga [3, p. 187]. These can be **difficult to design** [177, 18:20].

Since the saga is only ACD[3], **isolation needs to be handled separately** [see 20, p. 126 for common approaches].

A common misconception is that a saga can handle technical failures. Different mechanisms are needed to handle such cases, such as **Retry** (see § 7.3) or sacrificing availability, e.g. with a two-phase commit [178].

10.3.4 Example

Order processing in *ExampleEshop* (see § 3.4) is a complex process that involves multiple services. When a user places an order, the system needs to check the availability of the products, reserve them, charge the user, notify the warehouse, and send a confirmation email. If any of these steps fail, the system needs to roll back the changes and notify the user.

The steps in the saga are ordered to minimize the compensating transactions needed by running the step most likely to fail first.

1. Charge the user (checkpoint)
2. Send a confirmation email
3. Reserve all products
4. Send them out for delivery

The payment marks a checkpoint from which the system proceeds with forward recovery. If not all products are in stock, the system can notify the user and retry once the products are available again. Alternatively, if the customer decides to cancel the order, the system can undo the reservation and issue a refund.

10.3.5 Related patterns

- **Event Sourcing** (see § 10.6) can be used to implement compensating transactions [2, p. 132]

10.3.6 Further reading

- **Compensating Transaction** by Microsoft [95]
- *Long-running Transactions* [179] and *Compensating Transactions* [180] on Wikipedia

3. Missing the "I" from ACID (see § 3.1)

10.4 Choreography-Based Sagas

Distribute responsibility for a saga across participating services

This pattern is based on **Choreography** by Richardson [20, p. 118, 173], Newman [3, p. 192], Microsoft [174] and Deenadayalan [181].

Hohpe et al. [4, p. 284] consider this a *distributed* **Pipes and Filters**.

10.4.1 Context

A **Saga** needs to be implemented (see § 10.3), and the interactions between services are **simple or few**. The services need to be autonomous and have local control. A central orchestrator (see § 10.5) would be an unwanted **single point of failure** in the system.

10.4.2 Solution

Services pass each other events using **Publisher–Subscriber** (see § 4.2) or **Queue-Based Load Levelling** (see § 7.2). They move the transaction forward on receiving successful events and roll back on receiving failure events (see fig. 10.4).

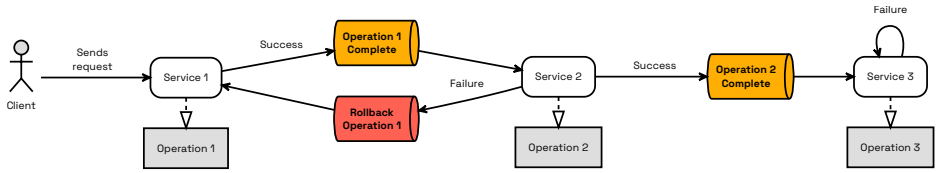

Figure 10.4: Choreography-based Saga

10.4.3 Potential issues

Using choreography means no single place documents the entire process, which may make it **hard to understand**. This can make it **unsuitable for complex sagas**.

It can also make it harder to **monitor the state** of the process – one solution is to use a **Correlation Identifier** [4, p. 154, 63] and log all events passed through the system [3, p. 193].

Regarding development, choreography can lead to **cyclic dependencies** or **tighter coupling** between services [20, p. 121].

10.4.4 Example

ExampleEshop (see § 3.4) screens and compresses attachments in a review. This process is composed of several steps and needs to be separated into different services, either because of security (see § 9.2.4) or different hardware requirements.

1. The review attachment repository saves the attachment and creates a new message with a **Claim Check** (see § 8.4) token
2. The spam and inappropriate content filter checks the attachment
3. The image processing service compresses the attachment
4. The review is added to the database

The services communicate with each other using a message broker (see § 4.2) and subscribe to the events needed to move the process forward (see fig. 10.5).

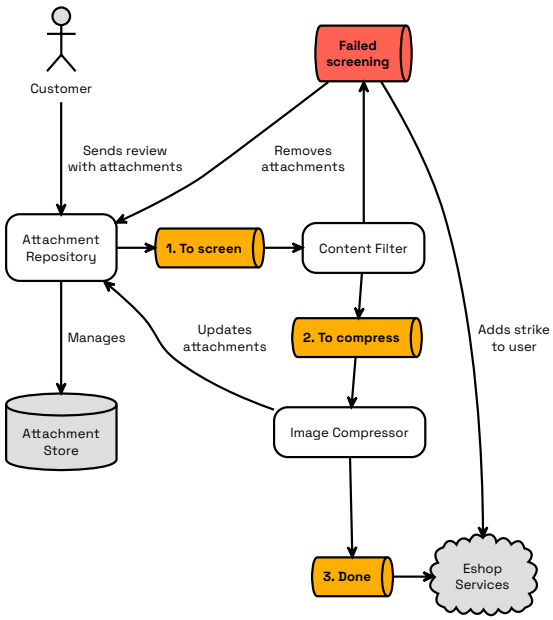

Figure 10.5: Product review processing using a choreographed saga

10.4.5 Related patterns

- **Orchestration** if the interactions are complex or there are many services (see § 10.4)

10.5 Orchestration-Based Sagas

Centralise responsibility for a saga in a single service

This pattern is based on the original behaviour of **Sagas** [172], and defined as **Orchestration** by Rotem-Gal-Oz [13, p. 170], Richardson [20, p. 121, 173], Newman [3, p. 189], Azure [174], and Deenadayalan [182].

Process Manager by Hohpe et al. [4, p. 278], **Workflodize** by Rotem-Gal-Oz [13, p. 35], and **Scheduler Agent Supervisor** by Microsoft [183] implement a similar concept but do not discuss failure management; if such they imply failing-forward.

10.5.1 Context

A **Saga** (see § 10.3) needs to be implemented, but the interactions between services are **complex**, **many in number**, or **need to be independent**. It is desirable to have the workflow defined and monitored in a single place.

10.5.2 Solution

Implement or appoint a central **Orchestrator** that operates a state machine describing the saga (see fig. 10.6). The orchestrator **sends commands** to the services involved in the saga, instructing them on what to do next. The services then **respond** with the outcome of the operation.

10.5.3 Potential issues

Using an orchestrator may lead to **increased coupling** between it and the services. Implementing it as a new service to manage the saga leads to **increased complexity**. The orchestrator can become a **single point of failure** and introduces a risk of **centralising business logic** in a single service.

10.5.4 Example

As the order processing in *ExampleEshop* (see § 3.4) involves a complex process and multiple services, they decided to use a central orchestrator to manage the process. The orchestrator is responsible for coordinating the services and triggering compensating transactions in the case of a failure by implementing a state machine. This way, the process is readable and easily changed or extended. As it waits for external services for most of the process, the orchestrator uses asynchronous programming to run multiple

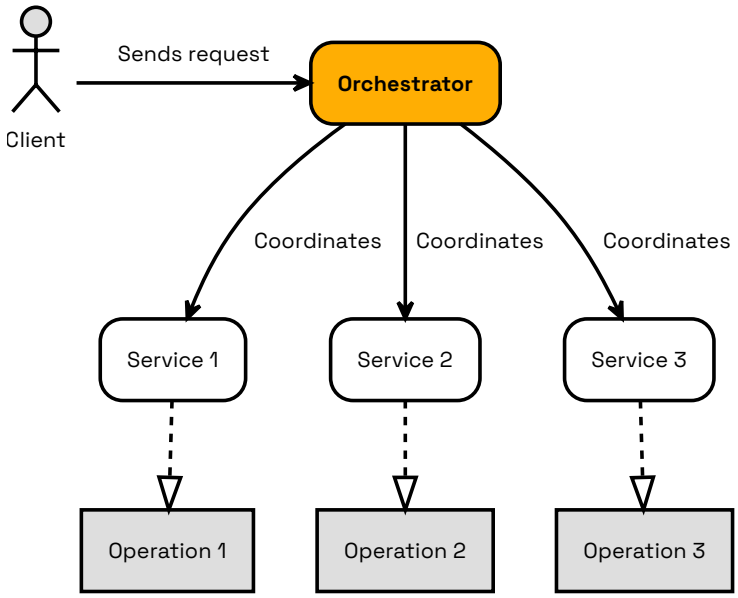

Figure 10.6: Orchestration-based Saga

sagas concurrently. It also logs each step (see § 10.6.4) in the process, which can be used for debugging or analytics.

10.5.5 Related patterns

- **Choreography** if the number of services is small (see § 10.4)

10.6 Event Sourcing

Store data as a sequence of changes to preserve history

This pattern is based on **Event Sourcing** by Fowler [184], later by Richardson [20, p. 184, 185], Newman [3, p. 434], Microsoft [186] and Deenadayalan [187].

10.6.1 Context

See the context defined in 10.2.1. Furthermore, the application would benefit from storing the entire history of state changes due to one of the following reasons [184].

1. The history needs to be (or would benefit from being) **auditable** or **analysed**, and as such there needs to be a high level of trust in the log
2. It needs to be able to **reconstruct the state** at any given time or analyse the impact of changes
3. The application would benefit from running **in-memory** due to performance requirements

10.6.2 Solution

Implement the **Command and Query Responsibility Segregation (CQRS)** (see § 8.5) pattern. On top of it, store each event change as a separate record in an append-only event store (see fig. 10.7).

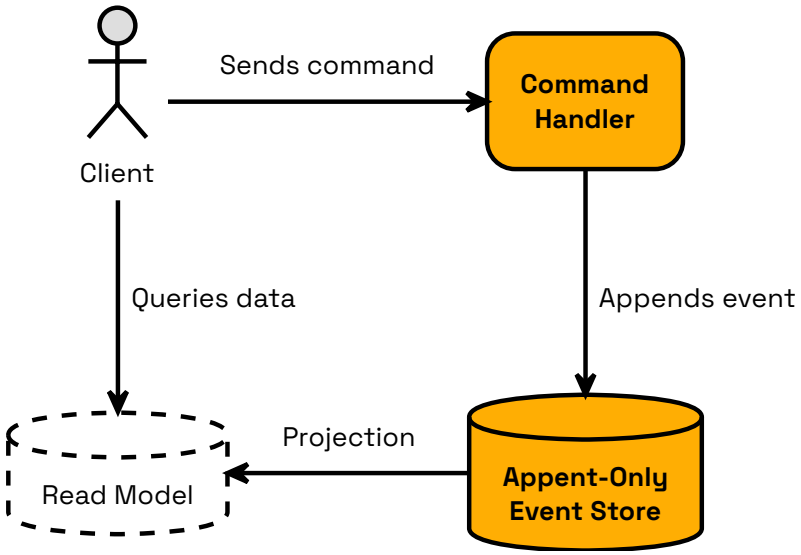

Figure 10.7: Event Sourcing

10.6.3 Potential issues

Working with an event-sourced system may be unfamiliar and thus have a **learning curve**. Communication with external system needs to be carefully handled if needing to replay the history, such as deterministic **identifier generation**. **Schema evolution** and **deleting data** can also be challenging.

10.6.4 Example

ExampleEshop (see § 3.4) employs a backend system to manage products. On change, this system triggers a rebuild of the read-only database replicas (see § 8.5.4). Since rebuilding the whole database is expensive, the system only sends the changes to the replicas. However, an issue arises when a replica fails, as without a history of the changes, it is hard to tell which changes have already been applied, and a full rebuild has to be triggered, prolonging the downtime.

To address this, the system uses event sourcing, as it is already using CQRS. Each replica holds the event it has processed last, and when it fails, it only needs to request the events it has missed. This way, the system can recover much faster. This also has business benefits, as the system can now provide a history of changes, which can be used for analytics or to recover from user errors.

Another point where the system uses event sourcing is in the order processing orchestrator (see § 10.3.4). This allows for easy transactional messaging between services, as the orchestrator can simply replay the events if a service fails, and simplifies debugging in the event that a transaction fails, which is important as a problem in the order processing can result in significant financial losses.

10.6.5 Related patterns

- **Command Query Responsibility Segregation (CQRS)** can facilitate event sourcing by splitting read and write operations beforehand (see § 8.5)

10.6.6 Further Reading

- Sam Newman's thoughts on CQRS and Event Sourcing [3, p. 434]
- *Event-driven architecture* on Wikipedia [188]

11 Migration and Compatibility Patterns

This chapter presents patterns for working with legacy, external or incompatible systems.

1. The **Anti-Corruption Layer** pattern in § 11.1 is a way to isolate somehow *corrupted* (i.e. poor or incompatible) design of other services
2. The **Incremental Replacement (Strangler Fig)** pattern in § 11.2 shows how to replace a legacy system piece by piece
3. The **Messaging Bridge** pattern in § 11.3 connects two systems with incompatible messaging middleware

Other patterns that could be considered part of this category include:

- **Ambassador** (see § 5.2) and **Sidecar** (see § 5.1), which can be used to extend the functionality of (legacy) services

Other patterns for migrating monolithic systems are already described in depth by Newman [189].

11.1 Anti-Corruption Layer (ACL)

Mediate between modern and legacy systems

This pattern is based on the **Anticorruption Layer**[1] defined by Evans [190, p. 364], later by Richardson [20, p. 446, 191], Deenadayalan [192] and Microsoft [193].

It has a similar notion to the **Adapter**[2] pattern by Burns [65, later 22, p. 31], although the latter focuses predominantly on its usage in regards to enabling **Health Monitoring** (see § 7.4) and is deployed as a **Sidecar** (see § 5.1).

Fowler's **Gateway** [7, p. 466] can be seen as an implementation of this pattern [194, 195].

1. Not to be confused with Access Control Lists, which have the same acronym
2. Not to be confused with the well-known **Adapter** pattern by Gamma et al. [24]

11.1.1 Context

An application needs to work with an external system that satisfies one of the following conditions.

1. It uses outdated technologies that would bring in unwanted dependencies or are incompatible
2. It has a poorly designed (*corrupted*) API
3. It is impractical to change (e.g. a legacy system)
4. It uses different model semantics

11.1.2 Solution

Implement an anti-corruption layer that isolates the two systems, acting as an intermediary that translates between them.

This layer can be a separate software or a different service (see fig. 11.1).

Figure 11.1: Anti-corruption layer (ACL)

11.1.3 Potential issues

Any data transformation inevitably causes increased **increased latency**. When scaling the system up, the ACL may become a **bottleneck** or a **single point of failure**. If the underlying service is scalable, the ACL has to be designed with that in mind.

If the ACL is implemented as an additional service, it adds **maintenance cost**. Furthermore, if it is a short-term solution, the ACL needs to be tracked as **technical debt**.

Lastly, there is a risk of **persisting corruption** – if the API of the ACL is poorly designed, the "corruption" will be propagated anyway.

11.1.4 Example

ExampleEshop (see § 3.4) uses an old warehouse system to manage its inventory. However, this warehouse system is old, uses legacy technologies and has

a complex API in a foreign language. To decrease coupling and to prevent having to pull in legacy dependencies, the eshop uses an anti-corruption layer to translate between the two systems. This layer provides an API that uses the same semantics as the rest of the system, so developers working on the eshop do not need to know the intricacies of the warehouse system. This also means that once this system is replaced, the rest of the system will not need to be updated.

11.1.5 Related patterns

- The ACL can be deployed as a **Sidecar** for increased performance (see § 5.1)
- The **Incremental Replacement (Strangler Fig)** pattern can be used to phase out the underlying system and ACL gradually (see § 11.2)
- A **Messaging Bridge** can be used if systems use incompatible messaging systems (see § 11.3)

11.1.6 Further Reading

- If there is no underlying issue with the system you want to adapt, the **Facade** [24, p. 175, 196], **Adapter** [24, p. 139, 197] and **Mediator** [24, p. 273, 198] patterns may still be applicable
- Fowler's example on how to use the anti-corruption layer to refactor code working with external services [194]

11.2 Incremental Replacement (Strangler Fig)

Replace a legacy system step-by-step

This pattern is based on the **Strangler Fig Application** originally defined by Fowler [199, later 200], later by Richardson [20, p. 430, 201], Newman [189, p. 79, 3, p. 79], Deenadayalan [202] and Microsoft [203].

Fowler later renamed the pattern to **Strangler Fig**, as it was commonly shortened to *strangler application* – which had a violent connotation, different to the original plant-based metaphor [204] – in hopes of strengthening the original intention of the name [199].

In this spirit[3], this work primarily refers to this pattern with the more descriptive **Incremental Replacement**, avoiding the metaphor altogether so as not to depend on the reader's prior botanical knowledge.

11.2.1 Context

For whatever reason, a legacy system needs to be replaced, but a *big bang*[4] release would bring unnecessary risks.

11.2.2 Solution

Add **Gateway Router** (see § 4.1) before the legacy and new applications. Initially, the router forwards all requests to the legacy system. Over time, it filters out requests that can be handled by the new system, gradually replacing the legacy system with the new one (see fig. 11.2).

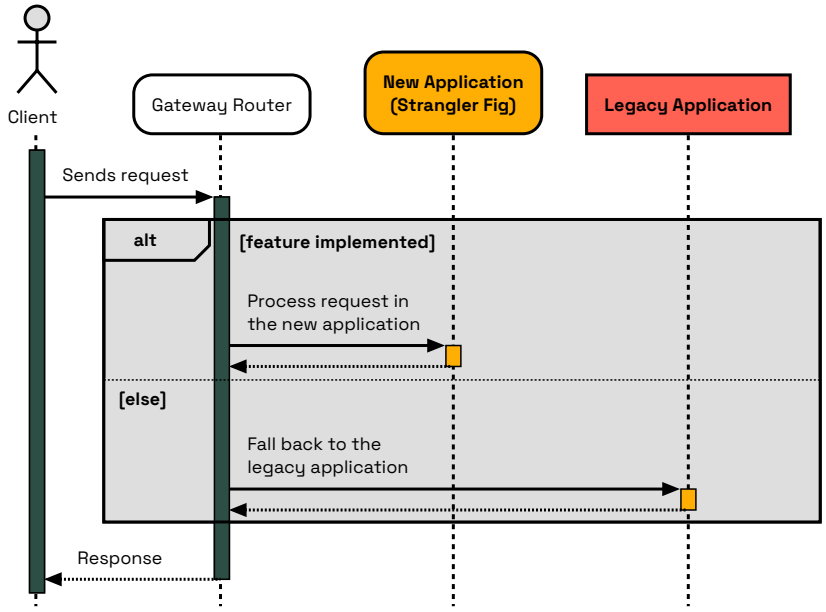

Figure 11.2: Incremental Replacement (Strangler Fig)

This spreads both the risk and the development effort over time. **Parallel Run** [189, p. 79] – i.e., running can ensure the new system emits the same behaviour as the legacy system before switching over.

3. Fowler himself admits he has no objection to changing the name but opted to strengthen instead the original metaphor as he had doubts whether it would catch on
4. *Big bang adoption* is a term commonly used to describe replacing an entire system at once, which often leads to problems [205]

Once the migration is complete, the router can be phased out, or it can be used as an **Anti-Corruption Layer** adapter (see § 11.1) for legacy clients.

11.2.3 Related patterns

- This pattern can be seen as a message-based temporary **Gateway Router** (see § 4.1)
- For communicating with a legacy system, consider using a **Anti-Corruption Layer** to isolate legacy domain logic and models (see § 11.1)

11.2.4 Example

ExampleEshop (see § 3.4) is working on implementing same-day delivery to improve its competitiveness. However, the technical limitations of the old system it uses in manage its warehouses (see § 11.1) prevent the implementation of this goal. Furthermore, since the original author left, the company lacks the necessary expertise with the system's legacy technologies to facilitate the needed changes.

The company has decided to implement a new warehouse management system to address this. However, this system is critical to business, and the company cannot afford the risk of a sudden switch. Instead, they decide to implement the new system gradually. They start by first reimplementing enough features to handle same-day delivery. Once this is done, the system is tested in parallel mode to see if it produces the same results as the old system. It is deployed to handle same-day delivery for a select subset of items if it proves stable.

Even though the new system cannot fully replace the old one yet, it already provides business value and can be gradually expanded with more features without time pressure.

11.2.5 Further reading

- Newman's book on migrating legacy systems to microservices contains many other useful patterns [189, p. 113]

11.3 Messaging Bridge

Connect two services with incompatible messaging middleware

This pattern is based on **Messaging Bridge** defined by Hohpe et al. [4, p. 132, 206] and later by Microsoft [207], and **Message Mover** by Fehling et al. [15, p. 225, 208].

11.3.1 Context

Two services need to communicate with each other but use incompatible messaging middleware. Linking them without changing (at least one of) the underlying services is preferred.

11.3.2 Solution

Create a new service (or change one of the existing parties) that acts as a bridge between the two brokers. This service receives messages from one broker and forwards them to the other (see fig. 11.3).

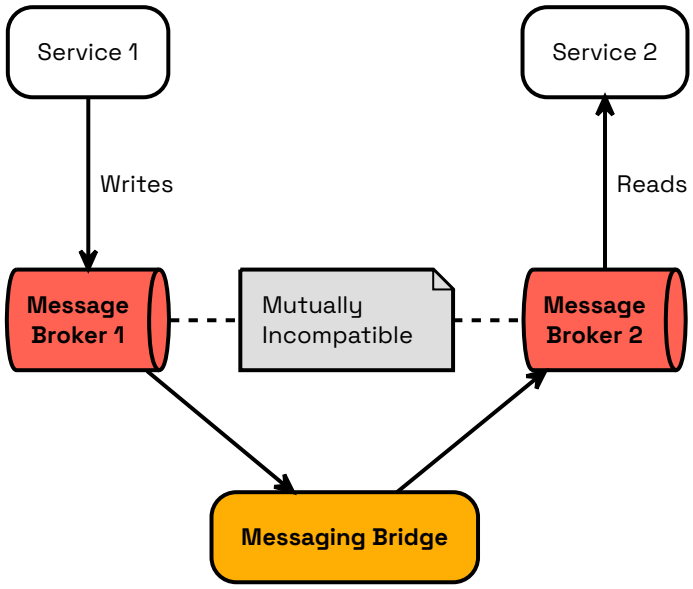

Figure 11.3: Messaging Bridge

11.3.3 Potential issues

Adding a bridge **increases complexity** and introduces an **additional point of failure** and **latency** to the system.

The two brokers may also have **incompatible features** that need to be reconciled.

11.3.4 Example

ExampleEshop (see § 3.4) wants to transition to a newer, more efficient messaging system for its new services. However, this process takes time and switching all services at once is unfeasible. To address this, the eshop deploys a messaging bridge that translates messages between the old and new systems. This allows them to gradually migrate services to the new system without updating the old services. Once all services are migrated, the bridge is removed.

11.3.5 Related patterns

- See **Publisher–Subscriber** for more information on message middleware (see § 4.2)
- The bridge may need to be scaled out using **Competing Consumers** (see § 6.1)
- If the systems also have incompatible domains, consider using an **Anti-Corruption Layer** (see § 11.1)

Conclusion

This thesis aimed to identify, catalogue and classify the most common distributed application architecture patterns in a structured and accessible way. It achieved this by conducting a comprehensive literature review and defining a methodology for this purpose.

The result is an exhaustive catalogue of the 33 most common patterns in this field, with thoroughly researched application criteria, advantages and disadvantages, densely cross-referenced, and including custom-designed visualisations and example use cases on a consistent domain.

These patterns are categorised to make them as easy as possible to navigate, understand, and, most importantly, apply to solve specific problems. To maximise this potential, the catalogue was fully transformed into a publicly available website, `https://jurf.github.io/daap/`.

Bibliography

1. Fowler, Martin. Design - Who needs an architect? *IEEE Software*. 2003, vol. 20, no. 5, pp. 11–13. Available from DOI: 10.1109/MS.2003.1231144.
2. Richards, Mark; Ford, Neal. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, 2020. ISBN 9781492043454.
3. Newman, Sam. *Building Microservices, 2nd Edition*. O'Reilly Media, 2021. ISBN 9781492034025.
4. Hohpe, Gregor; Woolf, Bobby. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003. Addison-Wesley Signature Series (Fowler). ISBN 978-0321200686.
5. Joshi, Unmesh. *Patterns of Distributed Systems*. Pearson Education, 2023. Addison-Wesley Signature Series (Fowler). ISBN 9780138222116.
6. Klimešová, Lucie. *Software Architecture Patterns and Design Principles* [online]. Brno, 2024 [visited on 2024-12-14]. Available from: https://is.muni.cz/th/b64nq/. MA thesis. Masaryk University, Faculty of Informatics. Supervised by Barbora Bühnová.
7. Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002. Addison-Wesley Signature Series (Fowler). ISBN 978-0-321-12742-6.
8. Fowler, Martin. *Enterprise Integration Patterns* [online]. [visited on 2024-11-23]. Available from: https://martinfowler.com/books/eip.html.
9. Schmidt, Douglas C.; Stal, Michael; Rohnert, Hans; Buschmann, Frank. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000. ISBN 978-0-471-60695-6.
10. Buschmann, Frank; Henney, Kevlin; Schmidt, Douglas C. *Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*. Wiley, 2007. ISBN 978-0-470-61347-9.
11. Nygard, Michael T. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007. ISBN 978-0-9787392-1-8.

12. NYGARD, Michael T. *Release It! Second Edition: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2018. ISBN 978-1680502398.

13. ROTEM-GAL-OZ, Arnon. *SOA Patterns*. Manning Publications, 2012. ISBN 9781933988269.

14. WILDER, Bill. *Cloud Architecture Patterns*. O'Reilly Media, 2012. Develop cloud-native applications. ISBN 9781449319779.

15. FEHLING, Christoph et al. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. Available from DOI: 10.1007/978-3-7091-1568-8.

16. HOMER, Alex; SHARP, John; BRADER, Larry; SWANSON, Masashi Narumoto Trent. *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*. Microsoft Developer Guidance, 2014. Microsoft patterns & practices. ISBN 978-1-62114-036-8.

17. MICROSOFT. *Cloud Design Patterns* [online]. 2024. [visited on 2024-10-22]. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/.

18. DEENADAYALAN, Anitha. *Cloud design patterns, architectures, and implementations* [online]. Amazon Web Services, 2023-07-28 [visited on 2024-10-22]. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/introduction.html.

19. NEWMAN, Sam. *Building Microservices*. O'Reilly Media, 2015. ISBN 9781491950357.

20. RICHARDSON, Chris. *Microservices Patterns: With examples in Java*. Manning Publications, 2018. ISBN 978-1617294549.

21. RICHARDSON, Chris. *A pattern language for microservices* [online]. 2024. [visited on 2024-10-20]. Available from: https://microservices.io/patterns/.

22. BURNS, Brendan. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*. O'Reilly Media, 2018. ISBN 9781491983614.

23. HOHPE, Gregor. *Conversation Patterns* [online]. 2017-01. [visited on 2024-12-05]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/conversation/index.html.

24. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994. Addison-Wesley Professional Computing Series. ISBN 9780321700698.

25. WIKIPEDIA CONTRIBUTORS. *Design Patterns* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-04]. Available from: https://en.wikipedia.org/w/index.php?title=Design_Patterns&oldid=1254518953.

26. MARTIN, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009. ISBN 978-0-13-235088-4.

27. DAVID, Alicia Bailey; GLORE, Peyton. The Impact of Design and Aesthetics on Usability, Credibility, and Learning in an Online Environment. *Online Journal of Distance Learning Administration*. 2010, vol. 13.

28. GUNN, Greg. *How to Apply a Color Palette to Your Design – Tutorial* [online]. The Futur Academy, 2024 [visited on 2024-12-06]. Available from: https://youtu.be/eXcKOqviLE0.

29. WORLD WIDE WEB CONSORTIUM. *Web Content Accessibility Guidelines (WCAG) 2.2* [online]. 2023-10-05. [visited on 2024-12-06]. Available from: https://www.w3.org/TR/WCAG22/.

30. NOVOTNÝ, Vít. Using Markdown Inside TEX Documents. *TUGboat* [online]. 2017, vol. 38, no. 2, pp. 214–217 [visited on 2020-07-31]. ISSN 0896-3207. Available from: https://tug.org/TUGboat/tb38-2/tb119novotny.pdf.

31. MACFARLANE, John. *Pandoc User's Guide* [online]. 2024-12-07. [visited on 2024-12-16]. Available from: https://pandoc.org/MANUAL.pdf.

32. COCKBURN, Alistair. *Hexagonal Architecture* [online]. 2005-01-04. [visited on 2024-12-06]. Available from: https://alistair.cockburn.us/hexagonal-architecture/.

33. LUKÁŠOVÁ, Helena. *Neverbální komunikace*. Masaryk University. FI:PV123 Visual Communication. Available also from: https://is.muni.cz/el/fi/podzim2024/PV123/um/2024_02_reprezentace.pdf?predmet=1642571.

34. SPEAR, Andrew D. Edmund Husserl: Intentionality and Intentional Content. *The Internet Encyclopedia of Philosophy* [online]. [N.d.] [visited on 2024-12-06]. ISSN 2161-0002. Available from: https://iep.utm.edu/huss-int/.

35. HAERDER, Theo; REUTER, Andreas. Principles of transaction-oriented database recovery. *ACM Comput. Surv.* 1983, vol. 15, no. 4, pp. 287–317. ISSN 0360-0300. Available from DOI: 10.1145/289.291.

36. WIKIPEDIA CONTRIBUTORS. *ACID* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-05]. Available from: https://en.wikipedia.org/w/index.php?title=ACID&oldid=1250666732.

37. PRITCHETT, Dan. BASE: An Acid Alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability. *Queue*. 2008, vol. 6, no. 3, pp. 48–55. ISSN 1542-7730. Available from DOI: 10.1145/1394127.1394128.

38. WIKIPEDIA CONTRIBUTORS. *Create, read, update and delete* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-28]. Available from: https://en.wikipedia.org/w/index.php?title=Create,_read,_update_and_delete&oldid=1249739822.

39. FOWLER, Martin. *What do you mean by "Event-Driven"?* [online]. 2017-02-07. [visited on 2024-10-28]. Available from: https://martinfowler.com/articles/201701-event-driven.html.

40. MICROSOFT. *Poison message handling* [online]. 2023-03-30. [visited on 2024-12-14]. Available from: https://learn.microsoft.com/en-us/dotnet/framework/wcf/feature-details/poison-message-handling.

41. WIKIPEDIA CONTRIBUTORS. *Fallacies of distributed computing* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-05]. Available from: https://en.wikipedia.org/w/index.php?title=Fallacies_of_distributed_computing&oldid=1253478356.

42. FOX, Armando; BREWER, Eric. Harvest, yield, and scalable tolerant systems. In: *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. 1999, pp. 174–178. Available from DOI: 10.1109/HOTOS.1999.798396.

43. GILBERT, Seth; LYNCH, Nancy. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*. 2002, vol. 33, no. 2, pp. 51–59. ISSN 0163-5700. Available from DOI: 10.1145/564585.564601.

44. RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN 0321245628.

45. MICROSOFT. *Gateway Routing pattern* [online]. 2022-08-29. [visited on 2024-10-22]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-routing.

46. DEENADAYALAN, Anitha. *API routing patterns* [online]. Amazon Web Services, 2023-07-28 [visited on 2024-10-29]. Cloud design patterns, architectures, and implementations. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/api-routing.html.

47. HOHPE, Gregor; WOOLF, Bobby. *Message Router* [online]. 2003. [visited on 2024-11-03]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageRouter.html.

48. HOHPE, Gregor; WOOLF, Bobby. *Content-Based Router* [online]. 2003. [visited on 2024-12-16]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html.

49. MICROSOFT. *Gateway Aggregation pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-aggregation.

50. MICROSOFT. *Gateway Offloading pattern* [online]. 2022-07-28. [visited on 2024-10-23]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-offloading.

51. RICHARDSON, Chris. *Pattern: API Gateway / Backends for Frontends* [online]. 2024. [visited on 2024-10-23]. Available from: https://microservices.io/patterns/apigateway.html.

52. WIKIPEDIA CONTRIBUTORS. *API management* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-16]. Available from: https://en.wikipedia.org/w/index.php?title=API_management&oldid=1258663368.

53. HOHPE, Gregor; WOOLF, Bobby. *Publish-Subscribe Channel* [online]. 2003. [visited on 2024-11-03]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html.

54. BUSCHMANN, Frank; HENNEY, Kevlin; SCHMIDT, Douglas C. *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. Wiley, 2007. ISBN 978-0-470-05902-9.

55. MICROSOFT. *Publisher-Subscriber pattern* [online]. 2018-12-07. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber.

56. DEENADAYALAN, Anitha. *Publish-subscribe pattern* [online]. Amazon Web Services, 2023-10-14 [visited on 2024-10-29]. Cloud design patterns, architectures, and implementations. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/publish-subscribe.html.

57. WIKIPEDIA CONTRIBUTORS. *Publish–subscribe pattern* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-16]. Available from: https://en.wikipedia.org/w/index.php?title=Publish%E2%80%93subscribe_pattern&oldid=1248914521.

58. WIKIPEDIA CONTRIBUTORS. *Inbox and outbox pattern* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-30]. Available from: https://en.wikipedia.org/w/index.php?title=Inbox_and_outbox_pattern&oldid=1260066808.

59. RICHARDSON, Chris. *Pattern: Messaging* [online]. 2024. [visited on 2024-10-28]. Available from: https://microservices.io/patterns/communication-style/messaging.html.

60. HOHPE, Gregor; WOOLF, Bobby. *Request-Reply* [online]. 2003. [visited on 2024-11-01]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/RequestReply.html.

61. MICROSOFT. *Asynchronous Request-Reply pattern* [online]. 2022-07-28. [visited on 2024-11-05]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/async-request-reply.

62. HOHPE, Gregor. *Asynchronous Request-Response* [online]. 2017-01. [visited on 2024-12-05]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/conversation/RequestResponse.html.

63. HOHPE, Gregor; WOOLF, Bobby. *Correlation Identifier* [online]. 2003. [visited on 2024-12-02]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/CorrelationIdentifier.html.

64. WIKIPEDIA CONTRIBUTORS. *Request–response* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-05]. Available from: https://en.wikipedia.org/w/index.php?title=Request%E2%80%93response&oldid=1253546792.

65. BURNS, Brendan. *The Distributed System ToolKit: Patterns for Composite Containers* [online]. 2015-06-29. [visited on 2024-10-28]. Available from: https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/.

66. RICHARDSON, Chris. *Pattern: Sidecar* [online]. 2024. [visited on 2024-10-23]. Available from: https://microservices.io/patterns/deployment/sidecar.html. Page under construction.

67. MICROSOFT. *Sidecar pattern* [online]. 2022-07-28. [visited on 2024-10-23]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar.

68. MICROSOFT. *Ambassador pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/ambassador.

69. HOHPE, Gregor; WOOLF, Bobby. *Channel Adapter* [online]. 2003. [visited on 2024-12-05]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/ChannelAdapter.html.

70. THOMAS, David; HUNT, Andrew. *The Pragmatic Programmer: Your journey to mastery, 20th Anniversary Edition*. Pearson Education, 2019. ISBN 978-0-13-595705-9.

71. CALÇADO, Phil. *The Back-end for Front-end Pattern (BFF)* [online]. 2015-09-18. [visited on 2024-10-23]. Available from: https://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff.html.

72. NEWMAN, Sam. *Backends For Frontends* [online]. 2015-11-18. [visited on 2024-03-06]. Available from: https://samnewman.io/patterns/architectural/bff/.

73. MICROSOFT. *Backends for Frontends pattern* [online]. 2022-07-28. [visited on 2024-10-23]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends.

74. MICROSOFT. *Pipes and Filters pattern* [online]. 2024-04-10. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters.

75. MICROSOFT. *Geodes pattern* [online]. 2024-03-18. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/geodes.

76. HOHPE, Gregor; WOOLF, Bobby. *Competing Consumers* [online]. 2003. [visited on 2024-10-23]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html.

77. MICROSOFT. *Competing Consumers pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/competing-consumers.

78. HOHPE, Gregor. *Load Balancer* [online]. 2017-01. [visited on 2024-12-05]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/conversation/LoadBalancer.html.

79. MICROSOFT. *Sharding pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/sharding.

80. MICROSOFT. *Sequential Convoy pattern* [online]. 2019-12-14. [visited on 2024-11-16]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/sequential-convoy.

81. WIKIPEDIA CONTRIBUTORS. *Shard (database architecture)* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-16]. Available from: https://en.wikipedia.org/w/index.php?title=Shard_(database_architecture)&oldid=1260915144.

82. WIKIPEDIA CONTRIBUTORS. *Data striping* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-16]. Available from: https://en.wikipedia.org/w/index.php?title=Data_striping&oldid=1255276345.

83. MICROSOFT. *Deployment Stamps pattern* [online]. 2023-12-13. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/deployment-stamp.

84. HOHPE, Gregor; WOOLF, Bobby. *Scatter-Gather* [online]. 2003. [visited on 2024-11-02]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/BroadcastAggregate.html.

85. DEENADAYALAN, Anitha. *Scatter-gather pattern* [online]. Amazon Web Services, 2024-05-07 [visited on 2024-10-29]. Cloud design patterns, architectures, and implementations. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/scatter-gather.html.

86. WIKIPEDIA CONTRIBUTORS. *Embarrassingly parallel* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-16]. Available from: https://en.wikipedia.org/w/index.php?title=Embarrassingly_parallel&oldid=1251754341.

87. RICHARDSON, Chris. *Pattern: Externalized configuration* [online]. 2024. [visited on 2024-10-30]. Available from: https://microservices.io/patterns/externalized-configuration.html.

88. FEHLING, Christoph et al. *Managed Configuration* [online]. 2020. [visited on 2024-11-17]. Available from: https://www.cloudcomputingpatterns.org/managed_configuration/.

89. MICROSOFT. *External Configuration Store pattern* [online]. 2021-09-14. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/external-configuration-store.

90. HOHPE, Gregor; WOOLF, Bobby. *Control Bus* [online]. 2003. [visited on 2024-11-13]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/ControlBus.html.

91. MICROSOFT. *Edge Workload Configuration pattern* [online]. 2022-10-12. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/edge-workload-configuration.

92. MICROSOFT. *Cloud design patterns that support reliability* [online]. 2024-10-10. [visited on 2024-12-15]. Azure Well-Architected Framework. Available from: https://learn.microsoft.com/en-us/azure/well-architected/reliability/design-patterns.

93. MICROSOFT. *Cache-Aside pattern* [online]. 2022-07-28. [visited on 2024-10-26]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside.

94. MICROSOFT. *Throttling pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/throttling.

95. MICROSOFT. *Compensating Transaction pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction.

96. MICROSOFT. *Bulkhead pattern* [online]. 2022-07-28. [visited on 2024-10-26]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead.

97. WIKIPEDIA CONTRIBUTORS. *Bulkhead (partition)* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-17]. Available from: https://en.wikipedia.org/w/index.php?title=Bulkhead_(partition)&oldid=1255984059.

98. WIKIPEDIA CONTRIBUTORS. *Thread pool* [online]. Wikipedia, The Free Encyclopedia, 2023 [visited on 2024-11-17]. Available from: https://en.wikipedia.org/w/index.php?title=Thread_pool&oldid=1138829275.

99. MICROSOFT. *Queue-Based Load Leveling pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling.

100. HOHPE, Gregor; WOOLF, Bobby. *Point-to-Point Channel* [online]. 2003. [visited on 2024-12-02]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/PointToPointChannel.html.

101. MICROSOFT. *Retry pattern* [online]. 2022-07-18. [visited on 2024-10-26]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/retry.

102. DEENADAYALAN, Anitha. *Retry with backoff pattern* [online]. Amazon Web Services, 2023-07-28 [visited on 2024-10-26]. Cloud design patterns, architectures, and implementations. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/retry-backoff.html.

103. HOHPE, Gregor. *Request-Response with Retry* [online]. 2017-01. [visited on 2024-12-05]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/conversation/RequestResponseRetry.html.

104. FEHLING, Christoph et al. *At-least-once Delivery* [online]. 2020. [visited on 2024-10-26]. Available from: https://www.cloudcomputingpatterns.org/at_least_once_delivery/.

105. WIKIPEDIA CONTRIBUTORS. *Exponential backoff* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-18]. Available from: https://en.wikipedia.org/w/index.php?title=Exponential_backoff&oldid=1244695962.

106. BROOKER, Marc. *Exponential Backoff And Jitter* [online]. Amazon Web Services, 2015-03-04 [visited on 2024-11-18]. Available from: https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/.

107. WIKIPEDIA CONTRIBUTORS. *Thundering herd problem* [online]. Wikipedia, The Free Encyclopedia, 2023 [visited on 2024-11-18]. Available from: https://en.wikipedia.org/w/index.php?title=Thundering_herd_problem&oldid=1177436868.

108. MICROSOFT. *Health Endpoint Monitoring pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring.

109. FEHLING, Christoph et al. *Watchdog* [online]. 2020. [visited on 2024-11-06]. Available from: https://www.cloudcomputingpatterns.org/watchdog/.

110. RICHARDSON, Chris. *Pattern: Health Check API* [online]. 2024. [visited on 2024-11-06]. Available from: https://microservices.io/patterns/observability/health-check-api.html.

111. JOSHI, Unmesh. *HeartBeat* [online]. Martin Fowler, 2023-11-23 [visited on 2024-12-11]. Catalog of Patterns of Distributed Systems. Available from: https://martinfowler.com/articles/patterns-of-distributed-systems/heartbeat.html.

112. HOHPE, Gregor; WOOLF, Bobby. *Control Bus* [online]. 2003. [visited on 2024-12-11]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/ControlBus.html.

113. HOHPE, Gregor; WOOLF, Bobby. *Test Message* [online]. 2003. [visited on 2024-12-11]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/TestMessage.html.

114. MICROSOFT. *Rate Limiting pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/rate-limiting-pattern.

115. WIKIPEDIA CONTRIBUTORS. *Denial-of-service attack* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-20]. Available from: https://en.wikipedia.org/w/index.php?title=Denial-of-service_attack&oldid=1258567977.

116. MAHDI, Nikrad. *An alternative approach to rate limiting* [online]. Figma Design, 2017-04-12 [visited on 2024-11-20]. Available from: https://medium.com/figma-design/an-alternative-approach-to-rate-limiting-f8a06cf7c94c.

117. NOORMOHAMMADPOUR, Max; RAGHAVENDRA, Cauligi. Datacenter Traffic Control: Understanding Techniques and Trade-offs. *IEEE Communications Surveys & Tutorials*. 2018, vol. 20, pp. 1492–1525. Available from DOI: 10.1109/COMST.2017.2782753.

118. WIKIPEDIA CONTRIBUTORS. *Rate limiting* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-20]. Available from: https://en.wikipedia.org/w/index.php?title=Rate_limiting&oldid=1239782071.

119. JOSHI, Unmesh. *Leader and Followers* [online]. Martin Fowler, 2023-11-23 [visited on 2024-11-03]. Catalog of Patterns of Distributed Systems. Available from: https://martinfowler.com/articles/patterns-of-distributed-systems/leader-follower.html.

120. MICROSOFT. *Leader Election pattern* [online]. 2024-05-30. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/leader-election.

121. REED, Benjamin; JUNQUEIRA, Flavio P. A simple totally ordered broadcast protocol. In: *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. Yorktown Heights, New York, USA: Association for Computing Machinery, 2008. LADIS '08. ISBN 9781605582962. Available from DOI: 10.1145/1529974.1529978.

122. ONGARO, Diego; OUSTERHOUT, John. In search of an understandable consensus algorithm. In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. Philadelphia, PA: USENIX Association, 2014, pp. 305–320. USENIX ATC'14. ISBN 9781931971102.

123. WIKIPEDIA CONTRIBUTORS. *Leader election* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-04]. Available from: https://en.wikipedia.org/w/index.php?title=Leader_election&oldid=1258669283.

124. MICROSOFT. *Cloud design patterns that support performance efficiency* [online]. 2024-10-10. [visited on 2024-12-15]. Azure Well-Architected Framework. Available from: https://learn.microsoft.com/en-us/azure/well-architected/performance-efficiency/design-patterns.

125. MICROSOFT. *Index Table pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/index-table.

126. MICROSOFT. *Materialized View pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/materialized-view.

127. MICROSOFT. *Compute Resource Consolidation pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/compute-resource-consolidation.

128. MICROSOFT. *Static Content Hosting pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/static-content-hosting.

129. MICROSOFT. *Valet Key pattern* [online]. 2024-05-08. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/valet-key.

130. RICHARDSON, Chris. *Pattern: Circuit Breaker* [online]. 2024. [visited on 2024-10-30]. Available from: https://microservices.io/patterns/reliability/circuit-breaker.html.

131. MICROSOFT. *Circuit Breaker pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker.

132. DEENADAYALAN, Anitha. *Circuit breaker pattern* [online]. Amazon Web Services, 2023-07-28 [visited on 2024-10-26]. Cloud design patterns, architectures, and implementations. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/circuit-breaker.html.

133. WIKIPEDIA CONTRIBUTORS. *Circuit breaker* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-19]. Available from: https://en.wikipedia.org/w/index.php?title=Circuit_breaker&oldid=1252964096.

134. DIEULOT, Alexandre. *instant.page* [online]. 2023-03-21. [visited on 2024-12-12]. Available from: https://instant.page/.

135. WIKIPEDIA CONTRIBUTORS. *Circuit breaker design pattern* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-16]. Available from: https://en.wikipedia.org/w/index.php?title=Circuit_breaker_design_pattern&oldid=1263311775.

136. FOWLER, Martin. *Remote Facade* [online]. 2003-03-05. [visited on 2024-11-22]. Available from: https://martinfowler.com/eaaCatalog/remoteFacade.html.

137. HOHPE, Gregor; WOOLF, Bobby. *Claim Check* [online]. 2003. [visited on 2024-10-20]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/StoreInLibrary.html.

138. MICROSOFT. *Claim-Check pattern* [online]. 2024-05-01. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/claim-check.

139. RICHARDSON, Chris. *Pattern: Command Query Responsibility Segregation (CQRS)* [online]. 2024. [visited on 2024-10-30]. Available from: https://microservices.io/patterns/data/cqrs.html.

140. MICROSOFT. *CQRS pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs.

141. DAHAN, Udi. *Clarified CQRS* [online]. 2009-12-09. [visited on 2024-10-28]. Available from: https://udidahan.com/2009/12/09/clarified-cqrs/.

142. YOUNG, Greg. *CQRS Documents* [online]. 2010-10. [visited on 2024-10-28]. Available from: https://cqrs.wordpress.com/wp-content/uploads/2010/11/cqrs_documents.pdf.

143. FOWLER, Martin. *CQRS* [online]. 2011-07-14. [visited on 2024-03-01]. Available from: https://martinfowler.com/bliki/CQRS.html.

144. WIKIPEDIA CONTRIBUTORS. *Command Query Responsibility Segregation* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-28]. Available from: https://en.wikipedia.org/w/index.php?title=Command_Query_Responsibility_Segregation&oldid=1251978026.

145. FOWLER, Martin. *Reporting Database* [online]. 2014-04-02. [visited on 2024-11-28]. Available from: https://martinfowler.com/bliki/ReportingDatabase.html.

146. RICHARDSON, Chris. *Pattern: Command-side replica* [online]. 2024. [visited on 2024-10-30]. Available from: https://microservices.io/patterns/data/command-side-replica.html.

147. FERNANDEZ-BUGLIONI, Eduardo. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley, 2013. Wiley Software Patterns Series. ISBN 9781119970484. Available also from: https://books.google.cz/books?id=3vppszXPdr0C.

148. MICROSOFT. *Cloud design patterns that support security* [online]. 2024-10-10. [visited on 2024-12-15]. Azure Well-Architected Framework. Available from: https://learn.microsoft.com/en-us/azure/well-architected/security/design-patterns.

149. MICROSOFT. *Quarantine pattern* [online]. 2024-01-19. [visited on 2024-12-15]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/quarantine.

150. DELESSY, Nelly; FERNANDEZ, Eduardo B; LARRONDO-PETRIE, Maria M. A pattern language for identity management. In: *2007 International Multi-Conference on Computing in the Global Information Technology (ICCGI'07)*. IEEE, 2007, pp. 31–31.

151. MICROSOFT. *Federated Identity Pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/federated-identity.

152. RICHARDSON, Chris. *Pattern: Access token* [online]. 2024. [visited on 2024-10-30]. Available from: https://microservices.io/patterns/security/access-token.html.

153. AL-SLAIS, Yaqoob; EL-MEDANY, Wael M. User-centric adaptive password policies to combat password fatigue. *Int. Arab J. Inf. Technol.* 2022, vol. 19, no. 1, pp. 55–62.

154. KORBAR, Bruno et al. Validating an agent-based model of human password behavior. In: *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*. 2016.

155. WIKIPEDIA CONTRIBUTORS. *Password fatigue* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-30]. Available from: https://en.wikipedia.org/w/index.php?title=Password_fatigue&oldid=1251069983.

156. HARDY, Norm. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.* [online]. 1988, vol. 22, no. 4, pp. 36–38 [visited on 2024-11-09]. ISSN 0163-5980. Available from DOI: 10.1145/54289.871709.

157. WIKIPEDIA CONTRIBUTORS. *Confused deputy problem* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-09]. Available from: https://en.wikipedia.org/w/index.php?title=Confused_deputy_problem&oldid=1230222963.

158. WIKIPEDIA CONTRIBUTORS. *Identity provider* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-29]. Available from: https://en.wikipedia.org/w/index.php?title=Identity_provider&oldid=1255298094.

159. WIKIPEDIA CONTRIBUTORS. *Federated identity* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-29]. Available from: https://en.wikipedia.org/w/index.php?title=Federated_identity&oldid=1250064587.

160. MICROSOFT. *Gatekeeper pattern* [online]. 2023-05-26. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/gatekeeper.

161. KIRSTENS ET AL. *Cross Site Scripting (XSS)* [online]. OWASP Foundation, 2024 [visited on 2024-12-13]. Available from: https://owasp.org/www-community/attacks/xss/.

162. WIKIPEDIA CONTRIBUTORS. *Honeypot (computing)* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-13]. Available from: https://en.wikipedia.org/w/index.php?title=Honeypot_(computing)&oldid=1256232168.

163. FEHLING, Christoph et al. *Transaction-based Processor* [online]. 2020. [visited on 2024-11-17]. Available from: https://www.cloudcomputingpatterns.org/transaction_based_processor/.

164. HOHPE, Gregor; WOOLF, Bobby. *Transactional Client* [online]. 2003. [visited on 2024-12-04]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/TransactionalClient.html.

165. RICHARDSON, Chris. *Pattern: Transactional outbox* [online]. 2024. [visited on 2024-10-30]. Available from: https://microservices.io/patterns/data/transactional-outbox.html.

166. DEENADAYALAN, Anitha. *Transactional outbox pattern* [online]. Amazon Web Services, 2023-07-28 [visited on 2024-10-29]. Cloud design patterns, architectures, and implementations. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/transactional-outbox.html.

167. JOSHI, Unmesh. *Two-Phase Commit* [online]. Martin Fowler, 2023-11-23 [visited on 2024-11-30]. Catalog of Patterns of Distributed Systems. Available from: https://martinfowler.com/articles/patterns-of-distributed-systems/two-phase-commit.html.

168. WIKIPEDIA CONTRIBUTORS. *Two-phase commit protocol* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-11-30]. Available from: https://en.wikipedia.org/w/index.php?title=Two-phase_commit_protocol&oldid=1259467470.

169. HELLAND, Pat. Life Beyond Distributed Transactions: An apostate's opinion. *Queue*. 2016, vol. 14, no. 5, pp. 69–98. ISSN 1542-7730. Available from DOI: 10.1145/3012426.3025012.

170. RICHARDSON, Chris. *Pattern: Polling publisher* [online]. 2024. [visited on 2024-10-30]. Available from: https://microservices.io/patterns/data/polling-publisher.html.

171. RICHARDSON, Chris. *Pattern: Transaction log tailing* [online]. 2024. [visited on 2024-10-30]. Available from: https://microservices.io/patterns/data/transaction-log-tailing.html.

172. GARCÍA-MOLINA, Héctor; SALEM, Kenneth. Sagas. *SIGMOD Rec.* 1987, vol. 16, no. 3, pp. 249–259. ISSN 0163-5808. Available from DOI: 10.1145/38714.38742.

173. RICHARDSON, Chris. *Pattern: Saga* [online]. 2024. [visited on 2024-10-30]. Available from: https://microservices.io/patterns/data/saga.html.

174. MICROSOFT. *Saga pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga.

175. DEENADAYALAN, Anitha. *Saga patterns* [online]. Amazon Web Services, 2023-07-28 [visited on 2024-10-29]. Cloud design patterns, architectures, and implementations. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/saga.html.

176. GARCÍA-MOLINA, Héctor et al. Modeling long-running activities as nested sagas. *Data Eng.* 1991, vol. 14, no. 1, pp. 14–18.

177. RICHARDSON, Chris. *Using sagas to maintain data consistency in a microservice architecture* [online]. Devoxx, 2017-05-17 [visited on 2024-12-01]. Available from: https://youtu.be/YPbGW3Fnmbc.

178. FRIEDRICHSEN, Uwe. *The limits of the Saga pattern* [online]. 2021-02-19. [visited on 2024-11-30]. Available from: https://www.ufried.com/blog/limits_of_saga_pattern/.

179. WIKIPEDIA CONTRIBUTORS. *Long-running transaction* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-02]. Available from: https://en.wikipedia.org/w/index.php?title=Long-running_transaction&oldid=1212089231.

180. WIKIPEDIA CONTRIBUTORS. *Compensating transaction* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-02]. Available from: https://en.wikipedia.org/w/index.php?title=Compensating_transaction&oldid=1227504219.

181. DEENADAYALAN, Anitha. *AWS Choreography Pattern* [online]. Amazon Web Services, 2023-10-14 [visited on 2024-10-29]. Cloud design patterns, architectures, and implementations. Available from: https://aws.amazon.com/documentation/choreography.

182. DEENADAYALAN, Anitha. *Saga orchestration pattern* [online]. Amazon Web Services, 2023-07-28 [visited on 2024-10-29]. Cloud design patterns, architectures, and implementations. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/saga-orchestration.html.

183. MICROSOFT. *Scheduler Agent Supervisor pattern* [online]. 2022-07-28. [visited on 2024-12-02]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/scheduler-agent-supervisor.

184. FOWLER, Martin. *Event Sourcing* [online]. 2005-12-12. [visited on 2024-10-26]. Available from: https://martinfowler.com/eaaDev/EventSourcing.html.

185. RICHARDSON, Chris. *Pattern: Event sourcing* [online]. 2024. [visited on 2024-10-26]. Available from: https://microservices.io/patterns/data/event-sourcing.html.

186. MICROSOFT. *Event Sourcing pattern* [online]. 2022-07-28. [visited on 2024-10-26]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing.

187. DEENADAYALAN, Anitha. *Event sourcing pattern* [online]. Amazon Web Services, 2023-10-14 [visited on 2024-10-29]. Cloud design patterns, architectures, and implementations. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/event-sourcing.html.

188. WIKIPEDIA CONTRIBUTORS. *Event-driven architecture* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-16]. Available from: https://en.wikipedia.org/w/index.php?title=Event-driven_architecture&oldid=1262812523.

189. NEWMAN, Sam. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, 2019. ISBN 978-1-492-07554-7.

190. EVANS, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004. ISBN 978-0-321-12521-7.

191. RICHARDSON, Chris. *Pattern: Anti-corruption layer* [online]. 2024. [visited on 2024-10-30]. Available from: https://microservices.io/patterns/apigateway.html. Page under construction.

192. DEENADAYALAN, Anitha. *Anti-corruption layer pattern* [online]. Amazon Web Services, 2023-07-28 [visited on 2024-10-29]. Cloud design patterns, architectures, and implementations. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/acl.html.

193. MICROSOFT. *Anti-corruption Layer pattern* [online]. 2022-07-28. [visited on 2024-10-28]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer.

194. FOWLER, Martin. *Refactoring code that accesses external services* [online]. 2015-02-17. [visited on 2024-02-12]. Available from: https://martinfowler.com/articles/refactoring-external-service.html.

195. FOWLER, Martin. *Gateway* [online]. 2021-08-10. [visited on 2024-12-03]. Available from: https://martinfowler.com/articles/gateway-pattern.html.

196. WIKIPEDIA CONTRIBUTORS. *Facade pattern* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-03]. Available from: https://en.wikipedia.org/w/index.php?title=Facade_pattern&oldid=1250938293.

197. WIKIPEDIA CONTRIBUTORS. *Adapter pattern* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-03]. Available from: https://en.wikipedia.org/w/index.php?title=Adapter_pattern&oldid=1235788258.

198. WIKIPEDIA CONTRIBUTORS. *Mediator pattern* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-03]. Available from: https://en.wikipedia.org/w/index.php?title=Mediator_pattern&oldid=1234081821.

199. FOWLER, Martin. *Strangler Fig Application* [online]. 2004-06-01. [visited on 2024-12-03]. Available from: https://martinfowler.com/bliki/OriginalStranglerFigApplication.html.

200. FOWLER, Martin. *Strangler Fig* [online]. 2024-08-22. [visited on 2024-10-28]. Available from: https://martinfowler.com/bliki/StranglerFigApplication.html.

201. RICHARDSON, Chris. *Pattern: Strangler application* [online]. 2024. [visited on 2024-10-30]. Available from: https://microservices.io/patterns/communication-style/messaging.html.

202. DEENADAYALAN, Anitha. *Strangler fig pattern* [online]. Amazon Web Services, 2023-07-28 [visited on 2024-10-29]. Cloud design patterns, architectures, and implementations. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/strangler-fig.html.

203. MICROSOFT. *Strangler Fig pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig.

204. WIKIPEDIA CONTRIBUTORS. *Strangler fig application* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-03]. Available from: https://en.wikipedia.org/w/index.php?title=Strangler_fig&oldid=1250144779.

205. WIKIPEDIA CONTRIBUTORS. *Big bang adoption* [online]. Wikipedia, The Free Encyclopedia, 2024 [visited on 2024-12-03]. Available from: https://en.wikipedia.org/w/index.php?title=Big_bang_adoption&oldid=1230084412.

206. HOHPE, Gregor; WOOLF, Bobby. *Messaging Bridge* [online]. 2003. [visited on 2024-11-06]. Available from: https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingBridge.html.

207. MICROSOFT. *Messaging Bridge pattern* [online]. 2022-07-28. [visited on 2024-10-29]. Cloud Design Patterns. Available from: https://learn.microsoft.com/en-us/azure/architecture/patterns/messaging-bridge.

208. FEHLING, Christoph et al. *Message Mover* [online]. 2020. [visited on 2024-11-17]. Available from: https://www.cloudcomputingpatterns.org/message_mover/.

A Digital attachments

- `daap.zip` – source code of the website