O'REILLY®

Building Resilient Distributed Systems

Patterns and Practices for Stable Software



Sam Newman

Praise for *Building Resilient Distributed Systems*

Resilience isn't something you bolt onto a system. It comes from the architecture, the people, and their ability to respond when the world doesn't behave as expected, which frankly it rarely does. Sam Newman brings those pieces together brilliantly, combining deeply practical advice with the engineering judgment behind it. As AI changes how we build and operate software, that judgment matters more than ever. This is an important book about the fundamentals we cannot afford to forget and I believe are more important than ever.

—Rachel Laycock, CTO, Thoughtworks

Sam Newman has written the field guide for the day after the should-never-happen scenario happens. Clear, practical, and grounded in how systems and people actually behave, this book belongs within reach of every software architect.

—Scott Hanselman, VP and member of technical staff, Microsoft/GitHub

Building resilient distributed systems is a complicated thing to do well; there are so many things to consider and so many ways that even simple things, such as one part of the system invoking behavior in another, can go wrong. Sam Newman explores them in detail in this book.

—Dave Farley, coauthor of *Continuous Delivery* and author of *Modern Software Engineering*

Resiliency is one of those composite architectural characteristics that someone could write a whole book about—and Sam did! This book is a deep dive into the many facets of resiliency, covering not just the definitions but also how it intersects with other aspects of software architecture: hardware, software, and the larger organization. I wish more books like this existed, taking a comprehensive look at one critical aspect of successful modern software systems.

—Neal Ford, director and software architect, Thoughtworks, and coauthor of *Fundamentals of Software Architecture*, 2nd edition

Failure is a given in distributed systems. Knowing how to recover from it matters—but knowing how to prevent it in the first place, technically and organizationally, matters just as much. This book covers both, giving you the tools to build resilient systems and to handle them when they inevitably fail. Every team working on distributed systems should have this on their bookshelf.

—Mark Richards, founder, DeveloperToArchitect.com, and coauthor of *Fundamentals of Software Architecture*, 2nd edition

An excellent guide to resilient architecture—practical, opinionated (and I agree with the opinions), and entertaining to read.

—Adrian Cockcroft, OrionX.net, former Netflix and AWS architect

Wisdom-packed. Buy, read, pass it on to anyone who develops and maintains mission-critical applications.

—Dr. Venkat Subramaniam, award-winning author and founder, Agile Developer, Inc.

I have been building distributed systems at large scale for 30 years (sometimes even resiliently!). There is no other single reference that lays out the hard-won techniques and trade-offs of the discipline so concisely or clearly.

—Randy Shoup, SVP engineering, CircleCI

From tubthumping to thundering herds, from CAP theorem to culture and people, this book will tell you how to keep it together when your distributed systems might otherwise fall apart. Sam delivers on the promise of resilience with a catalog of patterns supported by clear explanations, practical examples, stories, and dry humor.

—Kevlin Henney, coauthor of *A Pattern Language for Distributed Computing*

Things will go wrong in any distributed system, but the impact can be mitigated through building in resilience. I've picked up an understanding of how to do that from a variety of places over a long time—plus bitter experience. This book lays it all out for you: how to build resilient systems and respond effectively to failures.

—Sarah Wells, independent tech consultant and author of *Enabling Microservice Success*

Resilience isn't an easy topic to write about, and yet Sam has somehow managed to jam many years of hard-won lessons into a single volume. If you're building a distributed system (and who isn't these days!), this is an essential read.

—Daniel Bryant, platform engineer and PMM, Syntasso

Most resilience books stop at circuit breakers and retries. Sam goes further, showing that a truly resilient system is a sociotechnical one, where the people and processes matter as much as the architecture.

—Luca Mezzalira, director, 50 Cents Media Ltd

A must-read for those interested in building large-scale systems. This book goes beyond the basic patterns than other books. It covers topics such as thundering herds, cell-based architectures, and running a service in production. It focuses on the people, process, and culture, and it's a page-turner.

—Tom Kerkove, principal software engineer, Microsoft

The book I want on the desk of every team putting agents into production. It is also the most honest writing on AI I have read this year. Sam shows names where LLMs help you build resilient systems, and where they have no business being. The teams I work with are wiring agents into everything right now. This book gives them the questions to ask first.

—Benedikt Stemmildt, cofounder, hackers&wizards

Most of what I know about reliability I learned in hindsight, when it was too late. This book is all the lessons I wish I'd had upfront: practical, pragmatic, and full of ideas I'll carry into my next system.

—Martha Lambert, product engineer, incident.io

Building Resilient Distributed Systems

Patterns and Practices for Stable Software

Sam Newman



Building Resilient Distributed Systems

by Sam Newman

Copyright © 2026 Sam Newman. All rights reserved.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Louise Corrigan

Development Editors: Virginia Wilson, Michele Cronin

Production Editor: Elizabeth Faerm

Copyeditor: Arthur Johnson

Proofreader: Sharon Wilkey

Indexer: Judith McConville

Cover Designer: Susan Brown

Cover Illustrator: Karen Montgomery

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

September 2026: First Edition

Revision History for the First Edition

- 2026-09-15: First Release

See *https://oreilly.com/catalog/errata.csp?isbn=9781098163549* for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Building Resilient Distributed Systems*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher’s views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-16354-9

[LSI]

Preface

Distributed systems come in many shapes and sizes, and they are increasingly a major part of how we build and deliver software. But distributed systems also create challenges. At face value, they appear to offer the ability to create more stable software. No longer should the death of a single machine take our system offline, right? Unfortunately, the world is not that simple.

While distributed systems seem simple on the surface, complexity emerges from a set of initially straightforward interactions between computers. Calls can take too long or fail, networking equipment can catch fire, configuration errors can send your network packets down the wrong trouser leg of the internet, data centers can be taken out by regional conflict, you could be hit by an indiscriminate distributed denial of service attack, or a simple cache failure could bring your whole system down.

It gets worse. A distributed system also consists of the people who operate and use the system, and they can be their own sources of instability. However, the human component is also the most vital one in taking our distributed systems and making them resilient.

In this book, I explore the nature of distributed systems and the notion of resiliency in general. Through the following 16 chapters, I'll take you on a journey in which we look at everything from the humble timeout to building collapse; retrying calls to CAP theorem; rate limiting to sociotechnical systems. By the end, you'll know a lot more about how to make your system more resilient from both a technical and a people point of view, and you'll have a much more rounded understanding of the notion of resilience in general.

Who This Book Is For

Building Resilient Distributed Systems is primarily for anyone in a technical role who is helping to build a distributed system. Developers, architects, testers, operations folks, and site reliability engineers alike will find a lot here that will help them in their day job.

For the nontechnical folks, there is a lot in here for you as well. You'll find a good deal of coverage of cultural and social aspects and how they can positively impact the resiliency of your distributed systems; the third part of the book is focused on these aspects.

Whether you are struggling with achieving acceptable resiliency for an existing system, hoping to avoid making too many mistakes as you start your own journey into the world of distributed systems, or looking to understand what resilience even is, this is the book for you.

Prerequisite Knowledge

In general, while I have done my best to explain terms clearly (and I include a Glossary at the end to help with this), this is not a book for people who are brand-new to working with software.

The technical sections of this book (Parts I and II) assume that you at least have experience in building simple server-based software and running it in production, and the content here is aimed squarely at technical practitioners.

What You Will Learn

This book explains how you can deliver resilient software while building distributed systems. I'll take you on a deep dive into the technical aspects, examining topics such as timeouts, disaster recovery, and yes, even CAP theorem.[1] But we'll explore the nontechnical side as well, looking at organizational culture, how teams work together, and how humankind's ability to make mistakes shouldn't automatically result in system outages.

Behind it all, you'll see the systems you create in a broader context, and by the end, you'll see how bringing together lessons from the world of human factors and system safety with established resiliency patterns can help you pick the right path through an ever more distributed future.

By the end of this book, you will understand these concepts:

- The importance of resilience in distributed systems
- The fundamentals of system stability, and the architectural patterns that can be used
- The human elements that need to be considered

And you will be able to do the following:

- Put ideas into practice that will make your systems more resilient
- Implement improved observability into the resiliency of your system
- Handle failure much better, should the worst happen

Navigating the Book

Building Resilient Distributed Systems is organized into three parts, followed by a Glossary, Appendix, and Bibliography. Part I, "Foundations", and Part II, "Dealing with Failure", focus primarily on technical aspects, with Part III, "People, Process, and Culture", looking at people and culture. I wrote the book with the idea that you can read it from start to finish, and on the technical side of things I have structured the content so that the topics get progressively more complex as you go, with more foundational topics coming earlier. You may already be an expert on many of these topics, but I still think there will be things you can learn about topics you think you know well. (I learned a lot researching Chapter 3, for example!)

Should you feel the desire to just jump into the middle of the book and delve into the topic you're most interested in, I would ask that you at least

read Chapter 1 first, as it lays out some fundamental concepts and explains the worldview I had when writing this book. Subsequent chapters will make much more sense if you've read Chapter 1 first!

Chapter 1, "What Is Resiliency?"

> Resiliency can mean different things to different people, so it's important to start with a shared understanding. This chapter looks at resiliency from a number of angles, exploring concepts from the wider resilience engineering space, and it also introduces the concept of sociotechnical systems.
>
> This chapter sets the scene for the rest of the book. Even if you plan to start in the middle of the book so you can read about the topic you are most interested in, please read this chapter first.

Part I: Foundations

The first part of this book focuses on the foundational technical aspects of making systems more resilient:

Chapter 2, "Fundamentals of Observability"

> Wanting your system to be resilient is one thing; knowing it's resilient is something else. In this chapter, I'll take you through how to collect the information you need to ensure you're meeting your targets. I'll also cover how to define service level objectives (SLOs), explore team versus system targets, and look at why error budgets can be useful.

Chapter 3, "Timeouts"

> Networks and computers can be frustrating, and they can stop working (or go slow) at the worst time. Dealing with this fundamental truth starts with knowing when to give up—in this chapter, you'll learn all about timeouts, including how to set them correctly by balancing the needs of users against what the system itself can handle.

Chapter 4, "Retries and Idempotency"

> If at first you don't succeed, try again! Or maybe give up? When computers or network calls start failing, trying again often makes sense, and that is the main thrust of this chapter. However, trying again means we have to deal with what happens if we end up doing the same work more than once, so I'll also take you on a deep dive into the topic of idempotency.

Chapter 5, "Thundering Herds"

> Systems are often brought down when there is too much demand. As I explore in this chapter, not all spikes in load are the same—these thundering herds can be internal or external, malicious or accidental, a sign of a problem or a sign of success. I'll look at everything here from denial of service attacks to how a cache failing can bring down a

system, while also presenting a range of mitigations to help fend off thundering herds of different shapes and sizes.

Chapter 6, "Rate Limiting"

This chapter explores how to reduce the amount of work your system is doing to keep it stable. It covers backpressure, load shedding, circuit breakers, and more.

Chapter 7, "Resiliency and AI"

Although I cover the role of AI throughout this book, I wanted one chapter to focus on the opportunities—and risks—that AI can bring. In addition to looking at the role of AI in building and operating our software, I'll also look at the impact of embedding AI in the products we build. And I'll explore the impact of AI (both positive and negative) on the human operators of our system.

Part II: Dealing with Failure

In the second part of the book, we dive into more complex technical topics and look at how to handle the failure of everything from individual computers to entire data centers. You'll also see how a system needs to adapt when these failures occur:

Chapter 8, "Handling Machine Failure"

> That computers fail isn't a surprise, so it's beholden on those of us building resilient software to understand the nature of these failures and work out how to mitigate them. In this chapter, you'll see how load balancing (both client-side and server-side) can help ensure that a service remains operational even when machines fail. Additionally, I'll explore the difference between stateless and stateful processing and explain why machine failure makes stateful processing painful to implement.

Chapter 9, "Message Brokers"

> A common technique for making sure messages get delivered reliably is to make use of a message broker. In order to provide delivery guarantees, brokers themselves have to be built to handle machine failure, and this chapter explores how this is done. Additionally, this chapter covers a number of resilience-related patterns for working with a message broker, such as the transactional outbox and consumer-managed inbox patterns.

Chapter 10, "Disaster Recovery"

> One machine failing isn't great, but having an entire data center fail is worse! In this chapter, I'll discuss patterns that can be used to recover after a large-scale outage, and I'll give you a simple model for defining your disaster recovery (DR) requirements. Also detailed here are the options around cross-site data replication, which is essential for many of the DR patterns we explore.

Chapter 11, "Progressive Collapse"

One small issue can often cause a cascade of failures, resulting in a much more significant collapse of the wider system. This is a phenomenon known as *progressive collapse*. Drawing on case studies in the built environment, this chapter will look at the lessons we can learn from progressive collapse in the built environment and at how to mitigate the risk of a progressive collapse in our distributed systems.

Chapter 12, "Using Formal Theorems to Understand Trade-Offs"

When things go wrong, we often have to adapt the way our system operates. Formal theorems can help us reason about the trade-offs available to us in our distributed system when things don't work the way we want, and we'll explore two of these theorems in this chapter: CAP and PACELC. This chapter will also explore Little's Law and show how it can be useful in reducing resource contention.

Part III: People, Process, and Culture

When considering the resiliency of a distributed system, we have to move beyond the technical and explore the behaviors, culture, and processes of the people building and maintaining the system itself:

Chapter 13, "The Sociotechnical Systems"

> The concept of the sociotechnical system has been around for decades, but it is only somewhat recently that this school of thinking has come to the forefront in digital system resiliency. In this chapter, I'll take you through the implications of approaching resiliency through a sociotechnical lens and introduce several models that can help you make sure you're taking a holistic view of resiliency.

Chapter 14, "Incident Management and Post-Incident Response"

> When things go wrong, what do you do? In this chapter, I'll explore how to send good alerts and how these alerts can then be handled effectively. We'll look at how to avoid operator burnout, how to coordinate incident response, and how to deal effectively with the aftermath.

Chapter 15, "Constant Learning and Adaptation"

> To keep a system resilient, you need to continually improve. At the most basic level, this means making sure that you learn as much as possible when things go wrong. Learning goes beyond this, however, so you'll see how nurturing the right culture in your organization is vital, as well as how activities like game days can be a useful part of building a learning organization.

Chapter 16, "Conclusion"

> This chapter serves as a wrap-up of the main themes and patterns covered in the preceding 15 chapters.

Writing Process and Use of AI

No AI was used to write any of my words in this book, nor did I use AI to draw any of the diagrams. The only AI tools I used during the three-year period of writing this book were Granola, which I used to capture notes from meetings with my editors, and NotebookLM, which I used sparingly to help explore the various papers and articles I used for research. Where citations were provided by NotebookLM, they were checked. This basically means that any errors or bad phrasing are down to me, and I wouldn't have it any other way.

The first commit by me for the book was on Monday, September 11, 2023, and at the time of writing the last commit was on Monday, July 6, 2026, with 554 commits during this period. This excludes the changes made as part of the review process, for which I moved over to Google Docs.

Most of the writing was done in AsciiDoc using Visual Studio Code on macOS, before the content was moved over to Google Docs for reviewer feedback. OmniGraffle was used to create the initial drafts of all the diagrams in the book, with Guardian Sans used for all text in the diagrams. O'Reilly's Atlas toolchain was used to create the various forms of the book from the source AsciiDoc, with a lot of help from O'Reilly's lovely production team.

The vast majority of this writing was done at home in Kent, although I do need to say a big thank you to the city of Sydney, Australia, for both the Glebe and Marrickville public libraries, where I spent a good chunk of time writing while working and visiting family overseas.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

> Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, names of services, error codes, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

O’Reilly Online Learning

> **NOTE**
>
> For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

141 Stony Circle, Suite 195

Santa Rosa, CA 95401

800-889-8969 (in the United States or Canada)

707-827-7019 (international or local)

707-829-0104 (fax)

support@oreilly.com

https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata and any additional information. You can access this page at *https://oreil.ly/building-resilient-distributed-systems*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

This book is dedicated to Robbie, Nella,[2] Indira, Donncha, Laurie, and Dan. I'm not sure what you'll all make of this book, but this one is for you. Your badgers are in the post.

As with my previous books, this one wouldn't be possible without the love and support of my wife, Lindy Stephens, who continues to have the patience of a saint. Without her, I doubt I would have written one book, let alone four.

A special thanks goes to my reviewers: Abby Bangser, Daniel Bryant, Lukasz Dynowski, Martha Lambert, Halvard Skogsrud, Venkat Subramaniam, and Sarah Wells. They all provided valuable feedback, ideas, and stories along the way and generally made this book better than it would have been otherwise. I also want to thank the lovely folks at O'Reilly: my editors Virginia Wilson and Michele Cronin; Louise Corrigan and Melissa Duffield for keeping me sane these last few years; and on the production side of things, Liz Faerm, Arthur Johnson, Kate Dullea, Gregory Hyman, and likely many other people from O'Reilly whom I'm sure I forgot to mention.

A whole host of people contributed to the book in other ways, either by helping me on specific topics, by supporting me, or just being good eggs in general. They are (in no particular order) Liz Fong-Jones, Jessica Kerr, Charity Majors, John Allspaw (for inspiring me to write this in the first

place), Mike Nyygard, Ben Butler-Cole, Constantin Valdor, Martin Fowler, Courtney Nash, Randy Shoup, Marc Brooker, Ellis and John, Kane “Mr. Businessman” Stephens, Stuart Rimell, Joe McKevitt, Jay Wright, David Ferrante, Mathew Leger, Alex Borisevich, Gregor Hohpe, Bobby Wolf, Adrian Cockcroft, Dave Farley, Luca Mezzalira, Mark Richards, Neal Ford, Rachel Laycock, Randy Shoup, Scott Hanselman, Tom Kerkove, Benedikt Stemmildt, everyone at Gilmanco Staynes, Jennifer Sole, John Blanche, Borehammer, the Royal Horticultural Society—and finally, the South Sydney Rabbitohs (#SSTID), Arsenal FC, and both the Kent and England cricket teams for continually giving me hope, then dashing it.

I’d also like to thank everyone who contributed to the talks, podcasts, books, papers, incident reports, and anecdotes without which this book would not be possible. I have tried to attribute everything where I thought it was needed, but I’m certain I’ve missed more than a few. Without the work that came before, this book wouldn’t exist. So to all of you who take the time to share your experiences, expertise, and, even more importantly, your mistakes, thank you!

---

1 Don’t worry—CAP is not actually as complex as you might think.

2 Officially my favorite niece.

Chapter 1. What Is Resiliency?

I get knocked down
But I get up again
You're never gonna keep me down

—"Tubthumping," Chumbawamba

The world is an increasingly chaotic place. Our industry is partway through an AI revolution, which depending on your point of view will be either the greatest thing ever or the greatest disaster ever. More broadly, the lack of stability in the current geopolitical backdrop sends echoes through our lives in ways that would have been difficult to predict only a decade before. From global pandemics and oil blockades to wars in Europe and climate change, it feels like we are speed running new versions of previous global upheavals. And no, we can't blame AI for all of this.

Against this backdrop, it is ever more important for us to be able to rely on things. Attempting to bring stability to the global economy, relations between nation-states, or the climate is well outside the realm of this book (and that of my ability), but I can at least try to help you all build more resilient distributed systems.

To begin our journey, let's look at the fundamental concepts of resiliency. I'll define what resiliency is and discuss the challenges of resiliency within the context of distributed systems. I will also look beyond the technology side of things to show how behaviors, culture, organization, and societal context can impact our ability to deliver resiliency.

Defining Resiliency

Resiliency is the ability of a system to withstand problems, recover from issues that impact it, and continue evolving in the face of changing circumstances to ensure resiliency over a longer period. A resilient system

is one that is predictable, that can be relied upon—and, if needed, can get back up again if it does get knocked down.

Resiliency is a desirable characteristic. Whether applying the term to a person, a mechanism, or a digital system, when did you ever think, "I don't want this to be resilient"?

As you'll see throughout this chapter, you can approach the concept of resiliency from a number of different angles. Fundamentally, this book is about helping you to deliver resilient *systems*. You can't just look at software or hardware in isolation; you need to look at the system as a whole. To get you started thinking more broadly about systems, let's take a look at resiliency from the twin viewpoints of technology and society.

Technology

From a technology viewpoint, you can look at resiliency in terms of making sure that your program doesn't crash—ensuring that it can handle errors gracefully and will not fall over in a blaze of glory (and stack traces), leaving your users at best bewildered and at worst in danger.

The concept of resiliency can also encompass your hardware. Can you tolerate a single machine unexpectedly powering off? Or should you invest in servers with redundant power supplies and networking to ensure they are less likely to fail?

Since the advent of the public cloud, and the broader move to virtualization and software-defined infrastructure, things have fundamentally shifted. Aspects around managing individual machines are likely not under your direct control. Instead, you have to consider which of a myriad of cloud products will deliver what you need.

We're also still in the midst of the biggest technical upheaval to impact our industry since the emergence of the public cloud. The use of AI is changing not only how we build and manage our software but also how our users interact with the software. Right now, there is no certainty in terms of how AI will change the work we do; all we know is that our work is changing.

This injects a degree of uncertainty into everything we do, in a way that is arguably unprecedented in the history of software development.

Later in this chapter, I'll show you the specific challenges that distributed systems introduce and that cause us to take into consideration concepts such as timeouts, retries, and much more—topics I will spend a lot of time on in this book. Before that, though, you have to expand your horizons somewhat. What about the people who actually build, operate, and use the systems you create?

Social

As a software developer, it can be tempting to try to focus just on the act of coding, to the exclusion of other "noise." Dedicating our attention to algorithms, data structures, network protocols, or other technical implementation details requires focus, commitment, and time. Unfortunately, it takes only a cursory examination of how things fail to realize that this approach is not only naive but also can be dangerous.

For example, an environment in which the act of raising safety concerns is discouraged or even punished will lead to issues that could have been caught before impacting users being missed. The country or industry sector you operate in might require that systems are built in certain ways to comply with local legislation.

Different types of users will have different expectations for the systems they use. The impact of failures can also be starkly different—an information display in a shopping mall crashing and serving up a stack trace isn't great, but it's not as bad as a self-driving car accelerating when it should brake.

This of course also deals very neatly with the adage that you should "keep politics out of technology." Once you understand that our systems are influenced by the people who build and use the software, the culture they operate in, and the wider societal context, you see that this statement doesn't really make much sense. For example, I worked with a Nordic government agency that was basing its technology on Kubernetes hosted on

Azure. It was only very recently that government guidance had changed and allowed the agency to make use of public cloud technology. The agency was keenly aware that the political winds could change and that this decision to allow the use of a public cloud vendor could be reversed. As a result, the agency was avoiding the use of Azure-specific technologies in order to ease its migration to a private cloud infrastructure if required. This in turn created challenges, as the agency ended up having to do more work itself rather than being able to take advantage of the Azure-specific functionality.

When considering resiliency, getting a grasp of the societal aspects at play can be difficult, especially if you have spent most of your career focusing on technical concerns. In Chapter 13, I will introduce a model that will provide some structure around these ideas and help you understand where you need to focus your time if you are to deliver truly resilient systems.

> **NOTE**
>
> Maintaining the resilience of your system requires technology and people working together in the right environment.

Why Resilience Matters

The majority of people in the world have their own computer in the form of a smart phone,[1] and many people may have more than that in the form of smart watches, smart TVs, tablets, desktops, and laptops. As a result, we are surrounded by software-based systems, and they permeate both our professional and private lives in ways that would have been difficult to predict in the previous millennium.

Software is such a key part of the way most of us live our lives, and so pivotal to the work we do. The global lockdowns that occurred during the COVID pandemic meant that there was a renewed focus on digital services, as they became even more vital.

As a result, the impact of software failures is all the more keenly felt. Toward the end of 2025, two Cloudflare outages in a matter of weeks took down major sites globally.[2] And those came after the AWS US-EAST-1 region suffered an outage that took down systems all over the world (an incident we'll examine in more detail in Chapter 11).[3]

During the past decade, the systems we are creating have become more and more distributed, more and more interconnected. As a result of the shift from mainframes to modern server infrastructure, the solutions delivered are increasingly running on ever more complex computer topologies, creating new challenges.

Given that the genie can't be put back into the bottle and that digital, software-based systems are here to stay, the resiliency of these systems has never been more key.

What Is a Distributed System?

The growing importance of software in our lives has coincided with the increasing distribution and scale of the systems we create. As you will see throughout this book, a distributed system can help us improve the resiliency of our systems, but at the same time, it introduces new sources of complexity that need to be taken into consideration. Before I get into that, however, let's look at what a distributed system actually is.

A distributed system is one in which two or more computers talk to one another over a network.[4] Distributed systems come in many shapes and sizes. Consider the simple web application shown in Figure 1-1: the code running the web backend runs on one computer, the database runs on a different computer, and the browser viewing the web frontend runs on yet another computer.

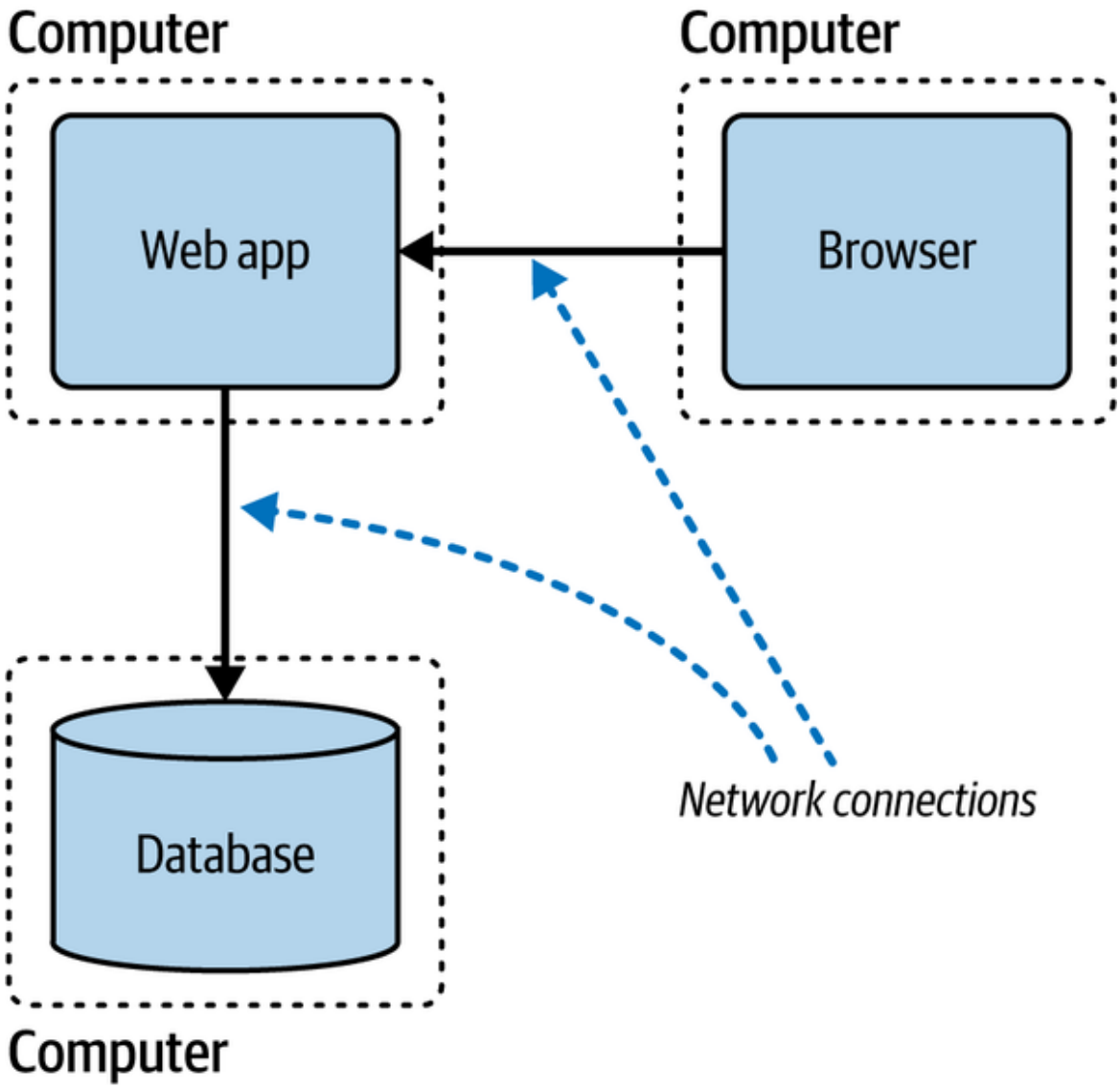


Figure 1-1. A simple web application distributed across three machines

But distributed systems can get more complex than that. Consider a microservice architecture, as shown in Figure 1-2. The arrows between the microservices indicate logical dependencies, which represent some form of network-based communication.[5] An instance of a microservice would live on its own machine. Now you have multiple services, each of which runs on different computers, communicating with one another over networks. This simplified diagram omits the fact that each of these microservices may well be talking to its own database, which will likely be on yet another machine, requiring another network hop. Oh, and each of these microservices might also be exposing its own frontend as well.

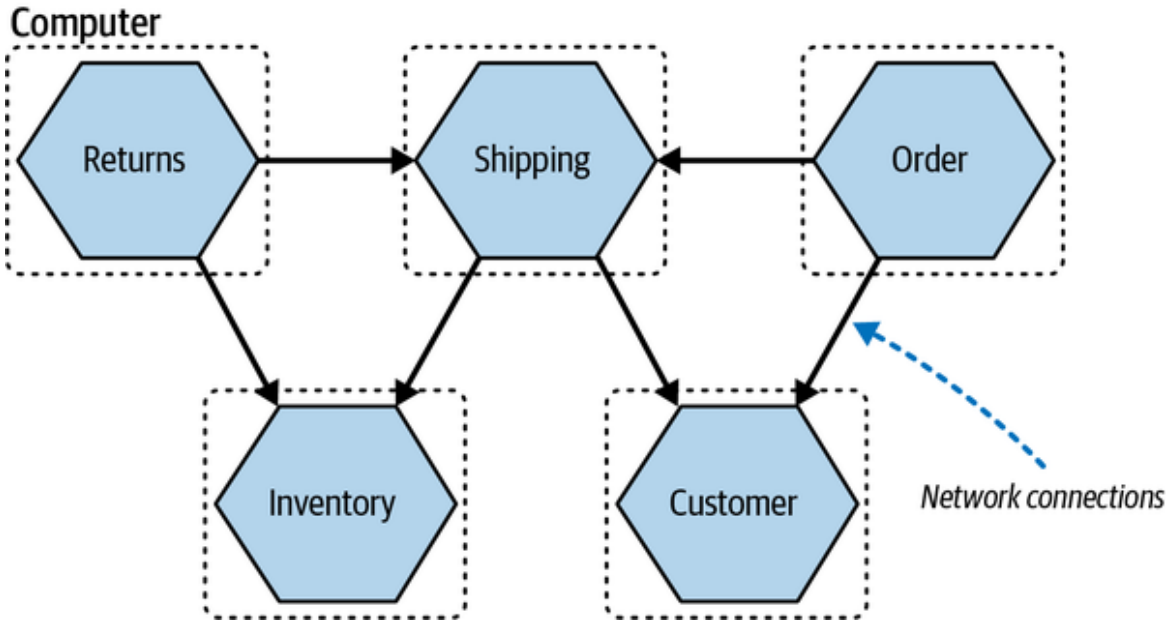


Figure 1-2. A microservice architecture

Note that the previous two diagrams are simplifications of what you might actually end up running in production. Many of the things shown on these diagrams will often have multiple copies to provide redundancy or improve the ability to handle load, which can further complicate your system.

> **WARNING**
>
> In general, the more computers you have talking to one another in a distributed system, the more complicated things get.

How Distributed Systems Can Fail (Us)

But this is all a rather dry description of what a distributed system is. It doesn't communicate what it's like to own and operate one. As a result, I turn yet again to my favorite definition of what a distributed system is:

> *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*
>
> —Leslie Lamport

This quote gets to the heart of the interesting ways in which distributed systems can fail. I've had machines vanish on me while I was using them (in one case because the wrong servers were packed up and shipped from the East Coast to the West Coast of the United States). Network cables can be severed, by accident or on purpose.[6] Domain Name System (DNS) misconfiguration can stop whole swaths of websites from working, such as the outage of the entire top-level .ru domain in 2024.[7]

Distributed systems may well be a necessity, but their nature as an interconnected system of independent machines often just exposes them to more sources of failure. Essentially, though, the vast array of failure modes that you can encounter—and the preceding list just scratches the surface—comes down to a few rules that apply to any distributed system.

The Three Golden Rules of Distributed Systems

I've found that there really are three fundamental rules that any distributed system is governed by:

1. You cannot beam information instantaneously between two points.
2. Sometimes you can't reach the thing you want to talk to.
3. Resource pools are not infinite.

These rules will influence so much of how you think about building more resilient systems, and you'll encounter them in different forms throughout the book. Let's briefly explore them so that you'll better understand the problems they create.

You cannot beam information instantaneously between two points

It takes time for information to move around a distributed system. Typically, as a developer, you use some sort of abstraction to transmit this information. Perhaps you're using an HTTP client library, or an RPC framework, or maybe you're making use of a message broker. Abstractions

in computing are incredibly useful—they allow us to hide detail and focus our minds at a higher level of abstraction. But by their nature, abstractions hide things, so you can end up forgetting what is happening behind the scenes.

When you send a piece of data from a program on one computer to another, a lot of things need to happen. The data to be sent needs to be converted into a form that can be transmitted; then it has to be broken into a series of packets, which are then sent to the destination. When the data arrives, the receiving computer has to reassemble the packets, then convert them into a format that the program on the receiving computer can understand.

All of this takes time. How *much* time can depend on a multitude of factors: the type of network being used, the distance between machines, the size of payload being sent, the speed of the computers at each end of this interaction. You may have the ability to change some of these factors, but likely not all.

It's possible that you might see the data you are sending arrive at its destination so quickly as to appear to be instantaneous. This is often the case for very small amounts of data. However, instant transmission of data between two computers is not possible.[8]

What problems does this cause? Well, it can lead to you seeing inconsistent state. You might send the same data to two different computers at the same time, but it doesn't mean that the two destinations will *receive* this data at the same time. In the worst cases, this inconsistency can be surfaced to the end user, causing understandable confusion—but if you are aware of this inconsistency, there may be ways for you to resolve it when it occurs.

In the context of distributed system design in general, the fact that you have to account for the time taken for data to move around your system will cause you a number of issues. When looked at specifically in the context of resiliency, however, it's the second of our three golden rules that arguably gives us more headaches.

Sometimes you can't reach the thing you want to talk to

As a software developer, you are increasingly working at higher levels of abstraction. You deploy applications to the cloud, configure your infrastructure using code, and often delegate work to dedicated cloud services to handle the detail of specific tasks for you. As a result, it can be all too easy to forget that under all these layers of abstractions, you have real physical computers, network cables, switches, power supplies, cooling, and everything else. Not to mention, of course, that humans play a critical role in delivering this infrastructure, and you can't expect them to be infallible.

Our computers are limited by the physical properties of the universe. That's why your data can't magically be transmitted instantaneously between two points. But this also means that you have to accept a host of scenarios in which computers you are trying to talk to can't be reached.

First, the commonplace issues. Say that you want to send data to a program on a computer, but that program has crashed and is not responding. Or perhaps the computer you are trying to reach has suffered some sort of hardware failure. Now, you might be able to reduce the chances of these things happening, but you can't reduce them to zero.

Sometimes the computer you want to talk to is there, but you just can't reach it. Perhaps someone has unplugged the wrong network cable. Or, in one real-world scenario that happened to me, perhaps rabbits have made their home in the cable ducting between buildings and have developed a taste for network cables. Are these scenarios under your control? You might have luck in reducing the incidence of network cables being accidentally unplugged, but good luck controlling rabbits . . ..

> **WARNING**
>
> Rabbits eating network cable is a sure way to increase network packet loss.

But what about issues that might be even less under your control? What happens if a leak in your data center takes out the cooling systems, meaning

you have to shut down a number of racks to avoid overheating? Or maybe someone drops an anchor out at sea and accidentally severs a major subsea network connection,[9] temporarily limiting access to one of your data centers. Or perhaps a regional conflict results in your data center taking damage from exploding drones.[10]

The people who build the hardware you use and operate the data centers your code runs in will do their best to reduce the chances of such failures happening (although there is only so much a site reliability engineer can do to stop a regional conflict). When failures *do* happen, hopefully there will be mechanisms in place so that these problems are hidden from you. But this cannot be guaranteed.

Sometimes the thing you want to talk to isn't there. This is a fact of life, and a key part of building a resilient distributed system is recognizing this fact and developing strategies to handle such a problem when it occurs.

Resource pools are not infinite

Our distributed systems need resources to work: CPU, memory, storage, network bandwidth, GPU—and yes, in this age of AI, we increasingly think of the importance of power generation. These resources are not infinite. Even if you are on the public cloud, you will be constrained in terms of how many resources are available to you, based either on how much money you can afford to spend or on what the cloud vendor can provide.

This can be a problem because if you run out of one of these resources, parts of your system can start to fail. If your computing resources are tied up handling other work and new work arrives, you're faced with the prospect of either rejecting the new work or else potentially having the whole system brought down.

Keeping our resource usage within acceptable bounds therefore becomes key. In our early chapters, we'll look at how to ensure that timeouts and retries are set to try to reduce resource use. In later chapters, we'll look at how to limit the rate of work coming into our systems to handle load better. But cutting across all of this is the general advice that, for a distributed

system, fast is better than slow. If you can process work quickly, you can free up computing resources that can then be used for other work. And of course, users are typically happier when a system is fast and responsive!

At a macro level, the current prevalence of people running LLM-based workloads places more pressure on the available compute—both for public clouds, where certain types of resources are harder to come by, or for on-premises (on-prem) situations in which the spikes in demand for memory are causing shortages and cost increases in traditional servers. I'll come back to this topic when we explore the role of AI in resiliency in Chapter 7, and when we look at disaster recovery in Chapter 10.

So I've explored what a distributed system is and introduced three golden rules. Now let's expand things somewhat and look at the most critical part of our system, the components that build, operate, and use the system itself —i.e., humans.

Distributed Systems Aren't Just Computers

In a topic with so many technical challenges, it can be easy to ignore the people who build, operate, and use distributed systems. As already mentioned previously in this chapter, the societal context in which our software systems are built and run (including aspects such as company culture and goals) has a profound impact on what and how we build them.

In the article "Programming as Theory Building" by Peter Naur, the author makes the argument that

> *programmers form or achieve a certain kind of insight, a theory, of the matters at hand. . . . A main claim of the Theory Building View of programming is that an essential part of any program, the theory of it, is something that could not conceivably be expressed, but is inextricably bound to human beings.*[11]

You'll sometimes hear discussion around the mental models that developers build into how our systems should and do work. This is exactly what Peter is referring to. Some of the angst of developers around the use of AI coding

agents comes from their fear that it can end up eroding our understanding of the systems we build, in turn exacerbating what is being referred to as *cognitive debt*. This could make it much more difficult to build resilient systems. As Margaret-Anne Storey recently observed in her blog post "How Generative and Agentic AI Shift Concern from Technical Debt to Cognitive Debt":

> *Cognitive debt, a term gaining traction recently, . . . communicates the notion that the debt compounded from going fast lives in the brains of the developers and affects their lived experiences and abilities to "go fast" or to make changes. Even if AI agents produce code that could be easy to understand, the humans involved may have simply lost the plot and may not understand what the program is supposed to do, how their intentions were implemented, or how to possibly change it.*

Humans are as central to our distributed systems as computers or networks. Fundamentally, we have to view the systems we build as a type of sociotechnical system. The term *sociotechnical* was coined to describe any complex system in which people and technology work together. In the software development industry, we tend to view *technology* through the lens of software and hardware, but more broadly the term describes the structures and processes by which work is done. We will touch on these issues throughout the book and will return to the notion of sociotechnical systems in much more detail in Chapter 13, where I will show how they can have a profound impact on the resiliency of the systems we build.

Failure Is Inevitable

As our systems have grown to involve more components, and more people, we have to confront the fact that it becomes increasingly difficult to stop parts of our system from failing.

Consider a component in your system that has a 1% chance of failing on a given day. If you have three of those components, the chance of you having a failure on a given day increases to just under 3%.[12] Once you have one

hundred components, each with the same failure rate, the likelihood of one of those components failing on a given day approaches 100%.

The core idea here is that as the number of components in your system (which could encompass computers, the parts that make up computers, services, network gear—and yes, people!) increases, it becomes more and more likely that one of those things will stop working on a given day. This means we have to shift our thinking to embrace the certainty of failure and build our systems in such a way that our systems can keep operating regardless.

I saw one example of this thinking while spending some time working at Google many years ago. In the reception area of one of the buildings in Mountain View was an old rack of machines, placed there as a sort of exhibit. I noticed a couple of things. First, these servers weren't in enclosed cases; they were just bare motherboards slotted into the rack. The main thing I noticed, though, was that the hard drives were attached by Velcro. I asked one of the Googlers why that was. "Oh," he said, "the hard drives fail so much we don't want them screwed in. We just rip them out, throw them in the bin, and Velcro in a new one."

The systems Google built assumed that hard drives would fail. It optimized the design of these servers to ensure that hard drive replacement was as easy as possible. The systems had been built to tolerate the failure of a hard drive, so while it was important that the drive was replaced eventually, chances were that a single hard drive failure wasn't going to cause any significant issues that would be visible to the user. With thousands of servers in a Google data center, it would be a daily task just to walk along a row of racks, replacing hard drives as they went. Sure, the drive failure was an issue, but it was one that could be dealt with simply. A hard drive failure was considered routine, and not worth calling someone about outside of normal business hours; it was just something they needed to know about during their normal workday.

From Safety Management to Resilience Engineering

When working in safety-critical systems, making sure things don't go wrong is important. Planes need to fly safely, nuclear reactors need to not explode, and patients need to survive routine procedures. Traditionally, ensuring these positive outcomes has been dominated by the mindset of making sure as few things as possible go wrong. This seems like a very sensible approach. However, as we've just seen, as our systems become more complex, making sure things don't go wrong in the first place becomes increasingly problematic.

In their paper *From Safety-I to Safety-II*, Erik Hollnagel et al. summarize the shift in thinking that has come to the fore over the last few decades. Describing the traditional view of safety management, in which you focus on making sure that as few things as possible go wrong, as "Safety-I," they make the case for a better way forward:

> *Safety management should therefore move from ensuring that "as few things as possible go wrong" to ensuring that "as many things as possible go right." We call this perspective Safety-II; it relates to the system's ability to succeed under varying conditions.*[13]

It's against this backdrop that the field of *resilience engineering* has emerged. Resilience engineering embraces the concepts of Safety-II, accepting the idea that, while reducing the failure of parts of a system is often a good idea, focusing on making sure that as many things go right as possible is a better way forward. So we can see Safety-I as a subset of Safety-II, as shown in Figure 1-3, with the discipline of resilience engineering being how we deliver on the goals of Safety-II.

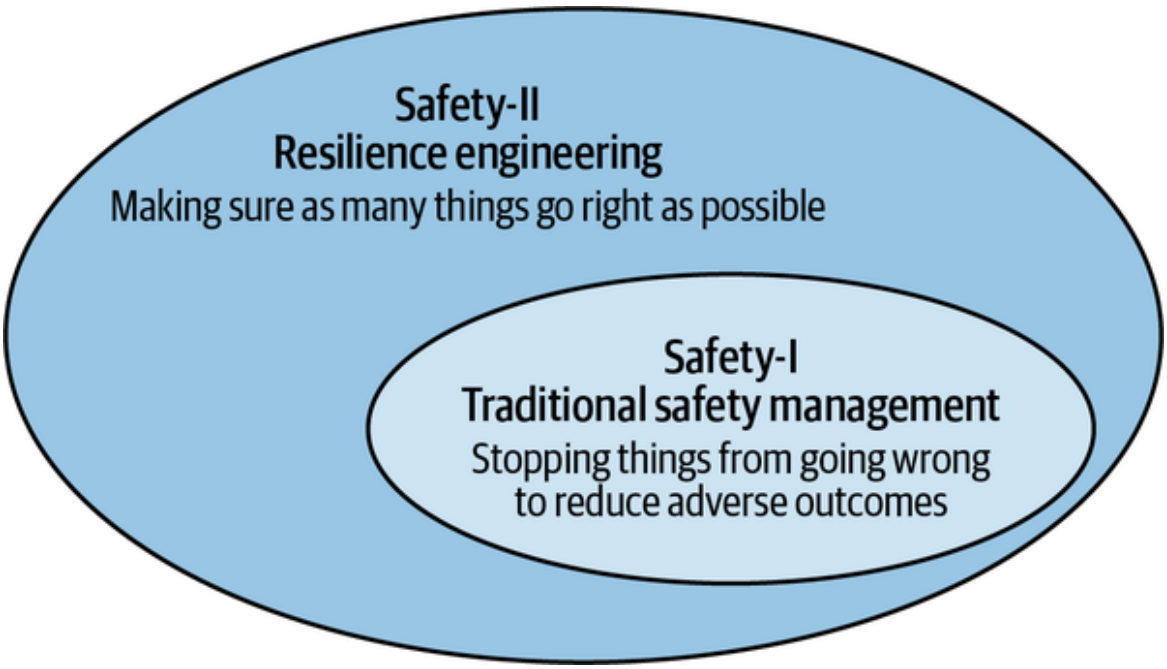


Figure 1-3. Resilience engineering goes beyond traditional approaches to safety management

> **TIP**
>
> Resilience engineering is about making sure that as many things go right as possible.

The Four Concepts for Resilience

So we've explored the concept of resiliency, and I've introduced the concept of resilience engineering—making sure as many things go right as possible. We've also seen how it's not just about the hardware and software; it's also about the people and environment in which we create and run our distributed systems.

Given these ideas, some concrete guidance is in order. What are the types of things we need to do to build resilient systems? In his paper "Four Concepts for Resilience and the Implications for the Future of Resilience Engineering," David Woods outlines a model for resilience that focuses on the following four core concepts:[14]

Robustness

The ability to absorb expected issues, such as handling a machine crashing

Rebound

Recovering after a traumatic event

Graceful extensibility

How well you can handle the unexpected

Sustained adaptability

Continual learning and transformation

I will expand each of these concepts next, looking at how they might apply in the context of building and operating a distributed system.

Robustness

Robustness is the concept whereby you put mitigations in place to deal with expected problems. From the technical viewpoint, you have a whole host of issues that you might expect: a machine can fail, a network connection can time out, a process might be unavailable. You can improve the robustness of your system to deal with these problems in a number of ways, such as automatically spinning up a replacement host, performing retries, or handling failure of a given microservice in a graceful manner. In the example from Google earlier, their engineers knew a single machine could fail, so they ensured the software they created could still operate if that was the case.

Making your system robust can also include the notion of deciding what not to do when things go wrong. Can certain features be disabled so that critical functionality is preserved? Again, this requires foresight in terms of being able to make these choices when something does go wrong.

Robustness goes beyond the technical, though. It can also apply to people. If you have a single person on call for your software, what happens if that

person gets sick or isn't reachable at the time of an incident? Potential solutions to this problem might be to have a backup on-call person or a well-documented playbook. Redundancy is crucial.

Wood's definition of robustness requires prior knowledge—you are putting measures into place to deal with things you expect to go wrong. This knowledge could be based on foresight; you could draw on your understanding of the computer system you are building, its supporting services, and your people to consider what might go wrong. But robustness can also come from hindsight; you may improve the robustness of your system after something you didn't expect happens. Perhaps you never considered the fact that your global filesystem could become unavailable, or perhaps you underestimated the impact of your customer service representatives not being available outside working hours.

One of the challenges around improving the robustness of our system is that as you increase the robustness of our application, you introduce more complexity to our system, which can be the source of new issues. You might decide to move your microservice architecture to Kubernetes, as you want to make it easier to run your microservice workloads, and you may improve some aspects of the robustness of your application as a result. But you've also introduced new potential pain points, such as likely needing more infrastructure to run Kubernetes itself, or extensive training to understand how to manage and use it. As such, any attempt to improve the robustness of an application has to be considered, not just in terms of a simple cost/benefit analysis of the initial work being done, but also in terms of whether you are prepared for the more complex system you end up with.

> **WARNING**
>
> Making your system more robust can increase system complexity and introduce new sources of instability!

A significant portion of this book will be dedicated to helping make your system robust—showing how you can put into place various mitigations for

the known issues that can occur with distributed systems.

Rebound

How well your system can recover, or rebound, from disruption is a key part of building a resilient system. All too often I see people focusing their time and energy into trying to eliminate the possibility of an outage, only to be totally unprepared once an outage actually occurs. By all means, do your best to protect against the bad things that you think might happen—improving your system's *robustness*—but you also have to be honest and understand that as your system grows in scale and complexity, eliminating all potential problems becomes unsustainable.

You can improve your ability to rebound from an incident by putting processes into place in advance. For example:

- Make replacing failed components easier—as in the example of Google using Velcro to hold hard drives in place.
- Have backups in place to better recover in the aftermath of data loss (assuming your backups are tested, of course!).
- Write and maintain a playbook that you can run through in the wake of a system outage to help resolve known issues.
- Clearly define roles and responsibilities for what's supposed to happen when an incident occurs, so everyone knows who the point person will be and how the incident response process will work.

Trying to think clearly about how to handle an outage while the outage is going on is going to be problematic due to the inherent stress and chaos of the situation. Having an agreed-upon plan of action in place in anticipation of this sort of problem can help you better rebound. We'll look at a number of patterns to help your system rebound throughout the book, and I'll also take you through a deep dive into incident handling in Chapter 14.

Graceful Extensibility

As the old saying goes:

> *No plan survives first contact with the enemy.*[15]
>
> —Helmuth von Moltke (heavily paraphrased)

With rebound and robustness, you are primarily dealing with the expected; you are putting mechanisms in place to deal with problems that you can foresee. But what happens when you are surprised? If you aren't prepared for surprise—prepared for the fact that your expected view of the world might be wrong—you end up with a brittle system. As you approach the limits of what you expect your system to be able to handle, things fall apart. How many companies were ready for the implication of a global pandemic and for the lockdowns that followed?

Flatter organizations—in which responsibility is distributed across the organization, rather than held centrally—will often be better prepared to deal with surprise. When the unexpected occurs, if people are restricted in what they have to do and have to adhere to a strict set of rules, their ability to deal with surprise will be critically curtailed.

Often, in a drive to optimize our system, we can as an unfortunate side effect increase the brittleness of our system. Take automation as an example. Automation is fantastic; it allows us to do more with the people we have, but it can also allow us to reduce the number of people we have, as more can be done with automation. This reduction in staff can be concerning, though. Automation can't handle surprise; our ability to gracefully extend our system, to handle surprise, comes from having people in place with the right skills, experience, flexibility, and responsibility to handle such situations as they arise.

Sustained Adaptability

Sustained adaptability speaks to the ability of the system to continually evolve and improve. Sustained adaptability requires us to not be complacent. As Woods puts it in his aforementioned "Four Concepts" paper:

> *No matter how good we have done before, no matter how successful we've been, the future could be different, and we might not be well adapted. We might be precarious and fragile in the face of that new future.*

That you haven't yet suffered from a catastrophic outage doesn't mean that one cannot happen. You need to challenge yourself to make sure that you are constantly adapting what you do as an organization to ensure future resiliency. Topics I'll explore in Chapter 15 such as psychological safety and chaos engineering can be key tools in helping to create a learning organization that can adapt as needed.

Sustained adaptability often requires a more holistic view of the system to see where changes need to be made. This is, paradoxically, where a drive toward smaller, autonomous teams with increased local, focused responsibility can end up with you losing sight of the bigger picture. There is a balancing act between global and local optimization when it comes to organizational dynamics, and that balance isn't static.

Fostering a culture that creates an environment in which people can share information freely, without fear of retribution, is vital to encourage learning in the wake of an incident. Having the bandwidth to really examine these surprises and extract the key lessons requires time, energy, and people—which will reduce the resources available to you to deliver features in the short term. Deciding to embrace sustained adaptability is partly about finding the balancing point between short-term delivery and longer-term adaptability.

To work toward sustained adaptability means that you are looking to discover what you don't know. This requires continuing investment, not one-off transactional activities—the term *sustained* is vital here. It's about making sustained adaptability a core part of your organizational strategy and culture.

How Resilient Do You Need to Be?

The reality is that saying "I want my system to be resilient" is on the face of it both a perfectly sensible statement and also silly. Why do I say it's silly? Well, who would say that they *don't* want their system to be resilient? Can you honestly think of a situation in which a *lack* of resiliency would be desirable?

So really the question likely is not "Do you want resiliency?" but "How much resiliency do you want?" Resiliency, as a quality attribute, is not binary.

Making your system more resilient comes at a cost. You might pay more for a better computer that has redundant power supplies. You may allow for more experienced people to be on call to react to a failure. You might invest in automating manual tasks to reduce the chance of human error. These might all be very sensible actions to take to improve the resiliency of your system, but they all cost.

That cost might be directly financial—a more expensive machine, a larger bill with your cloud provider. It could also be an opportunity cost—an engineer focusing on improving the resiliency of your system is one who isn't delivering new features. Occasionally the cost can even be felt directly in terms of trade-offs around usability—for example, you might make a decision to require users in the field to use corporate mobile devices that might not be as nice to use as their own phones but have capabilities that improve the resiliency of your system.

How much resiliency you want (or need) thus always involves a trade-off. It's a trade-off between the cost to improve resiliency and the potential impact of degraded resiliency. Losing a customer's order for an online shop isn't great, but a medical device overdosing a patient is much worse.[16]

So the need for some degree of resiliency is understandable, but deciding how much resiliency you need requires careful thought. We'll explore ways to be more specific about defining resiliency targets when we look at service level objectives in Chapter 2 and when we look at disaster recovery in Chapter 10.

A NOTE ON "FAULT TOLERANCE"

You'll find in the book that I mostly avoid using the term *fault tolerance*. In isolation, this term isn't terribly useful. When someone says "My system is fault tolerant!," what does that actually mean? Tolerant to what faults? Tolerant to what degree? Are they tolerant to a data center failing or just to a network timeout?

I have similar quibbles about *high availability*, although at least in our industry that term is typically shorthand for meaning that a given service can continue to operate if a machine fails.

When defining fault tolerance, it makes much more sense to my mind to describe what things your system can actually handle, or at the very least agree on what *fault tolerant* means for you and your team.

Summary

I covered a lot of ground in this chapter, content that I will expand on throughout the rest of the book. We looked at the nature of a distributed system and learned the three golden rules: you can't beam information instantaneously between two points, sometimes the thing you want to talk to isn't there, and resource pools are not infinite.

I also highlighted the profound shift from traditional safety management (Safety-I), where the focus was on stopping things from going wrong, toward an approach underpinned by resilience engineering (Safety-II), with a focus instead on making sure as many things go right as possible.

You were introduced to Wood's four concepts for resilience, along with some examples of how these concepts could be put into practice, which I have summarized in Table 1-1.

Table 1-1. Wood's four concepts for resilience

Concept	Description	Examples
Robustness	The ability to absorb expected issues	Dealing with machine failure, retrying calls to services
Rebound	Recovering after a traumatic event	Making computer repair easy, having backups
Graceful extensibility	How well you can handle the unexpected	Improving team autonomy, performing game day exercises
Sustained adaptability	Continual learning and transformation	Fostering a culture of psychological safety, following up after a postmortem

I also showed how the distributed systems we create are, by their nature, a type of sociotechnical system. You need an appreciation for the technology used to build the systems, together with an understanding of the people who created them. As you've started to see already, understanding our systems in this broader context is key to unlocking ways to improve resiliency.

In Chapter 2, we'll start our journey into resiliency by focusing initially on the technical side of our sociotechnical system, when I introduce some fundamental technical concepts for system stability.

---

1 Around 5.8 billion people owned a smart phone as of 2025, according to the GSMA's 2026 Mobile Economy report.

2 Aisha Down, "Cloudfare Apologises After Latest Outage Takes Down LinkedIn and Zoom", *The Guardian*, December 5, 2025.

[3] Liv McMahon, "Amazon Apologises to Customers Impacted by Huge AWS Outage", *BBC*, October 23, 2025.

[4] The concept of a *computer* can get a bit fuzzy, but this could mean a physical machine, a virtual machine, or a container.

[5] Often microservice interactions are done at a level of abstraction that might hide the network calls from the developer—but the network calls are still there.

[6] A recent example of subsea cables being damaged resulted in multiple countries in Central and West Africa losing the internet.

[7] In January 2024, the top-level .ru domain suffered an outage, apparently due to a DNSSEC configuration issue. It's always DNS! (We'll look more at DNS in Chapter 8.)

[8] I'm aware that quantum entanglement appears to achieve this. However, at the time of writing we have yet to leverage quantum entanglement as a workable networking protocol.

[9] This happens a surprising number of times.

[10] This isn't a hypothetical. In 2026, during the US-Israeli conflict with Iran and its regional allies, drone and missile strikes caused significant damage to two AWS regions. The damage was forecast to take months to repair, and at the time of writing, these regions are still not available.

[11] Peter Naur, "Programming as Theory Building," originally published in 1985; reprinted in Peter Naur, *Computing: A Human Activity* (ACM Press/Addison-Wesley, 1992).

[12] To my simple brain, the chance of failure should be exactly 3%, but apparently it's not. My very patient wife has explained this to me more than once, but it still hasn't really sunk in. I am bad at maths.

[13] Erik Hollnagel et al., *From Safety-I to Safety-II: A White Paper* (The Resilient Health Care Net: published simultaneously by the University of Southern Denmark, University of Florida, and Macquarie University, Australia, 2015).

[14] David Woods, "Four Concepts for Resilience and the Implications for the Future of Resilience Engineering", *Reliability Engineering & System Safety* 141 (September 2015): 5–9.

[15] The original quote translated from the German is actually "No plan of operations extends with any certainty beyond the first encounter with the main enemy forces," but more than a hundred years of paraphrasing has brought us to this current version.

[16] I will explore a real-world example of the latter scenario in Chapter 13 when we look at the example of the Therac-25 X-ray machine.

Part I. Foundations

The first part of this book focuses on the foundational technical aspects of making systems more resilient.

Chapter 2. Fundamentals of Observability

A key part of resilience is our ability to know what is going on, so that we can react, we can plan, and we can learn from what is happening now and from what happens later. Without a sense of what our production system is doing, we are relying on our users to tell us when something goes wrong. And without having information about what our system did at a given point in time, we will find it very difficult to understand what went wrong after the fact. This in turn will massively undermine our desire to continually adapt our system to make it more resilient.

In this chapter, I want to highlight the importance of ensuring that your system is observable. A highly observable system will give you the information needed to know whether the system is resilient enough at a given point in time, and it will become a vital part of adapting your system over longer timescales. This chapter will introduce the fundamental concepts of observability, and in later chapters I'll highlight other ways in which observability can help.

Observability Versus Monitoring

As is often the case, the concept of observability has been around for decades but arrived in the world of software development only fairly recently. The observability of a system is the extent to which you can understand the internal state of the system from external outputs. It typically requires a more holistic understanding of your software—seeing it as a *system* rather than as a set of disparate entities.

In practice, the more observable a system is, the easier it will be for us to understand what the problem is when something goes wrong. Our understanding of the external outputs helps us track down the underlying

issue more quickly. The challenge is that we often will need to create these external outputs and then use different types of tools to understand what they are telling us.

Monitoring, on the other hand, is something we do. We *monitor* the system. We look at it. Things will start to go wrong if you focus on just the monitoring—the activity—without thinking about what you expect that activity to achieve.

More traditional approaches to monitoring would have you think in advance about what could go wrong and define alerting mechanisms to tell you when these things happened. But as the system becomes increasingly distributed, you'll encounter issues that would never have occurred to you. With a highly observable system, you'll have a collection of external outputs that you can interrogate in different ways. The concrete outcome of having an observable system is that you can ask questions of your production system that you never would have thought to ask previously.

We can therefore see monitoring as an activity—something we do—with observability being a property of the system.

The Three Pillars of Observability?

Observability has previously been distilled down to three core concepts, the "three pillars" of observability: metrics, logs, and traces. Now, I want to say upfront that I think this is overly reductive and may distract from the fundamental concept of observability in general.

More appropriately, we should view metrics, traces, and logs as what they are—useful signals. Understanding them is going to be incredibly useful when it comes to making your system more observable. So let's look at each of these three signals in turn before I explain why focusing *only* on them is going to be problematic.

Metrics

A *metric* is a measurement of some kind and can come in different forms. You can use a metric to capture discrete events like the duration of a call. You can also use metrics to capture more “signal"-type information, such as a CPU rate.

You want to be able to collect additional information about a metric. For example, if you collected a response time, you’d want to know the name of the service that the response time came from, a client identifier of who made the original call. Some observability tools struggle with providing additional fields for metrics like this; this is often because they are built to support low cardinality data.

Think of cardinality as being like columns on a database table. Some tools provide only a limited number of columns for storing additional information. Being able to store high cardinality data, though, can be very useful. Coming back to our example of response time, we might also want to store a field indicating the operation taking place, IDs related to important entities that the metric applies to, maybe a customer ID, an order ID, a basket ID, and perhaps a payment method. Having that richness of information then allows an operator to interrogate the data in lots of ways. Part of the reason I like Honeycomb as an observability tool is that it was built specifically with high cardinality data in mind.[1]

Logs

Logs can become one of the most vital mechanisms to help you understand what is happening in your production system. With simpler deployment architectures, our logfiles, what we put into them, and how we treat them are often an afterthought used only after the fact to work out what went wrong. With a distributed system, they become a vital tool—not only helping you diagnose what went wrong when you find out you have a problem, but also telling you that there was an issue that needed your attention in the first place.

When you collect the logs, extract useful information from them, and correlate them with other data, they become part of a more active

observation of your system.

In order for your logs to be truly useful as part of a queryable set of data about your system, I'd strongly recommend that you adopt a fully structured log format. Traditional log formatting is done using a semi-structured format, as seen in Example 2-1. Here, the initial fields are structured—the date, time, server name, log level, and so on all appear in the same place across all log lines. This means those fields can be easily queried. The remaining portion of each line, however, is unstructured. This becomes a problem when you want to search for data that appears in the unstructured portion. In Example 2-1, for example, if we wanted to find logs associated with order `456`, we'd have to use a regular expression to sift through the long lines, as the order ID is featured in inconsistent ways across the logs.

Example 2-1. Semi-structured log entries

```
2026-04-16 13:57:56 INFO [Order] Order 456 placed for €23.34 for customer 2274623
2026-04-16 13:57:58 INFO [Packing] Pack for order with ID 456 scheduled 2026-04-
17
```

With a fully structured log record, as seen in Example 2-2, all fields are potentially sources for queries. Each log line can still have a standard set of required fields, but additional structured data, such as information specific to a given operation, can be added on a case-by-case basis.

Example 2-2. A fully structured log in JSON format

```
{"@timestamp":"2026-04-16T13:57:56.561295200Z",
"log.level":"INFO",
"service.name":"OrderService",
"message":"Order 456 placed for €23.34 but customer 2274623",
"customer.id":2274623,
"order.id":456,
"order.amount":"€23.34"}
```

STRUCTURED LOGS

Prefer structured logs over semi-structured logs.

Traces

Whereas a log line or a metric represents individual data points, *traces* pull together more data to represent a whole set of events related to the flow of calls across a distributed system. Rather than looking at one call in isolation, you can aggregate the calls to visualize what is going on, as we see in the screenshot from Honeycomb shown in Figure 2-1.



Figure 2-1. A distributed trace shown in Honeycomb, allowing you to identify where time is being spent for operations that can span multiple services

Each trace consists of multiple spans, which you can see in Figure 2-1 as the rows in the main diagram. A *span* represents something that has happened—a unit of work. Each span shows start and end times (so you know how long the span took) and also allows you to associate lots of other information. If the span is related to a SQL query being made to a database, you can put that information into the span to aid debugging, for example.

Traces come into their own not only in helping diagnose issues like latency spikes but also in helping operators better understand how the system fits together. An architecture diagram is one thing, but a real trace that explains exactly what the production system was doing for a given operation can give so much more detail. A trace from a real running system is much more

likely to be an accurate representation of how the system works than a static architecture diagram would be!

Three Pillars Not Sufficient?

Although the three-pillar model of observability initially appealed to me greatly, over time I've moved away from that way of thinking, finding that it not only is overly reductive but also potentially misses the point.

Firstly, reducing a property of a system to implementation details seems backward to me. Observability is a property, and there are many ways I might be able to achieve that property. Focusing too much on specific implementation details runs the risk of us focusing on activity versus outcome. It's analogous to the current world of IT, where hundreds if not thousands of organizations have fallen in love with putting AI into their products without any clear understanding of what they are trying to achieve!

It's arguable that all three concepts are just specific examples of a more generic concept. In the second edition of *Observability Engineering*, by Charity Majors et al.,[2] the authors make the point that our telemetry is just a set of events describing what happened to our system at a given time; a log line, metric, or trace is simply a different type of event. By giving our events a formal structure, we make it possible for them to be collected and analyzed, thereby improving the observability of our system.

When it comes to making your system observable, think about the outputs you need from your system in terms of events that you can collect and interrogate. But for a system to be observable, you also need to be able to slice and dice these events so that you can work out whether your system is behaving properly (or point you in the right direction if things are going wrong). In later sections of this chapter, we'll look at good alerting and service level objectives, which can help improve the observability of your system. But before we cover those subjects, let's look at the best way to capture our metrics, traces, and logs.

OpenTelemetry

OpenTelemetry (OTel) is a widely adopted and well-supported open standard for gathering structured events in the form of metrics, traces, and logs. Broadly speaking, there are two parts to any OpenTelemetry implementation. Firstly, you need a way for your system to produce metrics, traces, and logs. Secondly, you need something that can collect these structured events and allow you to stitch together the resulting information to make your application observable.

Getting events from your system can be done in multiple ways. To start with, there are some "zero code" instrumentation options. For example, if you're using Java, a project like OpenTelemetry Instrumentation for Java can scan libraries you may be using and automatically instrument them for you. This means that database queries, for example, can appear automatically as spans. For your own code, you can also use the appropriate OTel library for your programming language and use the API to directly send your own information. In all cases the data ends up getting sent via the standard OpenTelemetry Protocol (OTLP).

For logs, you can use the standard instrumentation techniques, or you can use a local daemon that can just pick up normal textual logs and convert and send them via OTLP. This can be useful when integrating software that doesn't natively support OTel.

On the collection side, you just need an appropriate tool that can collect and process the OTLP information for you. Because it is an open standard, you can switch out one vendor for another later on without having to change your systems instrumentation. This lack of lock-in to a specific vendor is important, as there has been a lot of change in the observability tooling space, with some early front-runners, such as Lightstep, no longer being in business.[3]

Slicing and Dicing

As we covered in Chapter 1, a huge part of resilience engineering is about dealing with surprises (graceful extensibility) and about learning and improving the system to make sure we are adapting our system to remain resilient over longer timescales (sustained adaptability). When surprised, we need to be able to interrogate our system to work out what is happening. It's common to see static dashboards that surface information about known potential problems but fail to display anything useful when something surprising occurs. So when caught unaware, we need solutions that allow us to slice and dice our information in different ways and to run queries on our data that we never knew we needed to.

Static dashboards are very rarely the way. The clue is in the name—*static*. When something goes wrong, you need to be able to dive into the details to get the information you need. While the OpenTelemetry Protocol is a standard adopted by multiple vendors, how these vendors allow you to interact with the tools varies greatly. My advice is to adopt the OTel standard for instrumenting your system and then do a road test of a few vendors to find the one that best fits your needs.

> **TIP**
>
> Static dashboards give you pretty pictures but may not surface the information you need. Make sure you can slide and dice your observability data to find answers to questions you never knew you wanted to ask.

OBSERVABILITY SYSTEMS ARE PRODUCTION SYSTEMS

Our observability tools are vital to us being able to correctly operate our systems. If you decide to self-host your observability tools, the same degree of diligence needs to be applied in terms of managing and maintaining them as you apply to the software that you write and maintain.

For this reason, looking at solutions that are managed by experts can be a good way forward. This option allows you to focus on building your application. It also reduces the chances that an issue with your system ends up taking out your observability tools at the same time.

The "Golden" Signals

One of the biggest problems I've encountered with system monitoring is people capturing too much information and at the same time not capturing enough. So where to start? As introduced in the original *Site Reliability Engineering* book,[4] there are four "golden" signals that you should capture for each service in your system—saturation, latency, error rates, and traffic.

Saturation

As I discussed in Chapter 1, one of the three rules of distributed systems is that resource pools are not infinite. When your CPU, memory, storage, network bandwidth, or other computing resource runs out, your system may stop functioning.

To keep an eye on that, you'll need to observe the "saturation" of these computing resources. The goal is twofold. Firstly, looking at the trends of your resource saturation over time (e.g., the CPU rate) will give you a better understanding of how your system functions in certain conditions.

Secondly, you can define limits that, if breached, can trigger an escalation process such as an alert.

When defining acceptable limits around saturation, it's important to understand that your system will often start to degrade well before a resource runs out. As a resource becomes more constrained, it can be harder for a computer to allocate it across competing workloads. As a result, you may want to define a utilization target—an acceptable upper level of saturation. For example, you might decide that more than 80% disk saturation is potentially problematic.

Observing the saturation trends over time is also useful, and not just for long-term capacity planning. With the right observability tool, you can look at the rate of increase of a given metric to decide what sort of escalation process is required. Let's imagine you have an 80% maximum utilization target for disk space. If the disk space usage tips over to 81%, should you immediately send an alert? If the disk space has been hovering at 80% for weeks, and you can see that it's going up by only 1% every few weeks, that is likely something that can be handled during the workday. However, if you see that the disk space has been filling up very fast, and that at the start of the day you were down at 20% but in just a few hours you're already at 60%, you might realize that this trend will quickly blow through your utilization target, and thus a preemptive alert may be in order.

Latency

We want our users to be happy, so making sure our system is fast is important. We also want our system to be fast from a resilience point of view—a fast call will tend to use fewer of our precious computing resources than will slower calls. So keeping an eye on the latency of your service is key.

It's important to separate the latencies of different types of work. At a bare minimum, separating out the latency of errors from that of successful operations is important. Many errors can be very fast—if you are looking at the overall latency of a service and you include errors in that calculation,

you might get a rosier picture of your service's performance, which could be misleading if the error rate is spiking!

You may want to go further if specific operations have different expected latencies. For example, you might want to split out read versus write latencies.

Measuring latency on a single service can be useful, but as your system becomes increasingly interconnected, being able to visualize latency across call chains in your system becomes key. This is where traces come in, as we discussed earlier.

Error Rates

As we covered in Chapter 1, expecting that nothing will go wrong isn't sensible. So we need to capture error rates for each service. As with latency, these error rates should be split out based on an appropriate error code. For an HTTP service, for example, there is a significant difference between `5xx` server error response codes (which imply a problem on the server) and `4xx` client response codes (which imply an error on the client side).

Understanding what your normal error rates are expected to be becomes important in ensuring that you can spot dramatic increases in error rates of a particular type.

Traffic

How you'll measure the traffic may depend on the nature of your service. For an HTTP-based service, for example, HTTP requests per second is the most appropriate starting point. For a service that consumes from a queue and processes work sequentially, measuring the *queue depth* (size of the queue) would be more appropriate.

Correlation Is Key

Each of the signals we've covered in this section are useful in isolation, but they become even more useful when correlated with one another or with

other signals you may be collecting.

For example, if you see CPU and network saturation increasing linearly with traffic, you can draw conclusions about how much traffic your system may be able to handle. However, if you see CPU and network saturation increasing without an increase in traffic arriving into the system, something else might be at play. Or perhaps you're seeing an increasing error rate, which might imply some other problem; checking the error rate might help shine some light on the situation, as an increase in timeouts and retries might be the cause.

Figure 2-2 shows an example of Honeycomb's BubbleUp view. Here you can see that a spike in latency for a specific service has been selected, and Honeycomb has selected some other measures for the same period of time that appear to be very different from normal.



Figure 2-2. Honeycomb's BubbleUp view, which pulls out correlated information

Bringing in other metrics and events can also be helpful. For example, if you know when releases were made, you can see the impact of recent changes. Maybe the increase in saturation seemed to occur after a new

release, which could imply an issue with the recent changes. Capturing business metrics could be useful too. If you see traffic levels and saturation increasing but no increase in orders being placed, that again is something you'll want to dig into.

Whatever other signals you decide to add to your services and system as a whole, it's important that you have the first four signals covered well. It's also important that you have a tool that makes it easy to correlate information like this.

SLAs and SLOs and SLIs, Oh My!

We've talked a lot about the things you could be doing as the operator of a system—about the mindset you need and the information you might need to gather. But how do you know if you're doing too much—or not doing enough? How do you know if you are doing a good enough job, or that your system is working well enough?

Binary concepts of a system being "up" or "down" start to have less and less meaning as the system becomes more complex. If one process of an interconnected distributed system is down, is that a problem?

As your system becomes more complex, it becomes increasingly important to take a step back and think about things from a different vantage point. Think of a beehive. You could look at an individual bee and conclude that it's not happy, perhaps because it's lost one of its wings and can no longer fly. That's certainly a problem for that individual bee—but can you extend from that any observation about the health of the hive itself? No—you'd need to look at the health of the hive in a more holistic fashion. One bee being sick doesn't mean the whole hive is sick.

We can try to work out if a service is *healthy* by deciding, for example, what a good CPU level is, or what makes for an acceptable response time. If we detect that the actual values fall outside this safe level, we can trigger an alert. However, in many ways, these values are one step removed from what we actually want to track—namely, is the system working? Remember

that resilience engineering is about making sure as many things go right as possible. The more complex the interactions between the services, the further removed we are from knowing whether things are going right.

So we can gather a lot of data, but that data by itself doesn't answer the question "Is the system working properly?" To do that, we need to start thinking a bit more in terms of defining what acceptable behavior looks like. Much work has been done in the space of site reliability engineering (SRE), the focus of which is ensuring that our systems can be reliable while still allowing for change. From this space, we have a few useful concepts to explore.

Strap in—we're about to enter acronym city.

Service Level Agreements

A *service level agreement* (SLA) is an agreement reached between the people building a system and the people using the system. It describes not only what the users can expect but also what happens if the system doesn't reach the agreed-on level of acceptable behavior. SLAs tend to be very much at the "bare minimum" level of things, often to the point that if the system only just achieved its targets, the end users would still be unhappy.

As an example, AWS has an SLA for its compute service, Elastic Compute Cloud (EC2). If an instance is not available for its published SLAs, then you get a service credit rebate. Getting some money back is nice and all, but the rebate likely wouldn't fully compensate you if that computer being unavailable means your production system doesn't work. (In Chapter 8, we'll look at how to ensure that a single machine failing doesn't bring your system down.)

UNDERPROMISE, OVERDELIVER!!

As a general rule, rather than achieving the bare minimum, you want to be operating a system that can routinely achieve far in excess of what your SLA outlines. Otherwise, while you might avoid getting sued, you'll have little margin for error and likely some very unhappy customers.

Service Level Objectives

Mapping an SLA down to a team is problematic, especially if the SLA is somewhat broad and cross-cutting. At the team level, you should focus instead on *service level objectives* (SLOs). SLOs define what a team signs up to provide. Taken across the entire organization, achieving the SLOs of every team will satisfy (and hopefully greatly exceed) the requirements of the SLA. Example SLOs could include things like an expected uptime or the percentage of calls that complete successfully within a given time frame.

SLOs are automatically measurable. The measurement of an SLO is done by collecting service level indicators, which we'll discuss shortly. An SLO also is defined for a specific period. As an example, you might define an SLO for availability on a per-month or per-quarter basis.

As generally understood, SLOs focus the team's attention on user impact. If an SLO is breached, then the user experience is suffering. It's worth noting, of course, that the concept of a "user" can vary within a distributed system. Your service might be providing direct functionality to external users, but it could also be providing functionality that other services rely on, in which case your users end up being your colleagues in other parts of the organization (or perhaps third parties).

It is too simplistic to think of SLOs as existing only as a way of ensuring that SLAs are met. Yes, if the entire organization achieves all its SLOs, we would assume that the SLAs have also been achieved, but SLOs can speak to other goals not outlined in the SLA. You might have SLOs that reflect the team's wider responsibility to the general system—for example, having a target around the time taken to merge pull requests from other teams.

If you are using SLOs for "internal" purposes like this, make sure that it is easy to separate them from SLOs that have more immediate impact. If your time to merge cross-team pull requests dips below your target of 48 hours, that might not be great, but it is likely not as important as breaching your availability SLO.

Service Level Indicators

To determine if we are meeting our SLOs, we need to gather real data. This is what our service level indicators are. A *service level indicator* (SLI) is a measure of something our software does. It could be a response time from a process, a customer being registered, an error being raised to the customer, an order being placed.

A given SLI, once captured, should make it clear whether the system is in a "good" or "bad" state at that moment. Either our system delivered against that individual SLI, or it didn't. Of course, a single "bad" SLI may not be an issue when looking at the system as a whole, which is what the SLOs are for—we use lots of individual SLIs to determine whether we are meeting our SLOs.

Some SLIs could be simple enough to be derived from a single metric, but often they require multiple bits of data from our observability system. For example, if we had an SLI like "a customer should be able to place orders and get confirmation quickly," we'd need to capture successful order placements and then make sure the order placement response time was within an acceptable bound.

Error Budgets

As covered in Chapter 1, we need to make sure our system continues to change and adapt to remain resilient. However, when we try new things, we inject more potential instability into our systems. As such, a desire to maintain (or improve) system stability could result in change being discouraged, which could in turn degrade resilience over time. A system

that doesn't change becomes brittle, not resilient. *Error budgets* are an attempt to avoid this problem.

Error budgets help us understand how much error is acceptable in a system; they build on the concept of SLOs to help us understand how much risk we can take with changing our system to improve it over the longer term without jeopardizing its resilience in the short term.

Once you have an SLO, then working out your error budget for that SLO should be straightforward. For example, you might say that your service needs to be available 99.9% of the time per quarter, on a 24/7 basis. That means you are actually allowed to be down for 2 hours and 11 minutes per quarter. In terms of that SLO, this is your error budget.

Error budgets help give you a clear understanding of how well you are achieving an SLO, allowing you to make better decisions about what risks to take. If you are well under your error budget for the quarter, perhaps you are OK with rolling out a change to switch out one of the key libraries your code depends on, or with moving the service over to an alternative cloud service on your current provider. If you've already exceeded your error budget, you may want to put off that sort of work and instead focus more of the team's time on resolving why you breached the SLO.

Error budgets are, as much as anything, about giving teams breathing room to try new things, but they are also about helping team members focus their minds on providing good user impact over both short and long timescales.

Alerts

On occasion (hopefully rarely, but probably more than we'd like), something will happen in our systems that will require a human operator to be notified to take action. A service instance may have unexpectedly become unavailable, or we might be seeing a larger number of errors than we'd expect, or perhaps the entire system has become unavailable for our users. In such situations, we need people to be made aware of what's happening so they can try to fix things.

The problem is that with a distributed system, given the larger number of calls, a large number of processes, and the more complex underlying infrastructure, there will often be something going wrong. The challenge is to work out exactly what types of problems should cause a human to be informed and exactly how they should be told. It is also vital to make sure that the alerts themselves are actually useful and don't overwhelm us.

Some Problems Are Worse Than Others

When something goes wrong, we want to know about it—or do we? Are all problems the same? With distributed systems, it can become increasingly important to be able to prioritize issues in order to decide if, and how, a human operator should be involved. Often the biggest question I find myself asking when it comes to alerting is "Should this problem cause someone to be woken up at 3 a.m.?"

As your sources of potential problems increase, you'll need to get better at deciding what types of issues should cause what sorts of alerts. Otherwise, you may well find yourself struggling to separate the trivial from the urgent. Having defined SLOs can help—breaching an SLO (or a trend that implies that breach could be happening soon) can result in raising an appropriate alert, and we'll explore that in more depth shortly. But first, it's worth discussing the challenges that occur when we overload ourselves with alerts.

Alert Fatigue

Too many alerts can cause significant issues. In 1979, a partial meltdown of a reactor occurred at the Three Mile Island nuclear power station near Harrisburg, Pennsylvania. The investigation into the incident highlighted that the operators of the facility were so overwhelmed by the alerts they were seeing that it was impossible to determine what action needed to be taken. An alert indicated the fundamental problem that needed to be addressed, but this wasn't obvious to the operators, as so many other alerts were going off at the same time. During the public hearing into the incident,

one of the operators, Craig Faust, recalled that he "would have liked to have thrown away the alarm panel. It wasn't giving us any useful information." A presidential commission's report on the incident noted that "more than 100 alarms went off" in the control room and concluded that "little attention had been paid to the interaction between human beings and machines under the rapidly changing and confusing circumstances of an accident."[5]

More recently, we have seen the issue of too many alerts raised in the context of a series of incidents with the 737 MAX aircraft, including two separate crashes that killed 346 people in total. The initial report into the issues by the US National Transportation Safety Board (NTSB) drew attention to the confusing alerts that were triggered in real-world conditions and that the board considered to be contributing factors in the crashes. From the report:

> *Human factors research has identified that, for non-normal conditions, such as those involving a system failure with multiple alerts, where there may be multiple flight crew actions required, providing pilots with understanding as to which actions must take priority is a critical need. This is particularly true in the case of functions implemented across multiple airplane systems because a failure in one system within highly integrated system architectures can present multiple alerts and indications to the flight crew as each interfacing system registers the failure . . . Thus, it is important that system interactions and the flight deck interface be designed to help direct pilots to the highest priority action(s).*[6]

So here we are talking about operating a nuclear reactor and flying an aircraft. I suspect that many of you right now are wondering what this has to do with the system you are building. Now, it's possible (if not likely) that you aren't building safety-critical systems like this, but we can learn a lot from these examples. Both involve highly complex, interrelated systems in which a problem in one area can cause a problem in another. And when we generate too many alerts or don't give operators the ability to prioritize which alerts should be focused on, those operators can be overwhelmed, and disaster can follow. From the NTSB report again:

> *Research into pilot responses to multiple/simultaneous anomalous situations, along with data from accidents, indicates that multiple competing alerts may exceed available mental resources and narrow attentional focus leading to delayed or inadequately prioritized responses.*

So think twice about simply throwing up more alerts to an operator—you might not get what you want.

ALARM VERSUS ALERT

When looking more broadly into the topic of alerting, I found a lot of incredibly useful research and practices from a number of contexts, many of which were not specifically talking about alerting in IT systems. The term *alarm* is commonly encountered when looking into this topic in engineering and beyond, whereas we tend to use the term *alert* more commonly in IT. I spoke to a few people who see a distinction between these two terms, but oddly enough, those distinctions didn't seem consistent. Because most people seem to see the terms *alert* and *alarm* as practically the same—and because any distinctions drawn between the two weren't consistent—I have decided to standardize on the term *alert* for this book.

Toward Better Alerting

So we want to avoid having too many alerts, or having alerts that aren't useful. What guidelines can we look at to help us create better alerts?

Steven Shorrock offers some ideas in his article "Alarm Design: From Nuclear Power to WebOps," which is a great read and a good jumping-off point for more reading in this area. From the article:

> *The purpose of [alerts] is to direct the user's attention towards significant aspects of the operation or equipment that require timely attention.*[7]

Drawing on work from outside of software development, we have a useful set of rules from, of all places, the Engineering Equipment and Materials Users Association (EEMUA), which has come up with as good a description of what makes for a good alert as I've seen:

Relevant

Make sure the alert is of value.

Unique

Ensure that the alert isn't duplicating another.

Timely

We need to get the alert quickly enough to make use of it.

Prioritized

Give the operator enough information to decide in what order alerts should be dealt with.

Understandable

The information in the alert needs to be clear and readable.

Diagnostic

It needs to be clear what is wrong.

Advisory

Help the operator understand what actions need to be taken.

Focusing

Draw attention to the most important issues.

Looking back over my career to times when I've worked in production support, it's depressing to think how rarely the alerts I've had to deal with follow any of these rules.

Coming back to our discussion of SLOs, an alert in the context of a defined SLO is likely to already fulfill many of the EEMUA's criteria. SLOs should be well understood by the team, so by extension an alert around an SLO breach is going to do a good job of being *relevant*, *understandable*, and *focusing*. SLOs, though, typically look at long periods of time—if your 99th percentile latency has been slowly degrading over the last month, and you alert only after that the limit is breached, the alert has likely come too late—so in this context the alert would fail the *timely* criteria. Thus, having an alert triggered well before this SLO is breached, when the trend is quickly heading in the wrong direction, seems sensible.

For an alert to be useful, it's also important to make sure that it actually gets sent. This will be a function of making sure that the alert is generated correctly in the first place, but also that the infrastructure used to distribute the alert is itself resilient. This is why many organizations make use of dedicated third-party systems like incident.io and PagerDuty, as these services are designed to be highly available and are very likely to be isolated from an issue that impacts your own system.

Separation of Responsibility Considered Harmful?

Unfortunately, the people providing information to our alerting systems and the people actually on the receiving end of our alerts are all too often different people. From Shorrock again:

> *Understanding the nature of alarm handling, and the associated design issues, can help you—the expert in your work—to be a more informed user, helping to bring about the best alarm systems to support your work.*

In situations where the people building the system are isolated from the alerts being generated, you lose a vital feedback mechanism. This is why many prefer that the people who build the system be the same people who handle first-line support for the application. They will experience firsthand the pain of poor alerting and can do the work to fix the problem. More broadly, developers who also handle support for their application are highly

incentivized to make the system easy to operate and support; this in turn can have significant impact in improving the resilience of a system.

FURTHER READING

This chapter was designed as a general overview of observability concepts rather than a deep dive. There are a few excellent resources I can recommend if you want to learn more. From an observability point of view, I'd suggest starting with the previously mentioned *Observability Engineering*, 2nd edition. An excellent companion to that book is *Implementing Service Level Objectives* by Alex Hidalgo.[8]

Many of the SRE concepts I touch on in this chapter were first popularized by the aforementioned *Site Reliability Engineering* book by Betsy Beyer et al. It's worth mentioning that the book by Beyer and her colleagues is very much a snapshot of how Google did things when the book was released back in 2016. Which is to say that while many of the insights are very valuable and have been incredibly influential, it's likely Google does things quite differently today. Plus, you probably don't work at Google, and many of the approaches outlined in that book won't translate to other companies. As of writing this, however, an updated second edition of *Site Reliability Engineering* is being worked on, and I look forward to seeing if it takes a broader view. For a guide to implementing SRE practices in more typical organizations, I can recommend *Establishing SRE Foundations* by Vladyslav Ukis.[9]

Testing Observability

During an incident, emotions are running high, and people are stressed. This is a terrible time to find out that your system's observability isn't up to scratch, as you realize that you are missing some vital piece of information that would have helped you work out what was going on. Or perhaps it's something more prosaic, like the people on call not having enough experience in how to navigate the tooling in the first place.

Given the importance of understanding our system's behavior, it therefore seems appropriate to test its observability before a major incident occurs.

Start by defining certain scenarios that you would expect your situation to handle. For example, if you had a large increase in load, would you see this clearly in your telemetry? Could your observability system process enough data to match the increased load? Obviously, an exercise like this serves a dual purpose. A load test against your system is useful in and of itself in terms of helping you understand how the system behaves under increasing load. Taking this opportunity to also validate that you have the right telemetry just makes this exercise even more worthwhile.

You could also simulate outages of services by taking them offline and checking that the expected errors and alerts surface. Again, as with the load test scenario, this simulation also serves the benefit of making sure that your system handles a service being unavailable in the way you expect. For example, does traffic fall over to a standby service? Do circuit breakers open quickly enough to allow the system to fail fast?

Incident.io runs exercises like this called *drills*—for the load testing scenario, it uses Grafana's open source k6 performance-testing tool to generate load against the system; many organizations run similar exercises. We'll return to these ideas in Chapter 15, where, among other things, I'll introduce you to Google's Wheel of Misfortune!

Observability to the Rescue of AI?

As I touched on in Chapter 1, and as I'm sure you are already deeply aware of, AI is changing both how we build our software and what kinds of functionality we can provide to our users. One of the most challenging shifts around this has been that LLM-based functionality is inherently nondeterministic. This is a topic I'll return to in Chapter 7, but in a nutshell, a deterministic program will always do exactly the same thing given the same input. The program's behavior is fully reproducible. Throughout the history of programming, most of our programs have been deterministic. On the other hand, LLMs are nondeterministic—given the same input, they can

produce different output. This makes verifying the correctness of a system using LLMs more challenging.

It is arguable, though, whether a distributed system can really have been considered to be deterministic in the first place. Sure, parts of our system, like the programs that we wrote, were deterministic in and of themselves, but the wider distributed system is more than just a lot of programs. It includes the hardware the programs run on, the people who operate them, and the environment that impacts them. Lots of those nonsoftware portions of our system can't be considered to be deterministic in the way a piece of code is.

In taking this broader view regarding the sociotechnical nature of a distributed system, it becomes harder to make the argument that a distributed system was ever deterministic. As Charity Majors, founder of Honeycomb, said to me when discussing this concept, "We pretended that our systems were deterministic and predictable . . .but they weren't, and now they REALLY aren't."

Having good telemetry was always important to understanding what our distributed system was doing. If anything, the fact that part of that system is now less deterministic than it was in the past makes having good telemetry even more important.

Summary

In this chapter, we looked at the general concept of observability, showing that while metrics, traces, and logs are useful, there is a lot more to it. Making sure your data can be queried and presented in lots of ways is key. We looked at the basic data you need to be gathering—latency, saturation, error rates, and traffic levels. Finally, we looked at the challenges of making a good alert, saw how SLOs can help us know when intervention is required, and considered how error budgets can be used to help us understand how much risk we can take with system changes.

In future chapters, we will return to the topic of observability from time to time to call out specific situations in which gathering certain data or using your observability tools can be useful.

Next up, we'll look at one of the fundamental concepts in distributed system resiliency: the timeout.

---

1 I think transparency around vendor recommendations is important. I've never received financial payment for the numerous times I've recommended Honeycomb, but cofounder Charity Majors once sent me loads of unicorn stickers, so I'm clearly not unbiased in my recommendation.

2 Charity Majors et al., *Observability Engineering*, 2nd edition (O'Reilly, 2026).

3 Lightstep was one of the pioneers in this space. It was acquired by ServiceNow in 2021 and shut down in 2026.

4 Betsy Beyer et al., eds., *Site Reliability Engineering: How Google Runs Production Systems* (O'Reilly, 2016).

5 President's Commission on the Accident at Three Mile Island, *The Need for Change, the Legacy of TMI: Report of the President's Commission on the Accident at Three Mile Island* (The Commission, 1979).

6 National Transportation Safety Board, *Safety Recommendation Report*: *Assumptions Used in the Safety Assessment Process and the Effects of Multiple Alerts and Indications on Pilot Performance* (NTSB, 2019).

7 Steven Shorrock, "Alarm Design: From Nuclear Power to WebOps", *Humanistic Systems* (blog), October 16, 2015.

8 Alex Hidalgo, *Implementing Service Level Objectives* (O'Reilly, 2020).

9 Vladyslav Ukis, *Establishing SRE Foundations: A Step-by-Step Guide to Introducing Site Reliability Engineering in Software Delivery Organizations* (Addison-Wesley Professional, 2022).

Chapter 3. Timeouts

In Chapter 1, I introduced the three rules of distributed systems:

1. You cannot beam information instantaneously between two points.
2. Sometimes you can't reach the thing you want to talk to.
3. Resource pools are not infinite.

In this chapter, I'm going to address these points head-on, primarily from the viewpoint of clients talking to remote services. Specifically, I'm going to look at what happens when a server takes too long to respond, and at how timeouts can help. Along the way I'll share a real-world example of a system outage that I was involved with and talk about the role that timeouts played.

A BRIEF NOTE

This chapter focuses on scenarios in which one computer is calling another. The scenarios are typically outlined in terms of the types of communication that are common in a microservice architecture, but fundamentally these principles apply to any interprocess communication.

Throughout, I make heavy use of the term *request*. This should not be taken to mean that timeouts are useful only in the context of request-response-based communication patterns. Here I am talking more generically about a client sending something to a server and then requiring an acknowledgment. These concepts would apply equally when sending a message to a message broker and when (hopefully) getting a response.

The Problem with Time

When one computer attempts to talk to another and that other computer just doesn't respond, how you deal with the situation is somewhat straightforward: either you give up or you try again (I'll get to retries in Chapter 4). Things become a bit more complex when a client manages to establish a connection with the service it wanted to talk to, but then the communication takes way too long.

Sometimes the issue is that the service you are talking to is overloaded and can't respond quickly enough. Other times the issue is getting through to the service in the first place—the network might be highly congested, a proxy might be having an issue, or perhaps a network configuration change has resulted in all your network packets being sent via Luxembourg despite all your computers being in a data center in London.[1]

So what do you do in such situations? The answer is to make use of a timeout. Let me introduce the pattern before we spend the rest of this chapter exploring different aspects of timeouts.

Pattern: Timeout

A time-based threshold after which an operation will be terminated.

A *timeout* is a period of time that you are prepared to wait for something to happen. Once that timeout is reached, you terminate the operation.

How It Works

Timeouts are typically applied in any situation in which you are waiting for another computer to do something. As such, they are normally implemented in or around the code that handles establishing connections or sending messages to other machines. As an example, many programming languages now provide HTTP client libraries for interacting with other programs via HTTP, and these libraries will provide some ability to specify a timeout. For

instance, Example 3-1 creates an HTTP client where the timeout will be five seconds.

Example 3-1. Creating an HTTP client in Java with a timeout

```
HttpClient client = HttpClient.newBuilder()
      .connectTimeout(Duration.ofSeconds(5))
      .build();
```

When using the constructed client, if a timeout occurs, then an `HttpConnectTimeoutException` will be thrown, as seen in Example 3-2.

Example 3-2. Handling a timeout in Java

```
try {
    client.send(request, HttpResponse.BodyHandlers.ofString());
} catch (HttpConnectTimeoutException e) {
    System.out.println("Request timed out");
}
```

These libraries typically allow you to specify a general timeout for the client (as we have in this example) while also providing a more specific timeout on a per-request basis. Client libraries for other protocols, such as for talking to databases or message brokers, will also provide the ability to set timeouts accordingly.

A client deciding to time out an operation should be captured using OpenTelemetry, and it would make sense to capture that as a span within your distributed trace.

Dedicated libraries also exist to provide more sophisticated timeout behavior; these include Resilience4j for the Java ecosystem and Polly for .NET. These libraries come into play more when we discuss concepts like retries and circuit breakers, which we will do in more detail later in the book.

When to Use It

When sending a request to another server, why not just wait until you get the expected response? What is the purpose of having a timeout?

Firstly, waiting for something that may never happen is unlikely to be desired system behavior. It's better for an application to error than just hang in an unresponsive state. Secondly, sometimes just timing out and retrying the call might get the response much more quickly. Consider a situation in which the initial request hit a server that was running very slow due to load issues. If you timed out and retried the request, it might be routed to a different server (perhaps via a load balancer) that isn't suffering from these problems.

Finally, waiting forever (or too long, for that matter) will mean you'll hold on to more computing resources, which in turn can put pressure on the system. Remember: resource pools are not infinite. If we exhaust resource pools because the resources are tied up with operations that may never finish, that can make a bad situation much worse.

When one computer opens a network connection to another computer, various resources are tied up handling the connection at both ends. As the number of concurrent requests increases, the amount of load on the server increases, potentially to the point where the ability to handle existing requests degrades, and perhaps even resulting in the server grinding to a halt, with possibly far-reaching ramifications.

You can get into a vicious circle. A server is handling a lot of requests and has fewer available resources to handle them as quickly as it would like due to resource contention. So handling the requests takes longer, resulting in the number of requests building up, further degrading the processing time of the existing requests as resource contention worsens.

So, to protect both the server and the client, it is sensible to set a threshold at which you should give up. With that in mind, how do you specify what the timeout should be?

TIMEOUTS AREN'T JUST FOR HTTP

Though many of the examples here discuss timeouts in the context of HTTP calls, any interprocess call can benefit from (or struggle with) setting appropriate timeouts. Communication with message brokers will involve timeouts too, although this is often hidden from you by message broker SDKs. A timeout you may have to deal with more frequently is the timeout for database read and writes.

So just remember that, although some of the more detailed explanation here uses HTTP as an example, these ideas translate easily over to other protocols.

Setting a Timeout

If you set a timeout that is too short, there is a possibility that you might terminate a connection that would have worked and returned a response. This isn't ideal—if the client times out the request, it has to assume the request failed, even if the server that originally received the request did eventually process it. Timing out too quickly can be even more problematic if doing so triggers a subsequent retry, as you are then carrying out a second request for an operation that could have been completed successfully the first time. If you'd waited long enough, you wouldn't be generating a second request!

On the other hand, if you set a timeout that is too long, you increase the duration for which resources are tied up handling the request (on the client, the server, and any networking infrastructure being used to provide the connection). This in turn might end up decreasing the stability of your system as it struggles to handle the load.

But there is a third element that you also need to take into account: how long do your users want to wait for things to happen? There may be no point in waiting longer for a service to respond to a request than the human user would wait.

So you want a timeout that is long enough to wait for successful responses but not so long as to cause undue stress on the system. At the same time, you need to make sure that you are responding quickly enough to satisfy the users of your software.

As you might guess, this is a tricky balancing act. So where do you start? Well, the first thing to look at is how long the service normally takes to respond.

CONNECTION VERSUS REQUEST TIMEOUTS

Some client libraries give you control over two timeouts, often described as the connection timeout and the request timeout. The *connection timeout* relates to how long you'll wait to establish a connection to the destination server. Once you are connected, the *request timeout* determines how long you'll wait for your request to be handled.

At first glance, it might seem odd that you have two variables here. You open a connection, send a request, and then close the connection, right? Well, not always. HTTP/2, for example, is designed to allow multiple requests to be sent over the same connection. This allows for improved efficiency, as you aren't having to pay the cost to open and maintain lots of connections. For a deep dive into the nuances of HTTP/2, I can recommend "HTTP2 Explained" by Daniel Stenberg.

If, however, you send only one request per connection, then when working out your timeout, you'll need to consider the sum of the connection and request timeouts.

Analyzing Existing Response Times

If you have a good understanding of how long it normally takes to process a request, you will be much better placed to make a sensible decision about where timeouts should be set. In general, when analyzing response times,

displaying this information on a histogram tends to be the best way forward. A *histogram* is a type of chart that is commonly used to display frequency distributions. In this case, you'll be looking at the frequencies of request durations.

Figure 3-1 shows the frequency of all response times for a specific server presented in histogram form. The x-axis details the duration of the request, with the y-axis representing the number of times a request took that long. This should be an easy chart to create with your observability tool of choice, assuming you're actually collecting latencies in the first place. Remember that latency is one of the golden signals we discussed back in Chapter 2; thus the capture of service response time metrics is one of the first things your system should be providing.

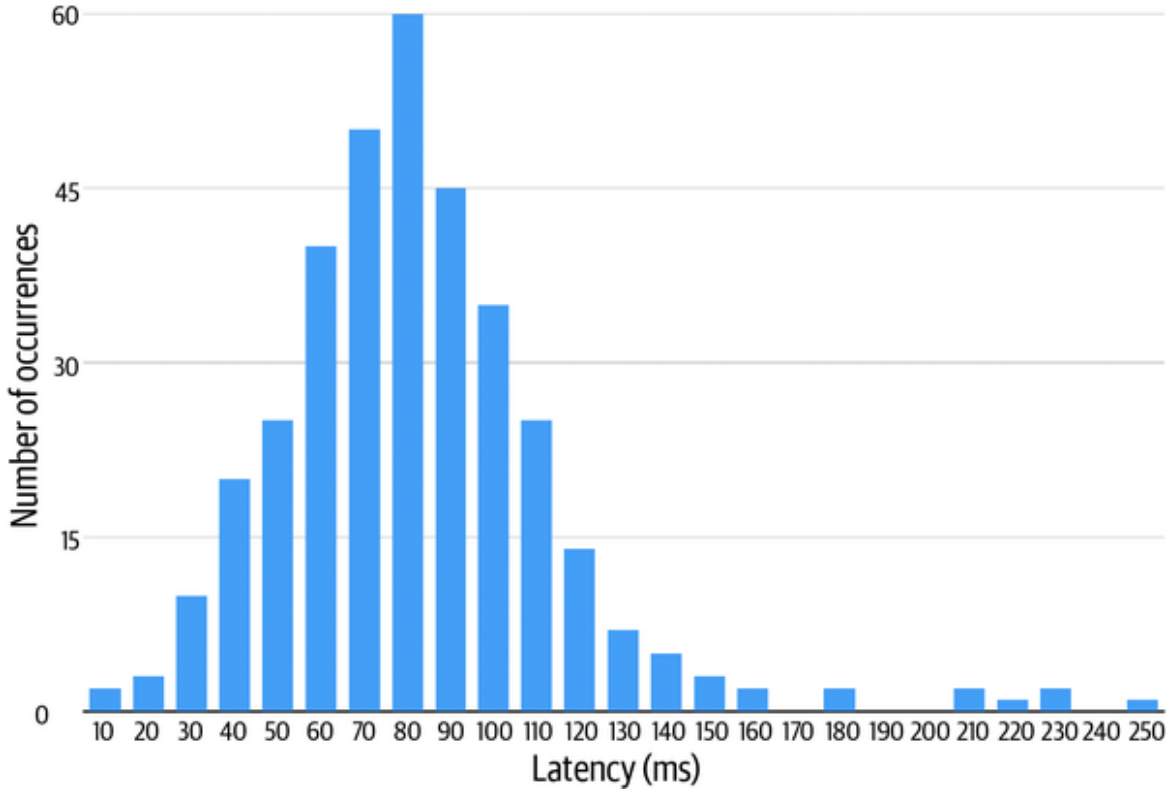


Figure 3-1. An example histogram

Displaying your latencies like this provides a better sense of how the different response times of the requests are distributed. In turn, you can make smarter decisions about things like where to set timeouts. You get to

see at a glance how response times are distributed, and that in turn can help you spot patterns in response times.

Of course, if you have your AI agent hooked up to your observability data, you could ask it to extract that information for you. Just keep in mind that you may want to check the data yourself to make sure the agent is getting its numbers right!

You could start by setting the timeout at a point where all your analyzed requests should have completed. So coming back to Figure 3-1, you could set the timeout at 250 milliseconds, as there are no instances of successful responses after this point.

The potential problem here is that you are setting the timeout based on a small number of outliers. These outliers are often referred to as *tail latencies*, and dealing with them can be problematic.

Tail Latencies

Tail latencies represent the outliers in response times—the small number of requests (perhaps 1% or less) that for whatever reason take a long time to be handled. In Figure 3-2, you can see that 99% of the requests have completed at the 180 ms mark (aka the 99th percentile); if you wanted to wait for *all* these operations to complete, you'd have to wait another 70 milliseconds, as you can still see requests completing out to 250 milliseconds.

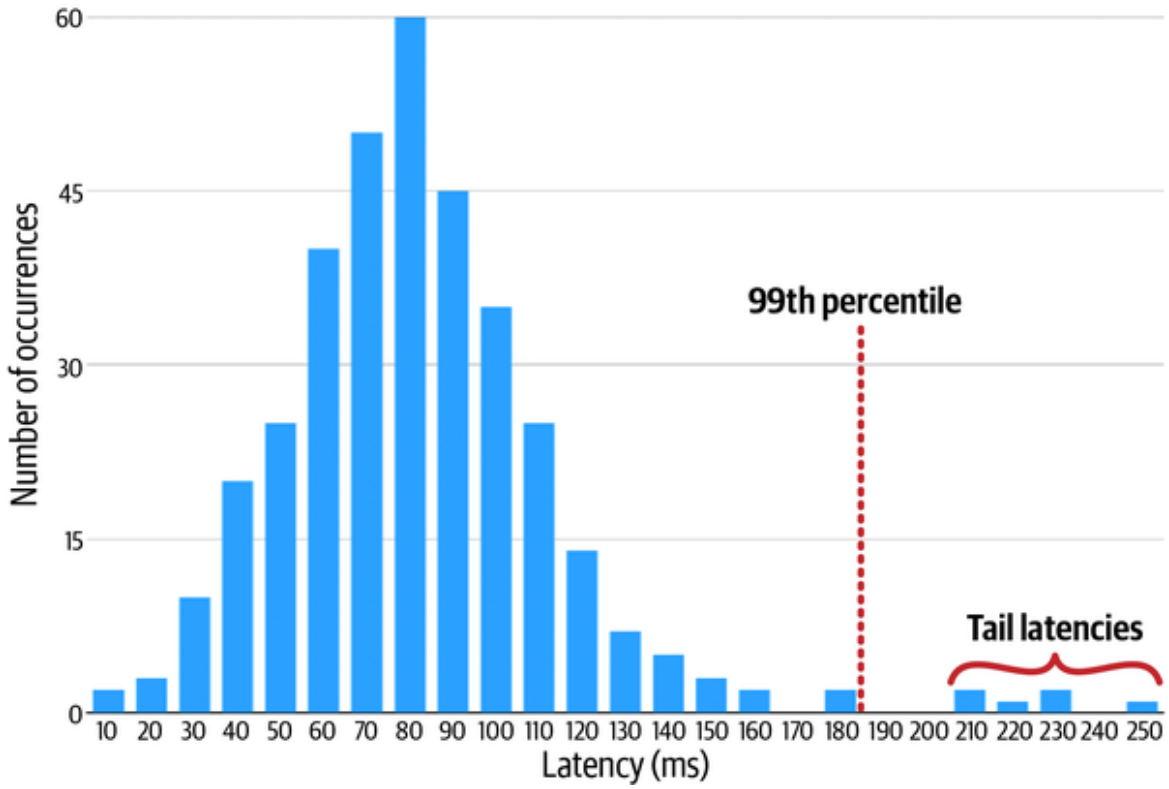


Figure 3-2. Highlighting the tail latencies

Tail latencies can occur for a number of reasons. It's possible that something unique about the nature of these requests gave them a very different latency profile—perhaps an expensive operation. Or maybe a timeout deeper in the networking stack caused additional retries, a page fault, or a request that wasn't able to benefit from cached data like the rest of the calls.

Tail latencies shouldn't just be ignored. Often, slow outliers like these have a significant detrimental impact on your users' experience, for example. They might represent only a small portion of your traffic, but if a tail latency results in an unhappy customer, that is likely something that needs to be dealt with.

Ideally, you would wait for these responses to complete. However, when it comes to setting a timeout, you may need to set it to exclude these rare tail latencies in order to keep the system stable; this allows you to free up resources faster so that they can serve other requests. So you could decide to set a timeout at the 99th percentile latency of 180 milliseconds—or

perhaps an even more aggressive timeout of 130 milliseconds, by which time just over 95% of the calls should have completed.

Once you have the right level of system stability, you can work to address these tail latencies, investigating them further so you can understand why they occur and then determine what you can do to reduce the frequency at which they occur. Techniques like request hedging can be useful to help with tail latencies. With request hedging, a client sends a request to multiple server backends and then uses the fastest-responding backend, cancelling the calls to the others. This isn't a trivial technique to implement, of course; not only does it assume that cancelling requests is possible, but you also have to be OK with the extra load this will generate on your system. However, once you're at the point of fixing tail latencies, you're likely moving on to the more complex things anyway!

Tight Latency Bounds

Although in general setting a timeout based on a percentile response time is an excellent place to start, in some niche situations it might not be appropriate. Consider a scenario in which the distribution of response times is very narrow—for example, the 99th percentile might be close to the 50th percentile.

In such a situation, a small change in response times could result in you timing out a large number of requests that would otherwise have succeeded. The typical solution is to add a bit of a buffer to the timeout to reduce the chances of it happening.

How Many Requests Can You Handle?

Working out a sensible timeout based on how long it takes to process requests is all well and good, but as a service handles more and more requests, you need to ensure that it remains stable while doing that work.

You might start by setting the timeout based on the 99th percentile but then realize that at that level the number of concurrent requests your server may

be asked to handle is too great. Remember, if you give a request more time to be processed, you can end up with more requests being handled at any given point in time. As a result, you may have to compromise your timeouts in an effort to maintain system stability.

Reducing a timeout to try to reduce the load on a server can have some unforeseen knock-on implications. For example, if you reduce the duration of a timeout, you may see an increased number of requests being timed out. If these timeouts result in retries, you might end up actually increasing the number of requests coming into the system!

There is an interesting association between the number of requests coming in, how long they take to process, and the resulting concurrency of a system. For example, if you can reduce how long it takes to process a request, this in turn can reduce how many requests you're trying to handle in the first place. Requests can be processed and responses returned to the clients more quickly, freeing up computing resources.

I'll explore this further when I introduce Little's Law in Chapter 12. I'll also show how a server can protect itself from having too much work in Chapter 6.

> **WARNING**
>
> Setting a timeout too short—or too long—can increase the load you're already dealing with and further exacerbate the problem.

Using User Expectations

By understanding latency distributions and taking into account what load your system can handle, you can set a timeout threshold accordingly. But what happens if the end-user expectations don't match?

Imagine a scenario in which you have a service that's rendering a complex HTML page that is going to be sent to the user's browser. Just by looking at the normal response times and balancing that information against

acceptable load, you've decided that you can afford to wait up to six seconds for the request to complete, and you set a timeout accordingly. However, you know that your users won't wait that long for a page to load and in many cases will simply give up or just retry a page before the six-second threshold is reached.

In a situation like this, in which the end-user expectation for acceptable latency is lower than what is needed to maintain system stability, adopting a more aggressive timeout might make more sense. If you are overly generous and wait longer for the request to be processed, but the end user has already given up and decided to retry, you can end up with two issues. Firstly, resources are tied up handling a request that is no longer needed. Secondly, the user has initiated a second request that you also have to process.

In this example, you might look at how long your users would wait for the page to render. You find that most of them will wait only up to four seconds, so you can set the timeout for rendering the web page at four seconds instead. This specific problem may, of course, have other solutions. For example, you could break the rendering of the page into multiple less expensive calls that could be processed in parallel, have part of the page loaded from a cache (such as a content delivery network), or even start rendering some content while the rest loads behind the scenes. If something is taking too long, it's still OK to look at making it faster!

FAST IS GENERALLY BETTER THAN SLOW

If you can process requests faster, you will have fewer concurrent requests. This will in turn likely reduce resource contention and allow you to handle more traffic. So in general, finding ways to process requests quickly is a good idea. Just be aware that some techniques to speed up request processing (like caching) can have other trade-offs that need to be considered.

Fault Injection and Testing Timeouts

Simulating timeouts can be a great way to make sure that your timeout configuration is working appropriately. A common way to do this is to set up a test scenario and use fault injection to add delays in interservice communication in order to trigger timeouts. There are a host of ways to do this. Some libraries simulate timeouts purely on the client side by mocking the calls, such as through the use of WireMock. But you could also implement jitter at the real network level. For example, the Linux `tc` (traffic control) command is able to add delays in network-based communication.

Doing pure simulation might be useful too. At the time of writing, Marc Brooker has shared an early version of Stability Sim, a web-based simulator for looking at distributed system failures, which you can see in Figure 3-3. It's very early days, but I'm hopeful that this tool (or others like it) will help developers more easily simulate failures and test their approaches much more quickly.



Figure 3-3. Using Stability Sim to simulate a timeout with exponential backoff

Timeouts and Call Chains

In a distributed system, where operations consist of chains of calls between services, the question of timeouts becomes a bit murkier. Consider Figure 3-

4. The `Enrollment` service sends a request to `Subscription` with a 400 ms timeout. To handle this request, the `Subscription` service in turn needs to send a request to the `Media Library`, but here `Subscription` has specified a 750 ms timeout. Interactions like this aren't uncommon in finer-grained microservice architectures.



Figure 3-4. Mismatched timeouts in a chain of calls between services

You might already spot the potential problem here. `Subscription` might wait up to 750 milliseconds for the `Media Library` to respond. However, `Enrollment` will wait only 400 milliseconds for `Subscription`. This means that `Enrollment` could give up and time out its request, but `Subscription` will still be trying to process the call. You can see this scenario play out in Figure 3-5.

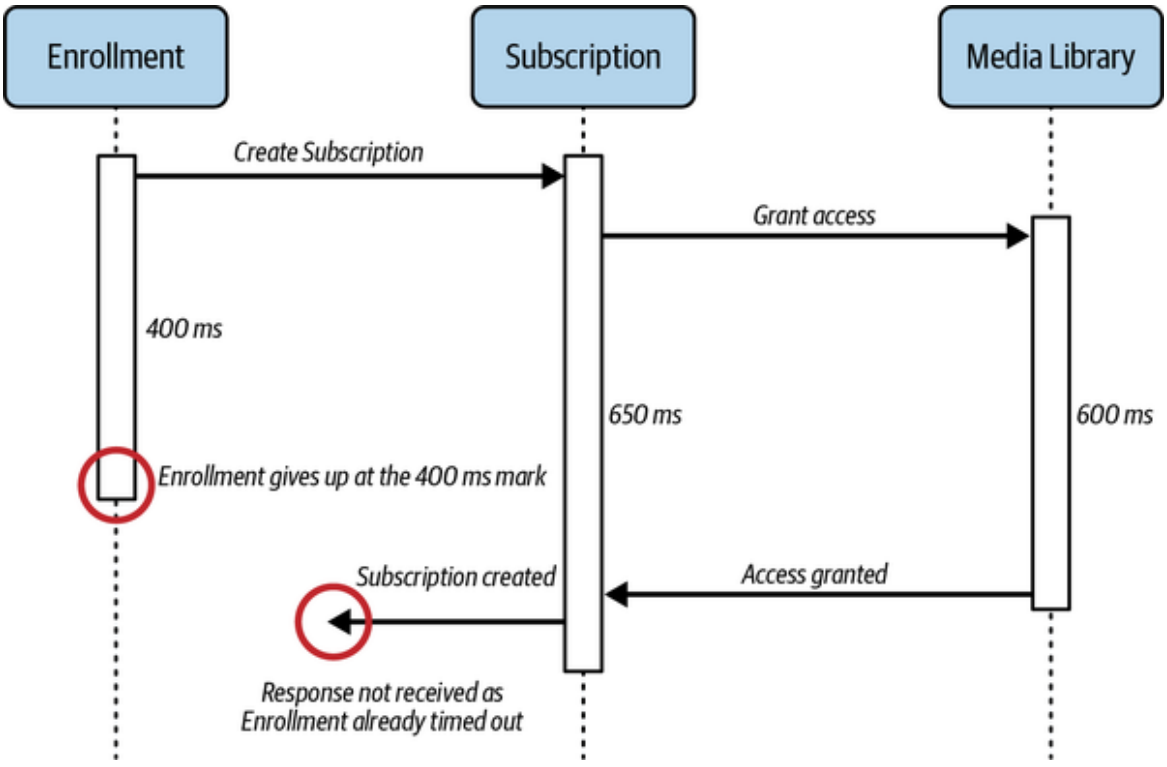


Figure 3-5. An early timeout leaves work still happening in services further down in the call chain

Even if the call from `Subscription` to `Media Library` did succeed after 400 milliseconds, `Enrollment` has already given up. You know there is no point in waiting 750 milliseconds to carry out some work if your consumer is giving up after 400 milliseconds. A fix here could be to reduce the timeout for the call from `Subscription` to `Media Library`. In this way, it's similar to the scenario of timing out a call based on user expectations—you shouldn't keep trying to render a web page long after the user requesting it has already given up.

Another issue with `Enrollment` giving up and erroring back to the user is that some of the work has been done. For example, the `Media Library` may have updated its database with information about the new subscription, even though the error to the user might have implied that the subscription hasn't been created. In Chapter 4, we'll look at the concept of idempotency, which can help here.

If we did want to change the timeout that `Subscription` used when calling `Media Library`, then that would require a bit of thought. In this scenario, we would certainly want the timeout to be sub-400 milliseconds, but just how low we set it may depend on what other work `Subscription` is doing as part of serving the request to `Enrollment` in the first place—and we'd also have to take into account the other clients that might also be calling `Subscription`. So ideally we'd want this timeout to change dynamically. Let's look at a pattern that can help resolve this complexity next.

Pattern: Timeout Budget Propagation

Propagate the timeout budget across a series of operations to ensure that work is terminated when the overall timeout expires.

If one service can kick off work in other services, it can be useful to keep track of the time remaining before a timeout is breached and have each subsequent operation use that budget to decide when to stop work. For this to happen, the timeout budget needs to be propagated with calls to other services.

How It Works

Let's reexamine the scenario in Figure 3-4. The problem we encountered originally was that the `Subscription` service was waiting so long for `Media Library` to respond that the initial call from `Enrollment` to `Subscription` might have already timed out. If instead you had told `Subscription` how long it should wait, it could give up more easily.

In Figure 3-6, we have a 500 ms timeout set for the overall operation. At the point where `Enrollment` calls `Subscription`, there are 475 milliseconds left in our budget, and this information is passed along as part of the request. When `Subscription` starts its work, it knows that it has only 475 milliseconds left at most. In this example, it's already used up 100 milliseconds before it calls `Media Library`, meaning only 375 milliseconds are left in the budget. You also have to consider that when `Enrollment` gets a response from `Subscription`, that response itself needs to be processed —so that time may need to be taken into account as well.



Figure 3-6. A timeout being propagated across a set of calls

In this example, we are propagating a timeout budget in terms of remaining milliseconds. This might be OK when you expect the request to be handled pretty quickly, but when there could be a delay between the client sending a call and the server handling it, this approach might be problematic. Consider sending a request over a message broker, for example. The message might be delivered into a queue, but it could take a while before the message gets picked up by the recipient. If you just pass along a number of remaining milliseconds in the message, then it's like the countdown stops until the message is picked up, whereas you'd actually want any delay in processing the message to be accounted for.

An alternative is to instead send an absolute time by which the operation should have been completed by including a timestamp as part of the

message payload. This then allows you to take the timeout deadline from the current clock time and determine how much time you have left—although if the response is also going back on a queue, you'd somehow need to account for the time taken for the response to be picked up and processed. For this to work, you'd have to have confidence that the clocks in the system are synchronized fairly closely.

One of the challenges here is that the various client-server protocols will need to allow for the timeout to be propagated. If you're sending HTTP requests, for example, the timeout could be sent in a custom header; if you're sending messages over a broker, the timeout could be sent as a piece of metadata. Sending the information is only part of it, though—you need to ensure that the servers receiving this information know what to do with it. Some protocols already support timeout budget propagation—for example, in gRPC this is known as deadline propagation.

Maintaining a single timeout for the entire operation like this also allows for retries. Each retry removes time from the overall deadline. I'll look at retries in more detail in Chapter 4.

When to Use It

Timeout budget propagation primarily deals with the problem of excess work being done. Without timeout propagation, the initiating service will time out, and this will be relayed to the user in some form. The fact that work is still being done by other services in relation to this now aborted operation is less than ideal. In situations where you have concerns about your system's ability to handle the excess resource consumption caused by these aborted operations, timeout propagation is worth considering.

While I can see the value of timeout budget propagation, I've not seen it commonly implemented in the wild. I suspect the main issue is that setting up the required plumbing to propagate this budget through a chain of calls is something that requires a lot of work to get right and maintain. Consider the fact that a lot of timeouts are set on a connection pool level—to

implement timeout budget propagation, you'll need some way to override those default timeouts.

CLOCK DRIFT AND TIMEOUTS

One nasty problem can cause issues with the concept of timeout propagation and with other aspects of distributed computing. It turns out that clocks in different computers don't agree.

Two clocks in two separate computers, even if they are set at the same time, will drift apart over time. Otherwise known as *clock drift*, this variation between the clocks in different computers causes a number of challenges around distributed systems. Clock drift is not limited to computers; you've likely seen it in your home—two clocks set at the same time will, over a period of days and months, end up drifting apart. The difference between the two clocks is called *clock skew*.

Clock drift causes a major issue with the concept of timeout propagation. Consider a simple scenario in which service A is calling service B. The local clock time in service A is 13:34:00.*100*. Service A sends a request to service B and wants the request to time out in 75 milliseconds, so it sends along a timeout timestamp of 13:34:00.*175*—a time that is, from service A's point of view, 75 milliseconds in the future. Unfortunately, service B's local clock is 150 milliseconds ahead of service A. When the request was sent from service A with a timeout of 13:34:00.*175*, service B already thought it was 13:34:00.*250*. So the moment the request is received, service B thinks it should be timed out immediately.

Another issue with clock skew with respect to timeouts stems from the attempts to fix it. The Network Time Protocol (NTP) exists to try and reduce the variation in clocks across computers. When a computer updates its time based on this protocol, the local clock time can jump forward or back a number of milliseconds. When working with tight latency bounds, this variation can make a big difference.

Atomic clocks are extremely accurate when compared to electrical clocks such as those found in normal computers, and this is why atomic clocks are often used to reduce clock skew to more tolerable levels,

often through the use of the Precision Time Protocol (PTP). Some cloud vendors actually provide access to the use of their atomic clocks for this purpose. That complex distributed solutions often end up relying on atomic clocks inside satellites in geosynchronous orbit might help explain how big a deal clock skew can be.

Other Timeout Considerations

In "Understanding Real-World Timeout Problems in Cloud Server Systems" by Ting Dai et al.,[2] the authors looked at a number of timeout-related bugs in cloud software. One of the things that their paper shared was that in many cases error messages were lacking when timeouts occurred, leading to problems in diagnosing the issue; this was something that happened in 83 of the 159 bugs they examined. This included some bugs where timeouts were not set, though they should have been. It's tricky to imagine having an error message that reports a missing timeout—if you knew enough to write such an error message, you probably would have just put the timeout in place! However, Dai et al. did cite examples in which timeouts were set but not reported on. In the case of one bug cited in Cassandra, if a timeout occurred when writing data to storage, no error was reported to the client.

If timeouts do occur, make sure these are logged appropriately, and if sensible to do so, this information should be exposed to the clients. This can help greatly in situations where timeouts are set too short, which can be just as much of an issue as timeouts that are too long.

Case Study: AdvertCorp

To help illustrate the role that timeouts can play with respect to system resiliency, I want to introduce a real-world project that I worked on. AdvertCorp (the company's name and details are changed to protect both

the guilty and the innocent!) provided classified ads through a very popular website.

The project I was working on was tasked with consolidating a number of existing services that were used to deliver similar functionality for various types of advertisements. Existing functionality for the different types of ads was slowly being migrated into the new system we were building, code-named Sauron, with many other types of ads still served up from older legacy applications that hadn't yet been migrated. To make this transition transparent to the end customer, we intercepted calls to all types of ads in Sauron, as outlined in Figure 3-7. This allowed us to perform some transformation of the pages that were coming from the legacy applications. Sauron, in effect, served the entirety of AdvertCorp's website, even if a lot of the content came from other sources.[3]

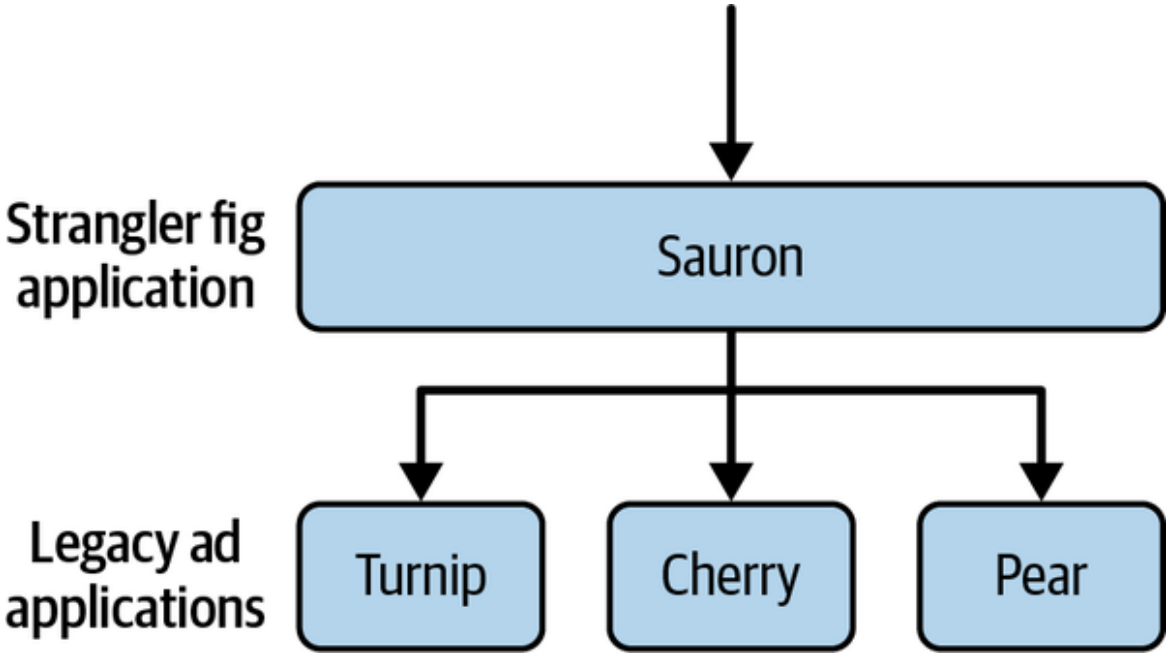


Figure 3-7. A strangler fig pattern being used to direct calls to older legacy systems

We had just moved the highest-volume and biggest-earning product over to the new system, but many of the ads were still being served by a number of older applications. In terms of both the number of requests and the money made by these legacy applications, there was a very long tail—many of these older applications received small amounts of traffic and generated small amounts of revenue. The new system had been live for a while and

was behaving very well, handling a not insignificant load. At that time we must have been handling around six thousand to seven thousand requests per second during peak, and although most of that was very heavily cached by reverse proxies sitting in front of our application servers, the searches for products (the most important aspect of the site) were mostly uncached and required a full server round-trip.

One morning, just before we hit our daily lunchtime peak, the system began behaving slowly and then started failing. We had some level of monitoring on our new core application, enough to tell us that each of our application nodes was hitting a 100% CPU spike, well above the normal levels even at peak. In a short period of time, Sauron became unresponsive, and the entire AdvertCorp website went down.

Overload

In the wake of the outage, we realized that the CPU spike we observed was caused directly by a drastic increase in the number of requests that the Sauron service was handling. Typically, each instance of Sauron would expect to have around 40 concurrent requests at peak. This is a relatively small number given the overall site traffic, but the majority of the inbound requests were handled effectively in a caching layer. In the lead-up to the outage, however, the number of concurrent requests we were handling went from around 40 to over 800. Sauron was a Java-based application, and each request that was being handled was mapped to an operating system thread. This meant that the underlying machines were suddenly trying to handle a vast number of threads, which ultimately led to the high CPU rates we saw. I couldn't appreciate it at the time, but in reality this was a great example of the perils of resource saturation.

Customer-Driven Denial of Service

Initially, we were concerned that the drastic increase in requests coming into Sauron was caused by a malicious attacker—in other words, we were worried we were the victim of a denial of service (DoS) attack. But then we

found that the source of the “attack” was in fact our own customers. We were seeing repeated retries coming from our customers, and we soon realized why. When customers requested certain pages from our website, the pages appeared to hang. The customers got tired of waiting and decided to refresh the page, in turn increasing the number of requests we were trying to handle.

Page Load Times

So why were some pages taking so long to load? Well, remember those downstream legacy applications? When a customer requested a page served by an older legacy application, the Sauron application worked as a proxy. Sauron served the request from the customer’s browser and then created its own request to the legacy application. This allowed Sauron to modify the page being rendered.

The way this worked was that Sauron made use of an HTTP client that managed a pool of workers that could be used for making requests to the legacy applications. This pool worked effectively as a throttling mechanism, capping the number of calls we could make. So when a request came in to Sauron that needed to be routed to one of the legacy applications, a worker from the connection pool was used to make the subsequent request to the legacy application, as we can see in Figure 3-8.

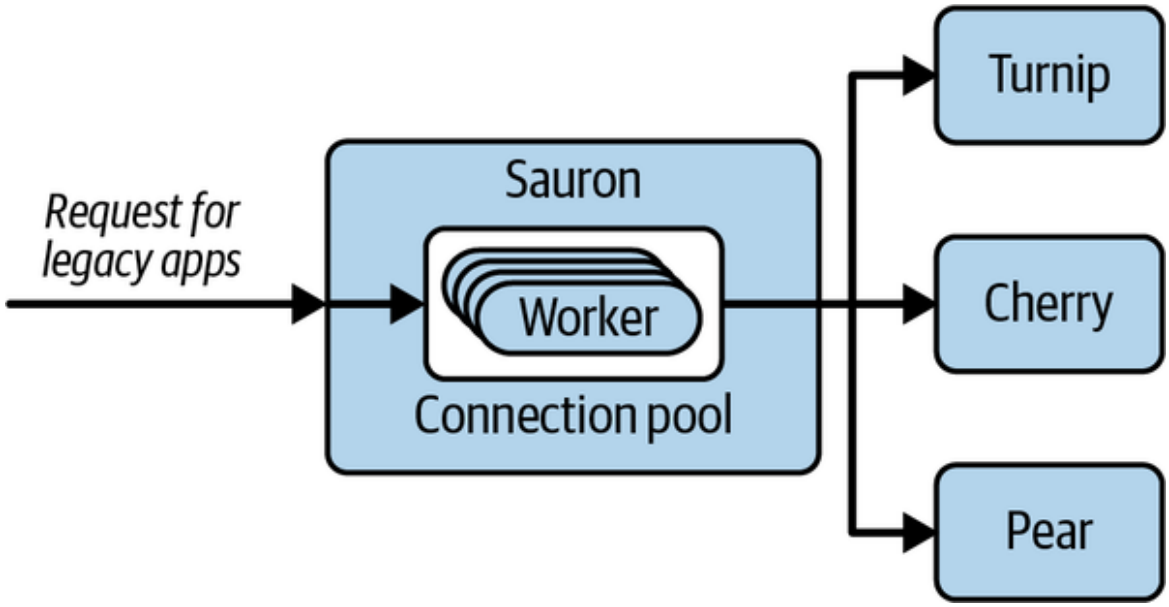


Figure 3-8. A connection pool used for making calls to legacy applications

It seemed that one of the downstream applications—codenamed Turnip—had started responding very slowly. In fact, it would allow a connection to be established and then not respond. The HTTP client we were using had a default timeout of 30 seconds. This meant we would wait for up to 30 seconds for Turnip to respond before timing out, as shown in Figure 3-9. This was for a system where our target render time to the browser was in the region of 4–6 seconds, and most customers wouldn't wait longer than 10 seconds. So the customers would give up waiting for the page to load before our timeouts expired. They would then either refresh the page, generating more load, or else give up altogether.

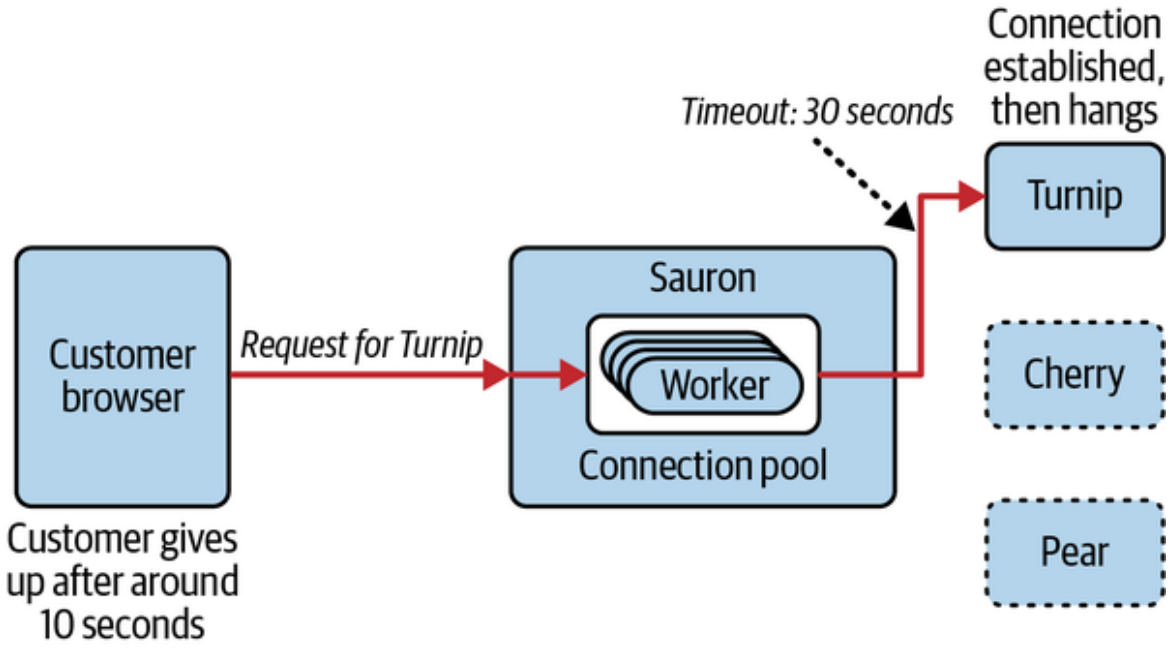


Figure 3-9. Timeout being reached due to the unresponsive Turnip application

This example of a default timeout being inappropriate is not uncommon. In Dai et al., one of the bugs they highlighted occurred in HBase (a distributed database). The default timeout for HBase clients was set to 24 days,[4] meaning that clients would hang for this period of time if the server failed. In reality, when an HBase server failed, you could normally expect recovery within 10 minutes, so the default timeout was changed to 20 minutes.

In our case, we realized that this default timeout was inappropriate on two levels. Firstly, it was way too long based on what customers would wait for. More importantly, though, the long timeouts were contributing to resource contention, which was a significant factor in the system collapsing. By reducing the timeouts for requests sent to the legacy application, our hope was that if we had issues with one of our legacy applications again, we would time out more quickly, reducing resource contention.

But we weren't done there. The purpose of Sauron's connection pool was to throttle the number of concurrent requests. This pool size was set to 20—significantly smaller than the 800 concurrent requests we were seeing. Even if all the workers were blocked waiting for a timeout, we shouldn't have seen anything like the number of requests blocked in Sauron itself; surely if there were no workers available, we would have given up trying to handle

the excess requests. We were clearly missing something—so what was going on?

Missing Timeouts

After some more digging, it became apparent that the bulk of the blocked requests were actually waiting for a worker to be available in the connection pool. With a pool size of 20, once all workers were occupied, inbound calls that wanted another worker would end up blocking until one became available. Only after a worker was actually made available would the worker's own 30-second timeout start to apply. Given the long timeout for workers to be made available, these inbound requests would be waiting a long time for a worker to become available. In effect, the HTTP client library connection pool became a bottleneck, as we see in Figure 3-10. It was a classic case of resource contention.

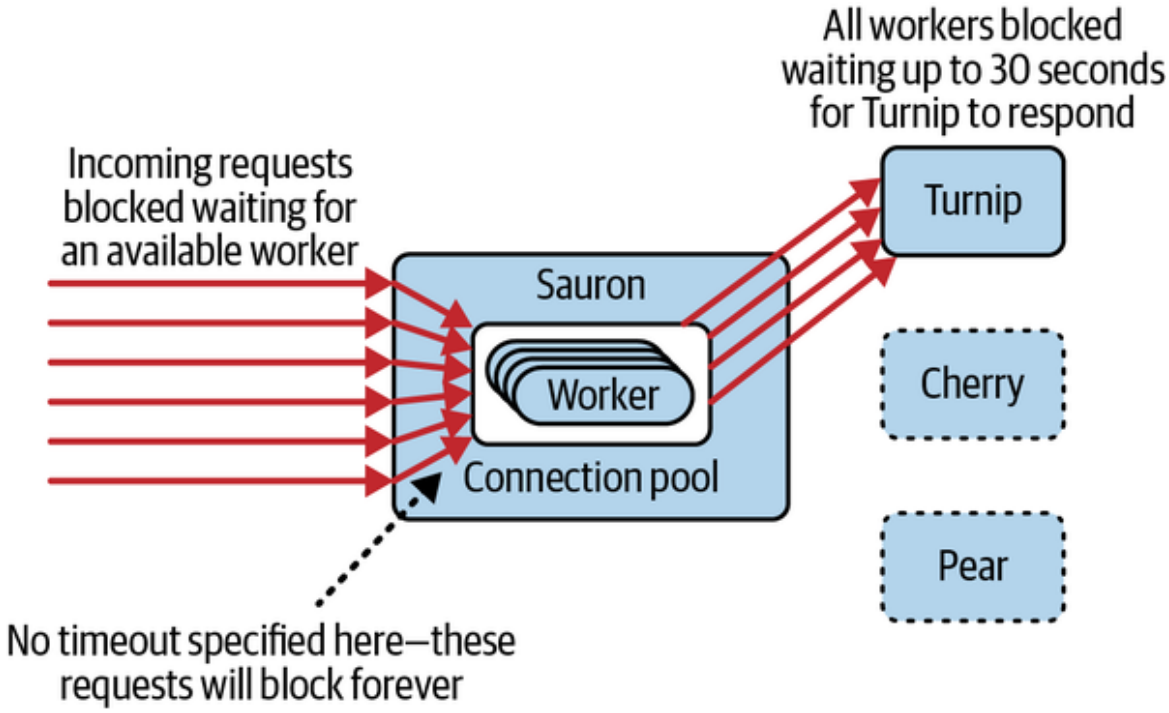


Figure 3-10. The Sauron worker pool became a bottleneck and caused requests to pile up

It turns out the HTTP client library we were using did allow specifying a different timeout—namely, for the amount of time to wait for a worker to

become available from the pool. The problem? By default, this wasn't set, meaning calls would potentially block forever waiting for a worker.

As we'd already established, our customers would end up refreshing their browsers after a while. But the original requests that they had sent were still being processed by the system and wouldn't go away until a worker became available. So a request would come in from a customer, it would block, and the customer would try sending in another request, while the original request was blocked, potentially forever. The number of requests Sauron was handling skyrocketed in a very short space of time. The net result? Sauron collapsed under the weight.

Coming back to the paper by Dai et al., of the 156 real-world bugs the authors examined, nearly 48% of them were caused by wrongly set or missing timeouts—the single biggest cause of timeout-related bugs they looked at. So it's good to know that at AdvertCorp we at least were in good company.

The fix here was obvious. We needed to apply a timeout for workers to become available. With two timeouts now in the mix—one for a worker to become available, and another for the worker itself to do the job—we had to reason about them as a whole. If we set each timeout at 5 seconds, an inbound request could wait up to 10 seconds. In the end, we decided to be quite aggressive with the timeouts we set. These legacy applications weren't as critical as other parts of our system, and therefore the risk of timing out requests that might have worked was tolerable, if the trade-off was increased certainty that the system as a whole would keep running. I don't recall the exact timeouts we used, but I think it was a second for each, meaning that inbound requests for legacy applications should at worst block for 2 seconds in total before the timeouts kicked in.

A Confluence of Events

It can be tempting to look for a single cause of a system failing. As we'll explore across this book, this way of looking at failure is overly simplistic, and in fact can be actively dangerous. It would be easy to say that the root

cause of AdvertCorp's website outage was the Turnip legacy application taking a long time to respond, the reasoning being that if it had responded more quickly, the long timeouts wouldn't have been an issue. But if the timeouts had been shorter, then even if Turnip started misbehaving, we would likely have protected the system. Likewise, the missing timeout on the connection pool was the real kicker; it's possible that if we'd just had that timeout in place, the system might have remained more stable, or it might have at least taken long enough to crash that we would have had a chance to resolve the issue without the system going down.

When looking at failures, it's always worth digging deeper. It's nearly always a number of factors that result in a failure. So if you want to reduce the chances of that type of failure happening again, then you may need to make multiple changes. In the case of AdvertCorp, we changed the timeout for the legacy applications and set a default timeout for the connection pool. But we weren't done there. We'll come back to AdvertCorp later in the book to look at other things we could have done, and other things we actually did, to try and improve the resiliency of our system.

Summary

On the face of it, timeouts are a conceptually simple idea. But as you've seen, there is still a lot of nuance here. I'm aware that I've covered a lot of different aspects of timeouts, and chances are that if you are just dipping your toe into the water here, you may feel a bit lost. So where should you start if right now you are unsure what your timeouts should be?

Well, firstly, make sure you have some. Eventually giving up is better than never giving up. So if you have no timeouts specified for your calls to other services, set a timeout now even if it's much higher than you think you need.

Next, take a look at your response times, using real data from production. Understand how your system behaves over time. If your system load doesn't change much from day to day, you may not need to gather too much data to work out what "normal" looks like. However, if your system can be

more “peaky”—like a retail site having to cope with Black Friday traffic—you may need to gather information over a longer period of time. If the system right now is able to handle the current load you are putting it under, then setting a response time at a point where 100% of your calls finish may be fine. But if you have a number of tail latencies, you might feel happier setting it around the 99th percentile area.

Once that is done, compare your new timeout with end-user expectation. If the user will have given up already by the time your threshold is reached, you could potentially afford to reduce it further. In situations where you have long call chains, consider timeout propagation. Just be aware that this is not a pattern I see implemented that regularly; definitely spend time getting your normal timeout behavior working well first, and with retries as well, before worrying too much about timeout propagation.

Now, I haven’t really spoken much about what you should do when a timeout occurs. Often the sensible thing to do is to just try again. So let’s look at retries next.

---

1 Yes, the last scenario really happened to me.

2 Ting Dai et al., “Understanding Real-World Timeout Problems in Cloud Server Systems”, in *Proceedings of IEEE International Conference on Cloud Engineering* (IEEE, 2018), 1–11.

3 This is actually an example of a *strangler fig pattern*, something I discuss in more detail in my book *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* (O’Reilly).

4 The timeout was specified in seconds, but the default value was set to `Integer.MAX_VALUE`, which in Java ended up being 24 days!

Chapter 4. Retries and Idempotency

'Tis a lesson you should heed:
Try, try, try again.
If at first you don't succeed,
Try, try, try again.

—William E. Hickson

When a call fails, trying again makes a lot of sense. When you fail to establish a connection, it's possible your request just happened to be routed to a machine that was overloaded or to a machine that just got shut down for maintenance. Perhaps you hit a gateway at a point where it was congested, or maybe your request was routed to a server that was slow to start up. In many of these cases, simply trying again may well see the subsequent request sent to a machine that can serve the request promptly.

If you do decide to retry, there are still many questions to answer. How many times should you retry? Should you wait between retries? And are there situations in which carrying out a retry can cause problems? We'll explore all these issues in this chapter.

Before I answer those more detailed questions, though, let's start at the beginning and introduce the retry pattern properly.

Pattern: Retry Remote Operation

Reattempt an operation on a remote endpoint when you cannot confirm that the previous attempt succeeded.

If you call a service and ask it to do something and then are told that the operation failed, you *may* want to retry the operation. The word *may* here is important. Sometimes when an operation fails, the nature of the failure

means that a retry doesn't make sense. For example, if the operation was to see the state of a record and that record doesn't exist, then a retry may not be necessary. Later in this chapter, we'll explore how different error types can help you decide whether an operation should be retried.

You may on occasion be unsure whether the operation worked—if you sent the request to perform the operation, for example, but received no response. Again, if it's important that the operation succeed, it's often important to retry.

How It Works

You should know the answer to these three questions when implementing retry logic:

1. Should this operation be retried?
2. How many times should I retry?
3. How long should I wait before each retry?

We'll explore these in detail later in this chapter. In my experience, it's common to have a default set of answers to these three questions that you might use for all remote operations and that can then be customized for specific circumstances. For example, you might define a list of HTTP status codes that can trigger a retry and then decide that you'll retry only three times, and that you'll wait 100 milliseconds between tries.

Your retry logic will normally be implemented in code inside your running service. You could hand-roll your own retry logic—for example, you could start with a simple loop, with a maximum number of retries as a break condition. However, once you start needing to retry operations in lots of different parts of your codebase—and if you want logic about things like exponential backoff and artificial network jitter (topics we'll explore later in this chapter)—writing all this code yourself can begin to look a bit complex. For this reason, as with timeouts, you may want to look to a

library like Resilience4j or Polly (or the equivalent for your programming language of choice) that can handle this retry logic for you.

Note that if you are using a service mesh such as Istio, you might have the option to have the service mesh implement retry logic on your behalf. This has the attraction of keeping your service code simpler, as it no longer needs to handle retries. Instead, the service mesh is configured with default retry behavior. However, in my experience this can lead to a lot of confusion among developers about what is going on. Personally, I think it is vital that developers understand when an operation is crossing a process boundary and going over a network—whatever abstractions or libraries you might be using should make this very clear to them. I understand the desire to create abstractions over complex operations; however, a network boundary is a fundamentally leaky abstraction. Attempting to hide that the network exists or to avoid confronting the fact that the other processes you are trying to talk to might not be there right now seems to be trying to hide from the developer the fact that they are working on a distributed system.

When to Use It

Remote operations that you need to make sure have succeeded will be strong candidates to be retried, and typically I would have every call to a remote service wrapped with some retry functionality. Whether a retry is actually carried when an operation fails will be decided dynamically. For example, only certain error codes might trigger a retry, or you may decide not to trigger a retry if you are already breaching a timeout threshold.

Some operations may be not vital to the running of a system, and thus you may decide *not* to retry so as to reduce the overall load on the system. In such situations, retries can be avoided.

Sometimes you may not get confirmation that an operation has failed but also don’t get confirmation that it succeeded either. The absence of a response to an operation cannot by itself be viewed as a guarantee of the outcome of the operation. It might have worked, or it might not have worked. In such a scenario, you may want to retry the operation just to

make sure that it has succeeded. This can cause problems if the original attempt did succeed and then you trigger it again; we'll explore that possibility when we look at the idempotent operation pattern later in this chapter.

RETRIES AND CLIENT LIBRARIES

If you are using a client library to communicate with another service, it's worth checking to make sure the client library isn't already doing retries for you.

For instance, the libraries used to communicate with message brokers will normally already have logic inside them to handle retry policies for connecting and communicating with the broker itself, and perhaps more besides. For example, the Kafka clients not only handle retries for you but also handle failovers.

In such a situation, it is normally safe to leave this job to the library itself, especially if it's from the vendor of a product you are using.

Should You Always Retry?

When a call from one computer to another errors, a naive approach would be to assume that attempting the call again is the right thing to do. The problem with this approach is that not all errors are equal.

Imagine you've sent a call to a server asking for a piece of data, only for the server to respond that the piece of data doesn't exist. When doing this using HTTP, you'd expect the server to respond with a `404 Not Found` response code in the header. This is an error. But does it make sense to retry in this case? Probably not. A `404` is telling you that the resource you asked for is not present. Do you really expect it to suddenly appear if you try again? Likewise, a server might reject a request because the client didn't have suitable permissions. Again, that's an error case, but not an error that you would expect retrying to fix.

On the other hand, what if your initial attempt to establish a connection to a server fails because a timeout is reached? Or perhaps a server hangs after you establish the initial connection? In these cases, it's possible that the error is transient. Perhaps your initial connection attempt hit a machine that was being restarted, and a subsequent attempt might cause your request to be routed to a different machine (perhaps via a load balancer) that would happily be able to serve the request. Maybe the server responds by telling you that it is currently overloaded and cannot handle any more requests at this time. In this case, a retry may well be in order only after a suitable delay.

It therefore becomes important to ensure that a client knows when a retry is or isn't sensible.

> **TIP**
>
> Be very clear about what operations should or shouldn't be retried. There is no point in doing more work (and putting more stress on your system) than is necessary.

A Better Type of Error

Your client should be able to differentiate errors that may be sensible to retry from those that are not. For that to happen, though, your client needs additional information around the nature of an error so it can make better-considered decisions. In my opinion, HTTP does an excellent job of providing access to clear, informative error codes.

An HTTP response contains a status code. The `4xx` and `5xx` status codes are for errors—`4xx` status codes indicate that the client has made a mistake in some way, and `5xx` codes indicate that the server is unable to process the request.

By and large, `4xx` status codes are not things you should retry. As mentioned previously, retrying a `404` doesn't seem to make sense. A `403 Forbidden` response code tells us that the request was unauthorized; unless

the logic around how the request is authorized is changed between tries, you could likely expect exactly the same response when a retry is sent.

An interesting exception is a `408 Request Timeout`, where you are being told that your *client* didn't respond to the server quickly enough. In such a situation, a retry from the client might be appropriate; however, if things are going so badly for a client that it wasn't responding quickly enough to a server, it's likely that it isn't in a position to do a retry anyway. Another status code of interest is the `429 Too Many Requests` header; I'll explore that more when we look at backpressure in Chapter 6.

On the `5xx` side of things, you'll see more status codes that indicate a retry is acceptable. For example, a `503 Service Unavailable` indicates the server is temporarily unable to handle the request. In such a situation, a retry seems reasonable.

Knowing which status codes you need to support in the client will depend on knowledge of what HTTP status codes (or equivalent codes from the protocol you are using) you can expect to be sent. For HTTP, the number of `4xx` and `5xx` status codes is large (there are nearly 40 in total), and not all of them may be appropriate in your use case.

All of this assumes you are using HTTP directly. If you are using HTTP underneath a higher-level abstraction, the HTTP status codes may not do what you expect. For example, with gRPC, if a server needs to send an error, the HTTP status code is set to `200` (which translates as "OK"). The rationale for this is that it's the endpoint, not the HTTP connection itself, that has errored. Personally, I think this is a questionable approach, as a lot of HTTP intermediaries are kept in the dark.[1] To be fair, gRPC does have its own list of status codes, and while it is not as rich as HTTP's, many of the commonly used error cases are addressed, allowing your clients to be more targeted in how these cases are handled.

So whatever protocol you use, do make sure you are providing clients with enough information that they can programmatically do the right thing. And if for some reason you've decided to implement your own error protocol,

please at least take a look at what HTTP does here, because there is some excellent prior art on which to base your own work.

> **TIP**
>
> Having clear error codes on a server API makes client-side error handling much easier.

How Many Retries Are Appropriate?

> *If at first you don't succeed, try, try again. Then quit. There's no point in being a damn fool about it.*
>
> —Derived from a quote by Stephen Leacock

Assuming you've decided that a retry is acceptable, how many retries should a client perform? This is essentially a topic we've already explored in a few ways in Chapter 3, but it's worth revisiting here. You'll want to retry as often as you can to carry out the operation, within an acceptable timeout, and without putting too much load on the system. If you keep retrying, you are not only generating more load on the system (often at a time when the system might least be able to handle it), but you also may be taking too long to respond to a real user.

Retrying too often can actually bring systems down. In 2017 Square suffered a large-scale outage that was caused in large part by one of the key systems hammering its own database. Multipass, which handled multifactor authentication, had retry logic in place when performing transactions against Redis. Unfortunately, the retry logic had a hardcoded limit of five hundred retries that would be attempted when calls to Redis failed, along with no delay between the retries. This caused a nasty spiral—if a call failed (perhaps due to Redis being overloaded at that particular time), clients would rapidly keep retrying the calls with a very high upper bound, ensuring that Redis remained overloaded; thus Redis was not given a chance to recover.

So a limit on the number of retries makes sense, but you want to make sure the number of retries being allowed won't cause the server to be overloaded. You'll also want a bit of a delay between the retries, something we'll get to shortly.

Fixed Retry Count

One of the easiest ways to cap the number of retries that will be carried out is simply to have a fixed number of retry attempts, and typically the number of retries is small. For example, you might set a retry limit of three, meaning that after an initial request fails, the client can attempt to make three more attempts to get a successful response (for four attempts in total).

This approach has the major benefit of being very simple to implement. If you are hand-rolling the code, this is likely just going to be a loop around the calls being sent, with an appropriate break condition. Retry limits are also widely supported in libraries like Polly and Resilience4j.

Whether you decide to hand-roll your retry limit or use an existing library that does this for you, I'd strongly recommend that you allow for the retry limit to be changed without needing the code to be rebuilt—for example, by having the retry limit in a configuration file. This gives you some ability to tweak retries in a production system quickly, without requiring a rebuild and redeploy of the code in question.

A simple retry limit like this can be highly effective in dealing with "dangerous" work in a system. Back in 2005 I helped work on a system for BankCo[2] that performed calculations on a pool of over a hundred machines. Requests to perform calculations were placed into a queue, and when a machine from the pool became available, it would pick up the job and perform the calculation. Unfortunately, due to a bug in the program running the calculations, a specific request that appeared to be totally valid would in fact cause the machine to crash. On crashing, the work request went back into the queue, the next available machine picked it up, and the cycle was repeated. In short order, every single machine in the pool crashed as a result of the request.

In our situation, we needed to add some metadata to the work item in the queue, specifying how many times that piece of work had been attempted. After three attempts, the work item in the queue was assumed to be "dangerous," and it was placed into a separate queue, where it could be inspected by an operator. This kept the dangerous work item out of the system and kept the rest of the system working. We'll explore this concept more in Chapter 9 when we look at the dead letter channel pattern.

> **WARNING**
>
> Sometimes trying to process every request you are sent is the wrong thing to do!

One of the downsides of specifying a total number of retries is that you need to make sure the total sum of the call times taken for each call is less than the total amount of time the client is happy to wait for the server's response. Let's revisit our example from Chapter 3 in Figure 4-1, where the `Enrollment` service sends a request to `Subscription` with a 400 ms timeout. To serve that request, `Subscription` in turn calls `Media Library`. If each call from `Subscription` to `Media Library` has a timeout of 200 milliseconds, and you set 2 as the maximum number of retries, then we have an issue.

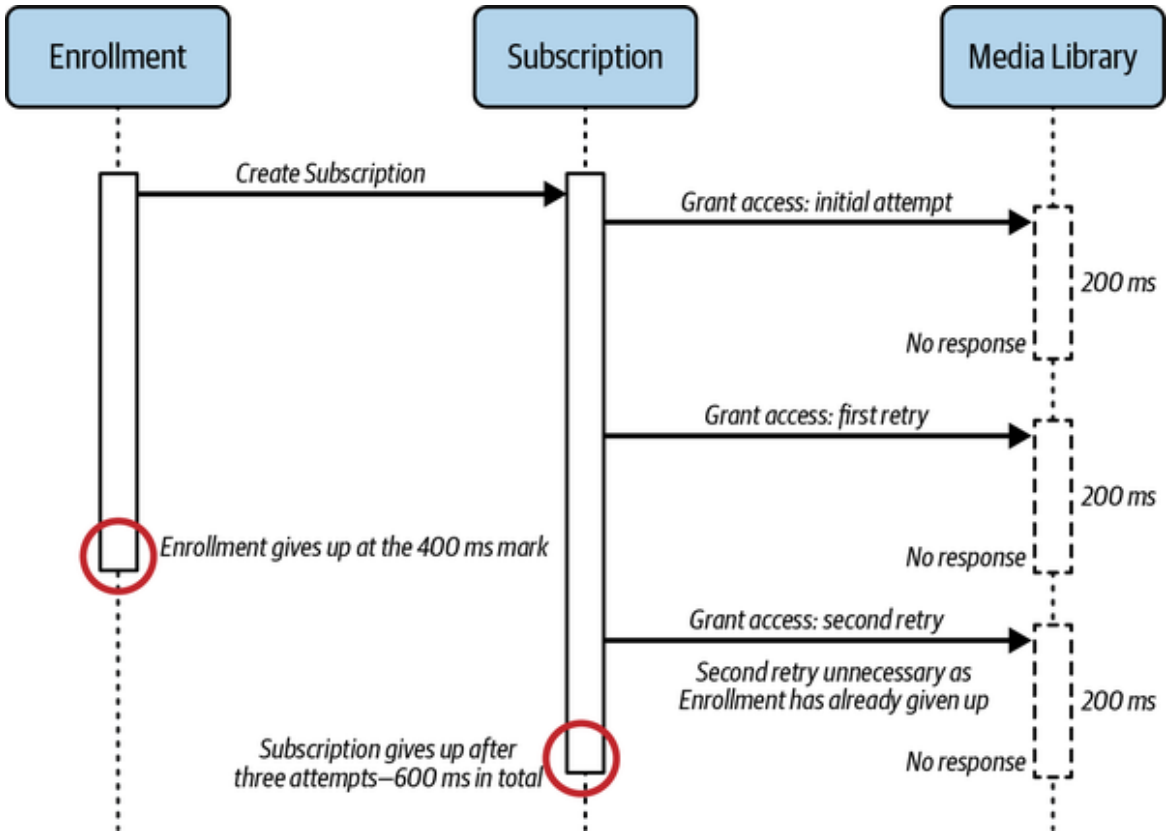


Figure 4-1. `Subscription` can end up exceeding the target timeout from `Enrollment` by making too many retries

`Subscription` could now wait up to 600 milliseconds for `Media Library`—200 milliseconds for the initial call that fails, plus 400 milliseconds for the two subsequent retries. However, `Enrollment` will wait only 400 milliseconds for `Subscription`. So after `Subscription`'s first call and initial retry, we've already hit `Enrollment`'s 400 ms limit. Despite that, `Subscription` decides to send another retry.

This means that even after `Enrollment` might have given up on `Subscription`, `Subscription` might still be doing work. Even if `Subscription` does finally get a response from `Media Library`, the work is wasted, as `Enrollment` has moved on. We've tied up computing resources for nothing.

An obvious answer here is to make sure that the maximum retry limit you set is still in line with any overarching timeout that might be in operation. So in this case, if you know that `Enrollment` will wait only 400

milliseconds, you have to either reduce the individual timeout for calls from `Subscription` to `Media Library` or else reduce the overall number of timeouts you might carry out.

This gets more complicated when you consider that a delay may be needed between retries, a topic I'll explore in more detail later in this chapter. Another wrinkle is that the overarching timeout you're dealing with might vary. `Enrollment` might be happy to wait up to 400 milliseconds for `Subscription`, but a different service calling `Subscription` might have different expectations.

So, although this method of limiting timeouts is easy to implement, it's not without its downsides. Could you solve these problems by using an alternative approach?

Dynamic Retry Limit

Coming back to our scenario in Figure 4-1, the issue you face with a simple maximum number of retries being applied is that `Subscription` might end up making calls even after `Enrollment` has given up. Ideally, `Subscription` would be aware of how long `Enrollment` was willing to wait and would use this knowledge to drive the number of retries attempted.

As I showed in Chapter 3, you can make use of the timeout propagation pattern to have a client send information to a server regarding how long the client is going to wait. You can then use this propagated timeout to dynamically limit how many retries are attempted.

Let's revisit our example and give `Enrollment` the ability to tell `Subscription` how long it will wait. In Figure 4-2, you can see that `Enrollment` is telling `Subscription` that it needs to complete within 400 milliseconds. `Subscription` can now stop attempting to call `Media Library` when this limit is reached. Aside from avoiding wasted work, this also means that you can adapt dynamically to the different needs of clients.

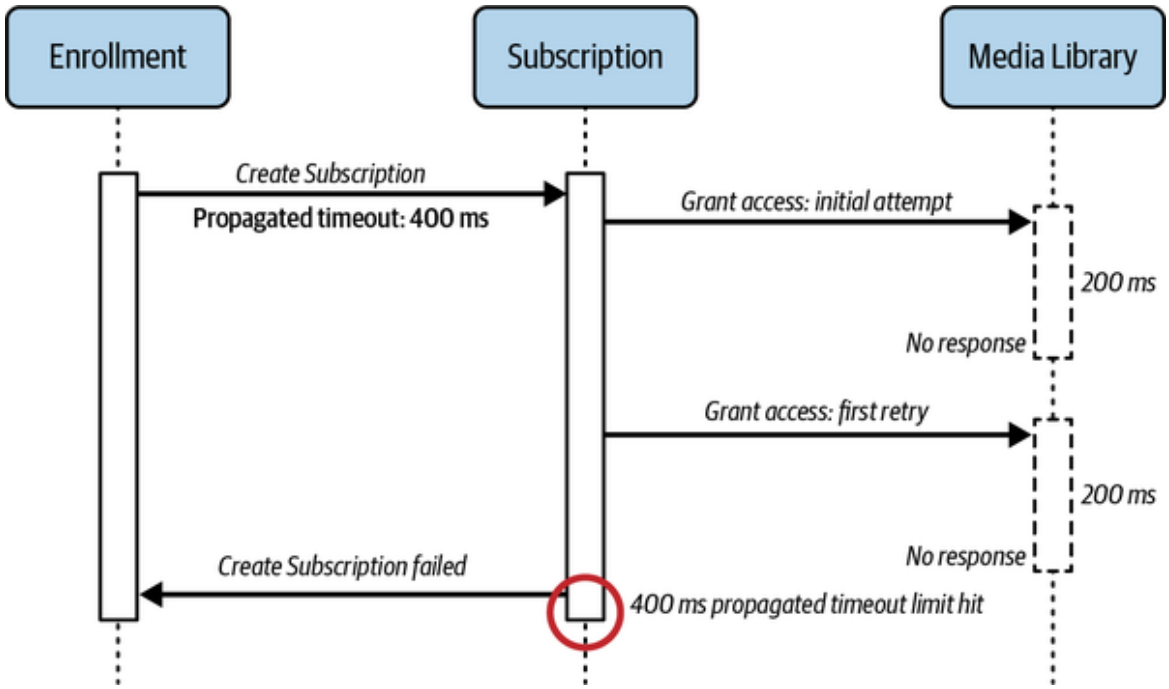


Figure 4-2. Using timeout propagation to limit retry attempts

It's important to note that each call from `Subscription` to `Media Library` has its own timeout. So in effect you have two timeouts to consider. The first timeout is the one propagated from `Enrollment` (the "parent" timeout), and the second timeout is the time you will wait for `Media Library` to respond.

Combining timeout propagation with retry limits can end up allowing you to make an arbitrary number of calls to supporting services. If the initial call and subsequent retry from `Subscription` to `Media Library` both error after 50 milliseconds, you still have 300 milliseconds left on your overarching timeout, so you could end up making several more attempts to get a successful response from `Media Library`.

This can be both good and bad. The good side of this is that the number of retries being sent will adapt dynamically, hopefully maximizing your chances of getting a successful response. However, you could end up hammering an unhealthy server in a situation where your calls are failing quickly. If each call to `Media Library` fails after 50 milliseconds, you could make eight attempts within 400 milliseconds! Does that make sense? At a certain point you might want to take the hint that `Media Library` is

likely having a bad day (or at the very least something is wrong with your network path to `Media Library`).

There are a few ways to deal with this concern over making too many retries when using an overarching timeout. The first is to combine a dynamic retry policy with a hard upper limit of retries that will be attempted—effectively combining both approaches. You could also make use of either a circuit breaker or a backpressure mechanism, two topics I'll look at in detail in Chapter 6.

As I discussed when I introduced timeout propagation previously, the added complexity of implementing the timeout propagation means this is not, in my experience, a common approach. Fundamentally, the communication protocol needs to allow for the timeout to be propagated, and that takes work, especially if you are retrofitting it into an existing system. So that is a challenge you will need to deal with if you want to be more dynamic in setting your retry limit.

Delays Between Retries

When performing a retry, it is sensible to leave a time gap—referred to as a *backoff*—between the attempts. If you retried immediately, there is a good chance that the situation that caused the original failure could still exist. If anything, an immediate retry could make the situation worse. Imagine if the server you sent the original request to timed out because it was unable to handle the load. Sending an almost immediate retry will just increase that load further.

You could decide to have a fixed duration between attempts, perhaps specifying a 100 ms backoff before you send a retry. In practice, though, it's much more common to use an exponential backoff.

Exponential Backoff

With an *exponential backoff*, the backoff duration increases after each retry, typically doubling. So if you wait 100 milliseconds before retrying your

first request, the second retry would follow 200 milliseconds later, with the third retry happening 400 milliseconds after that. But why are exponential backoffs preferred over simply waiting a fixed amount between attempts?

One of the main benefits of an exponential backoff is that it works as a type of rate limiting. When a server is overloaded, it's sensible to find ways to reduce the load in order to try and maintain system stability. Having clients wait longer and longer between retries will reduce the number of calls being made by clients in a given time period, effectively reducing the rate of calls. In Chapter 6, I'll look at some more sophisticated rate-limiting techniques, but some of the ideas I'll discuss there require information to be sent from client to server. With exponential backoff, you create some rate limiting in a way that is simple and easy to implement.

As is the nature of any exponential sequence, the durations between retries can end up getting big. This makes it even more important that you have some way of limiting the number of retries, or else you may find that the clients are hanging around for a very long time indeed. Because of how quickly the delay between retries can be when using exponential backoffs, it's common to change the multiplier; so, for example, rather than doubling the backoff after each failure, you might instead multiply the delay by 1.5.

Jitter

Even with exponential backoff, problems can still occur. If multiple clients are sending requests to a server and are retrying using the same deterministic exponential backoff, it's possible that the client retries can all line up and arrive very close together. You can see an exaggerated example of that in Figure 4-3.

As the load on the server increases, calls start to time out. These calls then get retried later on, even as new requests are coming in. When the retries arrive, they cause even more load, which could in turn increase the number of retries that could be expected.

Here you can see clear clusters where the server would be getting a lot of requests at once, followed by long gaps in which the server would have less

work to do. This is far from ideal, as you can switch back and forth between the server being potentially overwhelmed with calls and the server being underutilized.

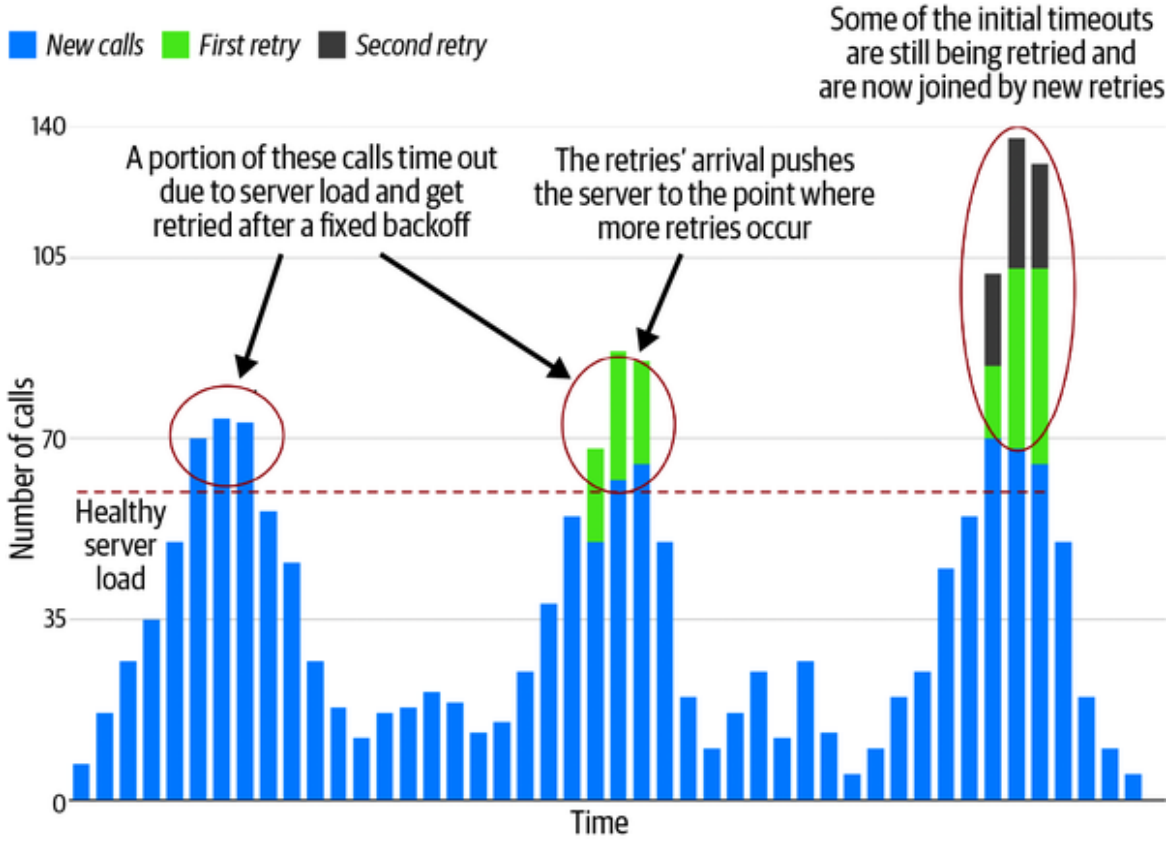


Figure 4-3. With a fixed backoff between retries, multiple retries can arrive at once, swamping a server

The other issue with lots of calls arriving at the same time is that it can make collisions more likely. Imagine two people talking to each other on a telephone call. If both people spoke at the same time, they wouldn't be able to hear each other. If both parties then waited the same amount of time before speaking again, the same problem would occur. This sounds like a pretty terrible telephone conversation, but these types of collisions happen in systems all the time.

Consider a situation in which multiple clients want to edit the same piece of data. Typically, if using a transactional datastore, only one client at a time would be allowed to edit the piece of data. While an edit is taking place, requests from other clients to edit the same piece of data would be rejected.

If the client edits that got rejected all waited for the same duration before retrying, you'd repeat the collision.

To resolve this, you can add some randomness—known as *jitter*—to the backoff that each client is using. The goal of using jitter is to hopefully space out the clients' retries, reducing the peaks while also reducing the idle periods for the server. You can see the smoothing effect this can have in Figure 4-4.

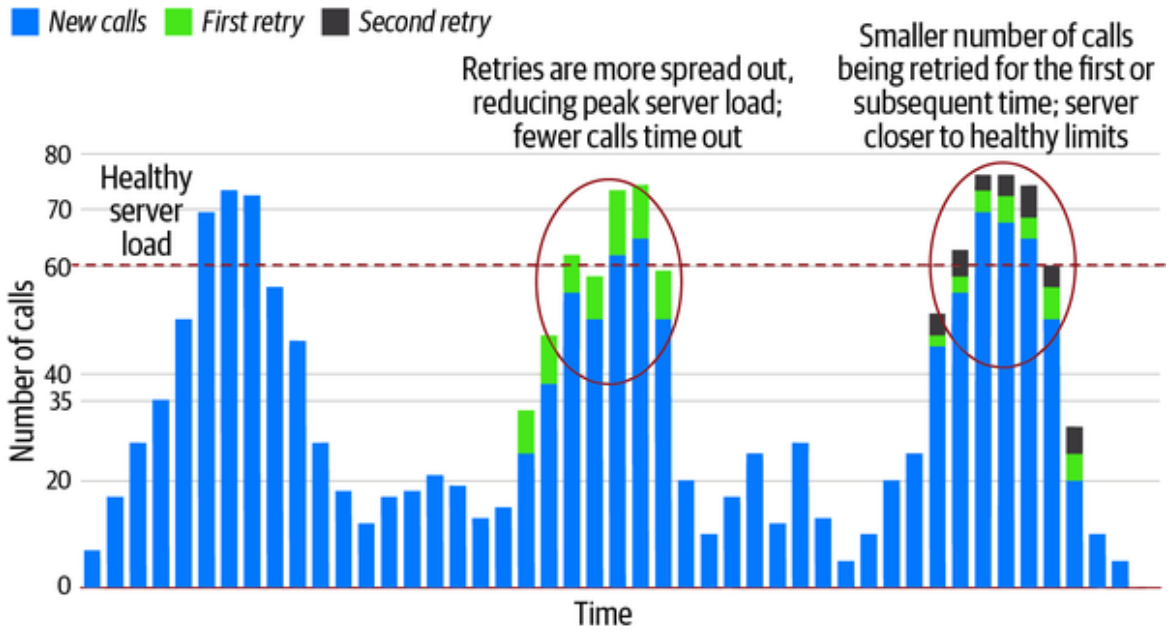


Figure 4-4. Applying jitter to retry backoffs can space out retries, evening out server load

With jitter, the retries hopefully arrive more spaced out. This aims to keep the server operating within normal bounds, therefore improving the chances that the retries will be successful. In the example here, the addition of jitter results in far fewer calls being retried for the second time and also reduces the number of timeouts in general.

Is Retrying Safe?

As we've seen so far, retries can really help in situations where an error appears to be transient. If something doesn't work the first time, try again! (At least until you hit your retry limit, that is.) Sometimes, however, trying something more than once can cause some nasty problems.

If you send a request to a service but don't receive an acknowledgment that the request was received, there are two possible reasons. The first is that the request got through to the service, but your acknowledgment got lost. The second possibility is that your request never actually got through.

This can cause a problem when it comes to retrying. Consider Figure 4-5, where a client is trying to send me £100 (how generous!). The initial request from the client to `Payment` is successfully received and processed, so I have my £100. Unfortunately, for whatever reason, the response is not received by the client. The client *really* wants to make sure that the payment gets processed (clearly a big fan of my work), so it decides to retry the payment. As a result, I end up receiving £200 rather than the expected £100.

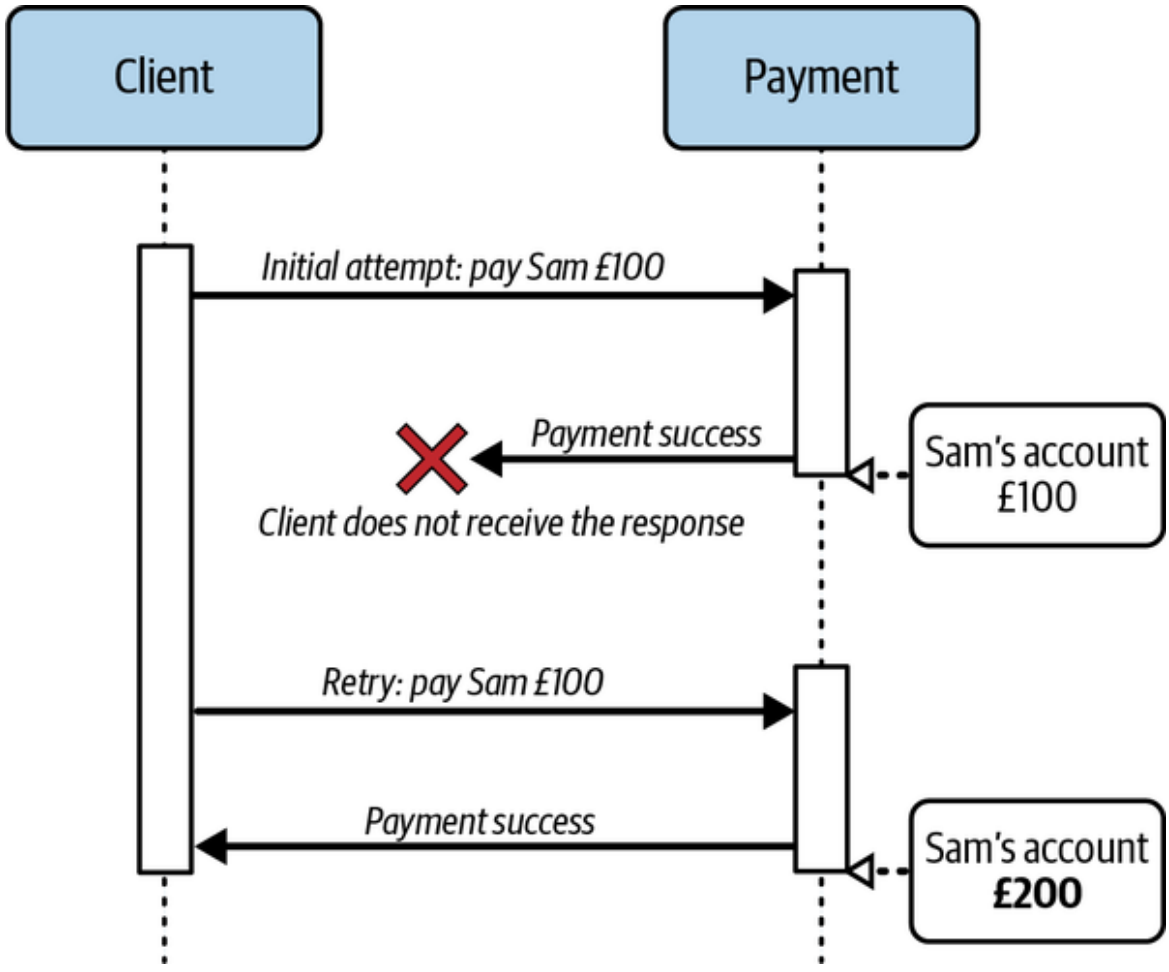


Figure 4-5. Retrying when you don't get a response can cause unforeseen situations

Now, paying twice is clearly undesirable behavior. In this situation, although the client wants to retry because that gives it a better chance of ensuring that the operation completes, the act of retrying isn't safe, as you cannot be certain about the outcome.

When faced with an operation that cannot be safely retried, you deny yourself one of the most useful mechanisms for dealing with a common class of problems. So what's the answer? Make retrying safe. This is where idempotent operations come in.

Pattern: Idempotent Operation

An operation that can be applied multiple times without changing the result.

To make retrying safe, you want certain operations to give the same result no matter how often they are called. In Figure 4-5, the clear intent is that I was supposed to be paid £100. The retries were required to ensure that this happened, but I didn't want them to potentially result in multiple £100 payments being made to me.[3]

What we want to do in this case is to make the payment operation *idempotent*. An *idempotent operation* is one that will give us the same result whether we execute it once or multiple times. If the initial payment attempt works but we keep retrying the operation, no additional payment is made.

Before I look at idempotency in our software, it may be worth considering idempotent operations that you might have experienced yourself. I suspect many of you have used a lift (aka an elevator in the US).[4] Aside from some very fancy exceptions, there usually are separate buttons that allow you to call a lift either to go up or to go down. When you press the "up" button, the lift is told you want to go up, and it will (hopefully) arrive for this purpose. Once the "up" button has been pressed, subsequent presses of that button don't change the outcome—the lift has been called and is on its way. So you can say that lift buttons are idempotent.

As an example of a real-world operation that is nonidempotent, most light switches act as a toggle. If the lights are currently off and I flick the light

switch, the lights come on. If I flick the light switch again, the lights go off. A light switch is therefore not idempotent (which should be obvious from its being a toggle rather than a button).

How It Works

To make an operation idempotent, we have to have some way of knowing whether the operation being attempted has already been processed. If it has, you can simply return the original result without having to do any work. If the operation hasn't been attempted, you can go ahead and process it.

There are two main patterns for making operations idempotent. The first, client idempotency keys, is the simpler of the two and involves the client sending through a unique ID for the operation; the server can then keep a record of processed requests using this ID and handle retry attempts accordingly. The second pattern, request fingerprinting, has the server store a local representation of the request (typically as a hash of the request body) and can be retrofitted into existing systems, albeit with a number of drawbacks. We'll explore both patterns in more detail shortly. Both solutions involve the server storing a record of each request that is processed and what the result is.

Aside from the specific implementation pattern you choose, there are a several issues to consider in terms of the interaction between client and server when idempotent operations are in play.

How to respond to repeated attempts

After an operation has been completed, if a server receives a second (or subsequent) request for that operation to be performed, the server should respond with the same semantic response as was sent for the original response. The fact that the client didn't receive the original response doesn't change what happened the first time around. If a client's first attempt to send payment of £100 worked, the server would respond with a confirmation that the payment was made. If the client attempts that same

payment again, the server should again just respond that the payment was made.

In the case of a failed payment, the same applies. The operation was still completed. If the first attempt to send £100 failed and the server sent an appropriate response, then a subsequent request to send the same payment should get the same answer.

Note here that I am stating that the response should be the same semantically. That is, the response should have the same meaning but doesn't have to be identical. For example, you are free to change the metadata in responses for second or subsequent attempts, perhaps to indicate that this request had been received multiple times.

It's also important to note that when handling a repeated attempt to perform an already applied operation, it's not the case that the server does nothing. It can still log the attempt, for example, or it might even raise an alert if it is seeing an unexpectedly large number of retries, which could point to an underlying issue.

What happens when state changes between attempts

An interesting edge case to consider is what happens if the resource being operated on is changed or even deleted between the original request being sent and a subsequent retry. Consider Figure 4-6, where you are upgrading the subscription for a customer. Client A sends an update request, which is accepted and processed by `Subscription`, but the response that's sent back confirming the payment never reaches Client A. Before Client A can retry the operation (remember, we'll likely have some sort of backoff in place), Client B comes in and deletes the same subscription that you just upgraded. Then Client A retries the original upgrade request.

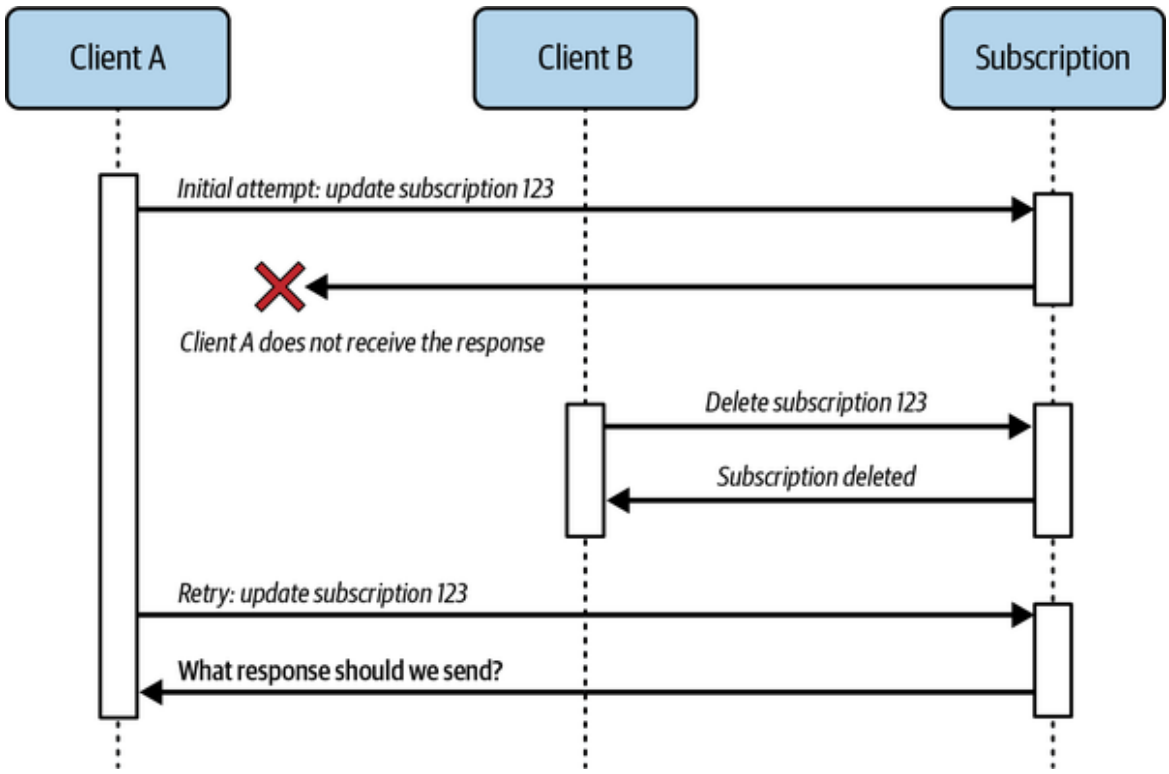


Figure 4-6. What happens when you retry an operation that already succeeded if the thing you changed got deleted in the meantime?

What should the server do in this situation? It might at first seem odd, but we still follow the rule about how we respond. The server should acknowledge the second request and state that it completed successfully. The subscription was updated. The fact that it was subsequently deleted doesn't change the fact that the original request was applied.

Think of it a different way. If Client A had received the original response, it wouldn't have had to retry. It would have received the success response before the deletion by Client B; the fact that the subscription was subsequently deleted is a separate issue.

The thorny issue here is that the original request may not have actually been applied in the first place. In Figure 4-7, the original request never makes it through. Before the retry is sent, the subscription is deleted. In this situation, an error would be appropriate—the request was not applied, and so telling the client that the subscription is no longer there is the appropriate response.

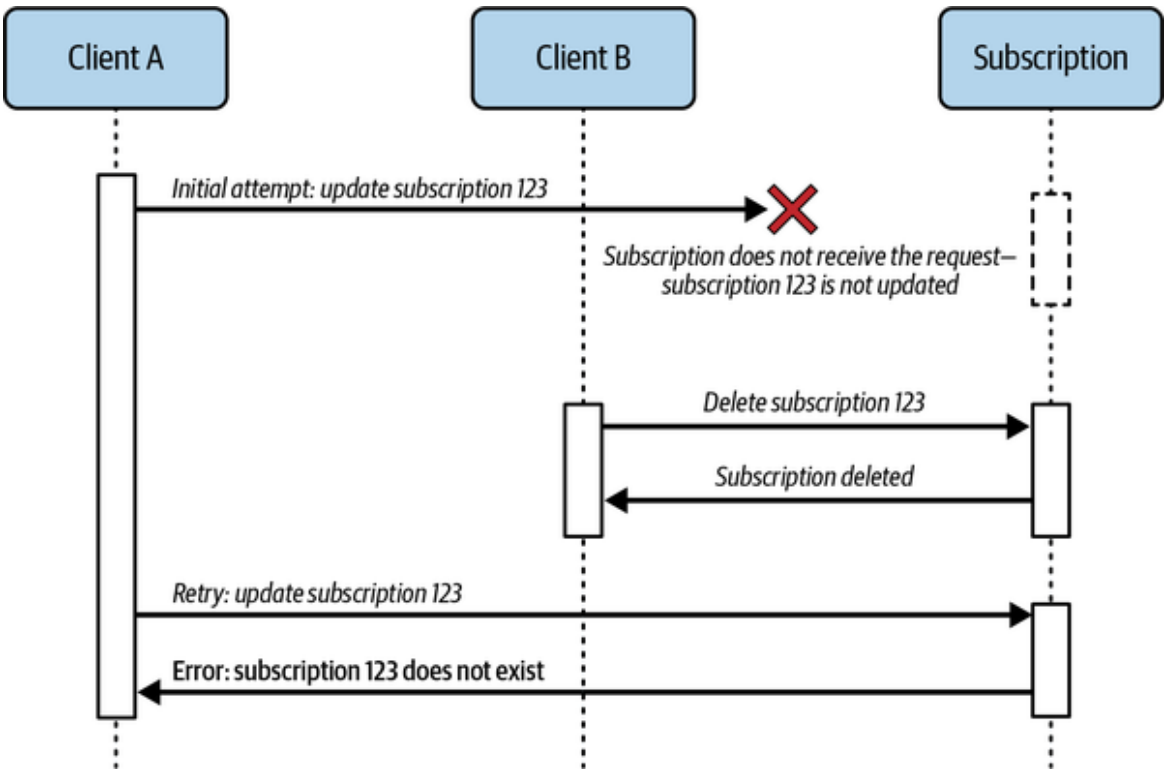


Figure 4-7. If the original upgrade request was not processed before the deletion, an error is the appropriate response

> **RESPONDING**
>
> How to respond to a request that had previously been processed might seem confusing at first. Just remember that if a server is seeing a retried operation that was previously processed, all you need to do is give the same semantic response again.

Handling concurrent duplicate requests

Here's another potential edge case: what happens if two service instances end up trying to process the same request at the same time? The circumstances under which this could happen might be rare, but it is a possibility. Consider an alternative version of our payment scenario, as shown in Figure 4-8. The client has sent a payment request, and it is picked up by service instance A. Service instance A checks whether the request has already been handled, sees that it hasn't, and starts working on the payment. However, service instance A is taking so long that the client gives up and

retries. This second attempt is handled by service instance B, which again checks the log and again finds out that the payment has not been handled.

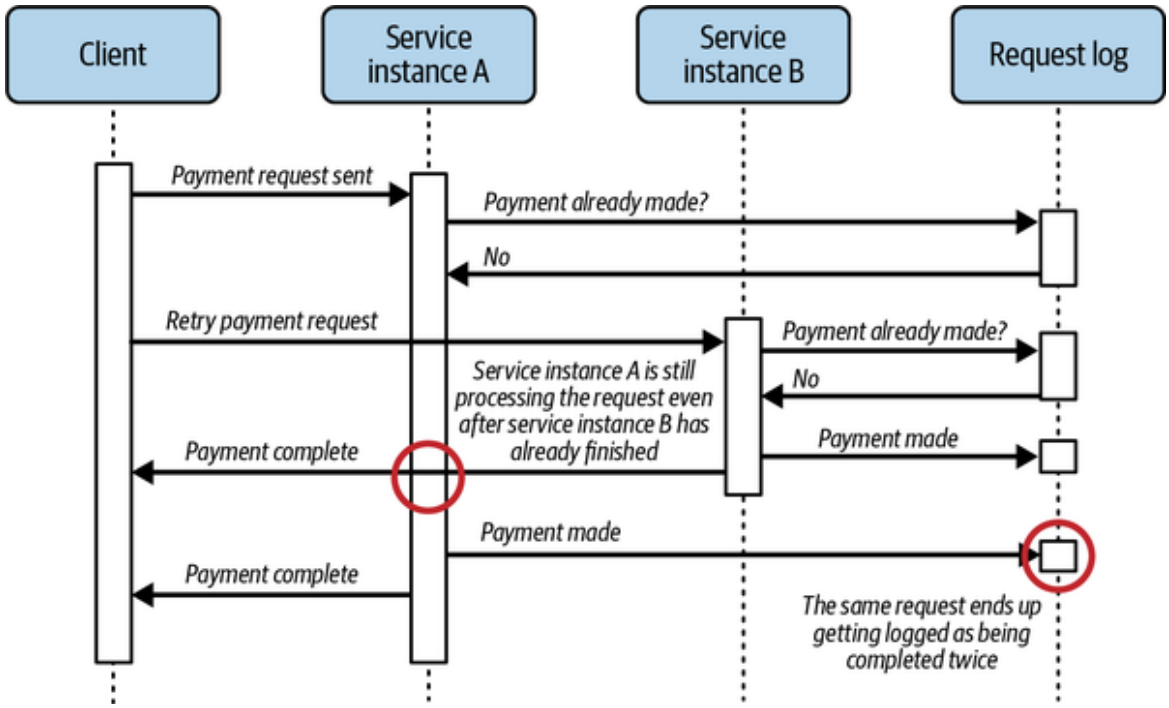


Figure 4-8. The problem with concurrent request handling

The problem we have now is that despite our attempt to make this operation idempotent, we still end up with two payments being made. This problem arises from the fact that we log that the request is handled only after it has been processed.

So what can we do? Well, one option could be for the server to log the request *before* processing. In that way, service instance A would have logged its payment request first, and then service instance B would have seen that this was already logged and would not have started its own processing attempt. This then introduces some more issues. What happens if service instance A crashes after logging the request and the payment isn't properly made? What happens if service instance A starts processing but then just hangs forever?

A more appropriate solution might therefore be to record the request processing in a more sophisticated way. Firstly, a service instance could log that processing was started and was in progress, and that any other service

instance should not also start processing. Along with this, there needs to be a timestamp or something similar to record when processing started. Once processing has completed, the record is updated accordingly. If another attempt is made to process the request, and it is seen that the processing of the request started but had not completed, then you could decide after a period of time that the processing hung and therefore the request should be reprocessed.

Fundamentally, what we are discussing here is the concept of two service instances coordinating work via a shared lock. When processing starts, a lock for that request is acquired. When processing is complete, that lock is released. A detailed discussion of distributed locking protocols is outside the scope of this book, but the subject is covered in depth in *Designing Data-Intensive Applications*, 2nd edition, by Martin Kleppmann and Chris Riccomini.[5] On the implementation side of things, a system such as etcd (which is a core part of Kubernetes) or Consul provides the ability to coordinate around time-based locks and thus could be a workable solution for this problem. You could also, of course, hand-roll your own solution, perhaps with some logic around rows in a database table, but that's not something I'd generally recommend. The complexities of shared locking mechanisms are not for the faint of heart, and corner cases abound. It's much better to rely on tried and tested implementations of proven algorithms rather than risk complicating your system with homegrown solutions that you have to maintain and that still have flaws.

Fundamentally, introducing the need to coordinate locks between service instances introduces more complexity to your solution. You have to decide whether this additional complexity is warranted. How likely is it that the same request might end up being processed by two (or more) instances of your service in parallel? And if this *does* happen, is it more sensible to spot that it occurred and clean things up afterward? The answer is likely going to depend a lot on your context—a key part of this would be understanding the business impact of such a situation occurring.

When to Use It

In general, any situation in which an operation involves a change in state in the remote service is a candidate for being made idempotent. Taking payment twice, placing two orders instead of one, and scheduling two deliveries when only one is needed are all examples of situations that are crying out for operations to be made idempotent.

Operations that are entirely read-based are, by their nature, already idempotent—and if you were worried about the performance impact of repeated retries of a read from a client, then caching can be used to offset this concern. Some non-read operations are also idempotent by nature. For example, consider a delete request. If I send a request to delete something, sending that request multiple times can't change the outcome—the item is still deleted.

Some write operations can be retried without causing any major problems. Consider generating a thumbnail for a larger image. Ideally you would do this once, but if you did it again, it's not the end of the world. Again, the image thumbnail could easily be cached to ensure a repeated operation didn't cause too much impact.

Making Operations Idempotent

So making operations idempotent is the way to go. But how do we do this? Well, there are two patterns I'll explore. First, I'll show how a client-generated unique ID can provide a very simple approach to idempotency. Then I'll show you how server-side fingerprinting can be used in situations where the client-server protocol can't be changed.

Both patterns have their merits, and they can in fact be used together.

Pattern: Client Idempotency Key

An ID generated on the client side to allow a server to identify whether this operation has already been processed.

When using a *client idempotency key*, a client sends along a unique ID for the operation to be performed. Before a server carries out an operation, it can first check the idempotency key to see whether that operation has already been processed. If not, the operation continues as normal. If the operation was previously carried out, then rather than the operation being repeated, the server can simply send back the same response as to the initial request.

This is a nice, simple solution that is easy to implement upfront, though retrofitting it into existing systems can be painful.

How it works

With client idempotency keys, the client needs to generate a unique ID for the request, and the server needs to expect to see that idempotency key and handle it accordingly. This means that the idempotency key needs to be a mandatory parameter in a client request. When using HTTP, the idempotency key would be sent as part of the request header. The convention is for this header to be called `Idempotency-Key`.

For a concrete example of idempotency keys in APIs in the wild, specifically in the area of online payments, Stripe, Adyen, and Worldpay all make use of an `Idempotency-Key` header in their APIs to allow for safe retries. Outside of the world of payments, the AWS APIs use their own ID with the name `ClientRequestIdentifier`.

There is the open question of how to generate the ID of the request. A sensible starting point would be to use a universally unique identifier (UUID), which can reasonably be considered to be unique and is easy to generate. If this request is being done as part of a larger trace of calls across a set of services, you might be tempted to reuse an existing trace ID for those calls; I'd strongly advise against this, though, as it could cause issues if you need to legitimately perform the same operation twice as part of the same trace.

If you are communicating via an alternative protocol rather than HTTP, you'll need to define your own standard for where this key should be found

and what it should be called. If using a protocol that abstracts HTTP itself, you may or may not be able to add your own headers. When looking at the use of idempotency keys in gRPC, for example, there doesn't seem to be a convention around this. You can use gRPC custom headers to create your own header that then appears in HTTP, in which case you could follow the convention around the use of `Idempotency-Key`; however, it isn't clear to me how common this is.

When a server receives a request and checks the idempotency key, it needs to check it against a list of previously processed requests. Initially, you might think to keep this list of previously handled requests in memory. The problem with that is twofold. Firstly, if the server is restarted, the list of processed IDs is lost. Secondly, it's common to have multiple instances of a service, and the list of processed idempotency keys should be shared across all instances if you want to avoid repeating work. An obvious fix is to store the idempotency keys in a shared datastore that all instances of a service can share from. Due consideration then needs to be given to any contention caused by the use of such a shared store, and also to how long it makes sense to keep these IDs for. I've seen some people storing them in datastores that support a time to live (TTL) natively to handle eviction of older keys. Version 7 of the UUID specification includes timestamps as part of the UUID itself, which is useful, as it makes them sortable and can also make eviction of older entries easier.

When to use it

The use of idempotency keys is a very simple solution to making operations idempotent and is always my preferred approach. The main difficulty with using idempotency keys is that the client needs to send them, and the server needs to accept them. This can be specified when designing a service interface in the first place. However, when retrofitting this into an existing API, you may have to start with the idempotency key being an optional parameter unless you want to introduce a breaking change. If you are unable to retrofit in the use of idempotency keys—perhaps in the case where you are dealing with legacy software that is difficult or impossible to

change—then server-side fingerprinting will be the way to go (and we'll look at that next).

Pattern: Server-Side Request Fingerprinting

An approach in which the server stores a representation of a request to spot that retried requests are subsequently received.

In situations where you want to make an operation idempotent but can't enforce the use of a client idempotency key, an alternative approach is for the server to generate a fingerprint based on the request to determine whether that request has been seen before. There is no need for the client to do anything special; the server keeps track of everything. This approach does not come without some significant downsides, however.

How it works

One of the simplest mechanisms for generating a fingerprint would be to create a checksum of the request itself, as you see in Figure 4-9.

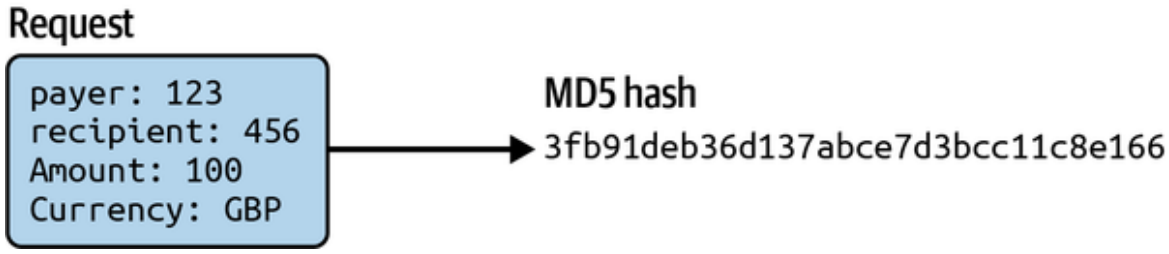


Figure 4-9. A fingerprint generated from request parameters

Here we are generating an MD5 hash from the request—the string that starts `3fb91`. . . . If one of the parameters of the request changed, the generated hash would also change. A subsequent request coming in with the same parameters would generate the same hash, allowing us to detect a duplicate request that we can then ignore. This fingerprint would then need to be stored in the same way that a client idempotency key would be stored. That is, we'd want the fingerprint to be stored in a durable fashion—and if multiple service instances could process requests, this datastore would need to be shared.

Where to use it

One of the benefits of this approach is that the clients aren't required to do anything new. This makes this technique useful in situations in which you are retrofitting idempotency into an existing API and don't want to (or can't) change the client in any way.

There is an issue with this approach, though. It is possible to produce false negatives—that is, you might think a request that comes through is a duplicate that should be ignored, when in fact it shouldn't be. All request fingerprinting can do is determine whether the two requests are the same. It cannot determine whether it was the client's intent to perform the operation only once.

Now, the likelihood of a client wanting to pay the exact same person the same amount twice seems remote. But consider a regular payment. What if you want to send £100 to the same person every week? If the request parameters look the same as the previous payment, you could think that payment has already been processed.

A protection against this would be to specify a time limit. For example, if you see two identical payments within a few seconds, you might assume that the subsequent request in the same time period with the same fingerprint is in fact a duplicate and should be ignored. But if a duplicate request was received much later, you might assume that the payment is intended to be processed a second time. In this way, you consider that the fingerprint has a period of validity; the exact duration for this would have to be determined based on the nature of the operation, and if you are using a datastore for your fingerprints that supports TTLs, it's possible that this task could be delegated to the datastore itself.

Even if you have a defined period of validity, is that certain to fix the problem of false negatives? Let's consider a different type of interaction. You have an API that allows you to spin up a virtual machine. How likely is it that a client would actually want to provision more than one identical VM in a short period of time? Actually, it's a pretty common situation—imagine spinning up a bunch of infrastructure to run a set of services, for example.[6]

For the reasons outlined, I'd always recommend client idempotency keys over server-side request fingerprinting. That said, there may be a reason to use both patterns together.

Using Client Idempotency Keys Together with Server-Side Fingerprinting

What happens if the client changes part of the request when sending a retry? There can be situations in which this is legitimate. For example, if you were propagating a timeout, you could send the remaining timeout budget to the server as part of the request (as I discussed in "Pattern: Timeout Budget Propagation"). After each retry, the remaining time would reduce, and the matching field in the request would change.

When using server-side fingerprinting in such a situation, the server would need to know that part of the request should be ignored when generating the hash. Now obviously, when using a protocol like HTTP, something like a propagated timeout would more sensibly live in the HTTP header rather than in the body. So ideally, metadata fields should be moved to the header (or the equivalent in alternative protocols) to solve issues like this. Of course, often the reason you are using server-side fingerprinting is that you can't make changes to the existing protocol—so you may have no choice other than to accommodate this when creating your fingerprint.

On the other hand, if using client idempotency keys, the fact that a field changed between the initial attempt and a retry isn't an issue at first glance. The retry comes through with the same idempotency key, so we assume it's the same; any field changes are ignored. This could result in nasty bugs occurring, though. If the client is (erroneously) changing fields in the request body for a retry, and the server is sending back confirmation that the request was successfully handled, it might not be clear to the client which version of its request actually got processed. Consider a payment of £100 being sent with the idempotency key of `123`, and then a retry being sent with the same idempotency key, but with the payment amount changed to

£200. If the client gets a response saying the payment was made, which amount was actually paid?

The AWS APIs take the blanket decision that a repeated request with the same idempotency key but different parameters in the body is an error case, and a validation error is thrown back to the client accordingly. I think this is generally a sensible solution—the goal in this situation is to tell the client that it was doing something silly.

With the AWS APIs, the desire to catch scenarios like this is why they use both client-generated idempotency keys *and* server-side fingerprinting. This approach allows them to go beyond documenting what they expect clients to do (don't change request body parameters for retries) and make sure clients actually are doing the right thing. So even if you are able to use client idempotency keys to make your operations idempotent, also making use of server-side fingerprinting can account for inappropriate client behavior quite neatly.

DECLARATIVE PROGRAMMING

Declarative programming is a style of programming that avoids the use of control flow logic (such as `if . . . else`) and has you focus on making statements about what you want to achieve rather than on how you should do it. The details around the mechanics of how this is done are hidden away.

Examples in which you will see declarative programming include infrastructure as code (IaC) toolchains such as Terraform and build systems such as Make and Maven.

A core tenet of declarative programming is that it is, by definition, supposed to be idempotent. When you have an Ansible script that defines how a machine is to be set up and you run that script a second time, nothing happens if the machine is already in the expected state—the machine isn't set up another time.

Thinking in a declarative style can be a way to help you focus on idempotency, even if it's not feasible to create your own declarative abstraction for your otherwise imperative code.

Testing Retries and Idempotency

Idempotency controls can easily be tested as part of a normal, functional automated test. Running tests that mimic client behavior against a server that is making use of idempotency keys or request fingerprinting should give you sufficient confidence in their implementation.

Testing retries is a bit trickier, and techniques similar to those outlined in Chapter 3 are likely appropriate. So inserting artificial jitter into the network to cause a timeout and trigger retries would be well worth consideration.

Further Reading

For a deeper dive into the benefits of exponential backoff and jitter, Marc Brooker has a great explanation using simulated traffic patterns over at the AWS Architecture Blog. For something more interactive, Sam Rose has an exceptional visual simulation of the impact of different retry mechanisms that runs entirely in the browser.

Summary

So: when at first you don't succeed, try, try again! Unless you shouldn't retry in the first place. And if you do decide to try again, you may want to limit how often you do it. . . . I get that this can be confusing, but let's try to bring it all together.

Firstly, when you try something and it doesn't work, try to find out from the nature of the failure whether a retry is appropriate. If it looks like a transient error, a retry may be in order. If it's the equivalent of an HTTP `404 Not Found` status code, then perhaps a retry doesn't make sense.

If you decide to retry, make sure you have an upper limit, and set it conservatively to ensure you aren't bombarding the server. Just make sure you consider the complexity of what happens with retries in call chains, as this might make you consider propagating timeouts.

Between retries, have a simple exponential backoff policy with some jitter. Client libraries like Resilience4j or Polly can do this for you.

Finally, retrying calls is often the best thing for a client to do to make sure something happens. But for that to work, it's essential the server implements the operation in an idempotent manner. Otherwise, you might end up with some unexpected behavior!

In both this chapter and Chapter 3, I discussed the need for clients to be wary of overloading servers. In Chapter 5, we'll turn our attention to what happens when we fail to stop a thundering herd and our system is in danger of being trampled underfoot.

[1] Also, at this point I'd like to send out a plea for you damn kids to keep off my lawn.

[2] Name changed to protect the innocent!

[3] Although as a struggling technical author, I'd be pretty happy with the situation. I do have a mortgage to pay!

[4] We could start discussing which of these terms is "correct," but the term *lift* is preferred in the international standard ISO 8100-20:2018, which governs "Lifts for the transport of persons and goods." If it's good enough for the International Organization for Standardization, it's good enough for me.

[5] Martin Kleppmann and Chris Riccomini, *Designing Data-Intensive Applications*, 2nd edition (O'Reilly, 2026).

[6] This is a real-world example and a situation that public cloud vendors need to deal with.

Chapter 5. Thundering Herds

You glimpse it first on the horizon. Is that smoke? No, it's something worse—the dust kicked up by a thundering herd, and it's heading your way. Any moment now, the herd will arrive, sweeping all before it, and leaving everything in its wake broken and destroyed.

Now obviously, as a metaphor for system problems, this might be a bit over the top, but I think you get the idea. When it comes to keeping systems up and running, few things worry us more than the prospect of some thundering herd arriving unexpectedly and taking the system down. Be it a sudden interest in a new product, a denial of service (DoS) attack launched by some nefarious ne'er-do-well, or even a sudden spike in load that your own system accidentally inflicted upon itself, the thundering herd is a potentially catastrophic event that may seem outside of your control.

In this chapter, I'll take you through several examples of the thundering herd. Some I've taken directly from real stories, while others are slightly more contrived examples of challenges that are commonly faced with distributed systems. As I take you through these examples, I'll also share with you some concrete tips on how the damage from these herds can be mitigated, and perhaps even how the herds themselves can be avoided altogether. Before we get there, though, I want to go into a little bit more detail about what thundering herds are and why they are so problematic.

What Is a Thundering Herd?

In computing, when a lot of work arrives that is all trying to access a constrained resource, it is commonly referred to as a *thundering herd*. The causes are manifold, as are the potential ways to deal with it. Fundamentally, though, some part of your system is being asked to do way too much work, which in turn risks that component failing, with some nasty knock-on effects.

One of the challenges with thundering herds is that they are nearly always unannounced and can often appear suddenly. This means that machines and networks that were previously working well within acceptable tolerances become saturated at short notice. Available network bandwidth vanishes, and CPUs are suddenly maxed out without any ability to take on more work. Memory gets filled up and starts getting swapped to disk, tanking performance. If these things started happening gradually, over time, you would hopefully pick up on the general trends and adapt. But when everything very quickly goes from being OK to being bad, you're in a situation in which you're having to fix a problem without the benefit of time.

Some herds come from outside your system, while others are generated internally. External sources include things such as DoS attacks or significant (and perhaps unexpected) increases in user traffic. DoS attacks are specifically intended to make your system unavailable and as such can be especially problematic. The intent and often the capabilities of such an attack are such that DoS attacks are worthy of special focus, and later in this chapter we'll explore them in more depth. But many thundering herds are generated internally and in the aftermath can feel like they were self-inflicted. They often occur in the wake of a failure elsewhere in the system, leading to unexpected consequences. Let's start by looking at some examples of internal thundering herds and at what we can do to mitigate them.

DEFINITIONS AND METAPHORS

In the context of computing, the term *thundering herd* has often been used in fairly narrow circumstances, typically to describe what happens when multiple threads or processes on a single computer are competing for the same resource. Fundamentally, though, it's a metaphor—it's drawing a parallel to a herd of animals charging forward in an unstoppable stampede. Personally, I think it's odd to limit the use of a metaphor to a very narrow set of circumstances, so I am using the term more broadly.

Load Balancer Domino

As we'll explore in more detail in Chapter 8, we typically think of load balancers as something that can help make our systems more robust. A load balancer distributes requests across multiple servers. If one of those servers starts misbehaving, requests will be distributed across the remaining servers, hopefully ensuring that a problem with one server won't be too impactful for the system as a whole.

However, things can still go wrong. In the face of a thundering herd, a load balancer can only do so much. And a service that was only just barely holding up to the amount of work being sent its way may well find itself in trouble if one of its instances goes down.

Example: Pet Service

Let's look at an example of this type of failure mode in the context of the `Pet` service. In Figure 5-1, we see four instances of the `Pet` service, each of which is currently handling 120 requests. Requests sent to these instances are routed via a load balancer, which attempts to ensure that the work is evenly distributed across the instances.

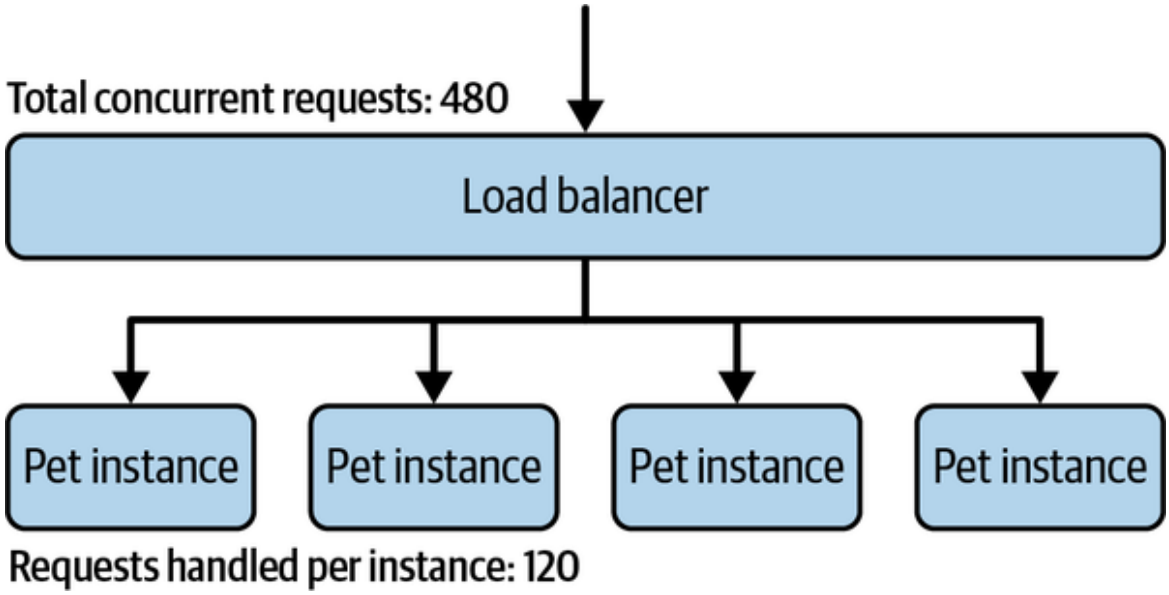


Figure 5-1. Four instances of the `Pet` service being load balanced

These instances are all very close to their limit—their computing resources are close to being saturated. If nothing goes wrong, the service can continue to handle this load. However, if one of these instances fails, as shown in Figure 5-2, then the requests that instance is handling would initially all be dropped, with the clients having to decide whether they want to retry them. If we assume that work is still arriving at a consistent rate, we'd expect the 120 requests that the failed node was handling to end up being redistributed across the remaining instances. We could then expect the remaining three instances to end up having to handle 160 concurrent requests each—and if our instances were already at their limit handling just 120 concurrent requests, that could be a major problem.

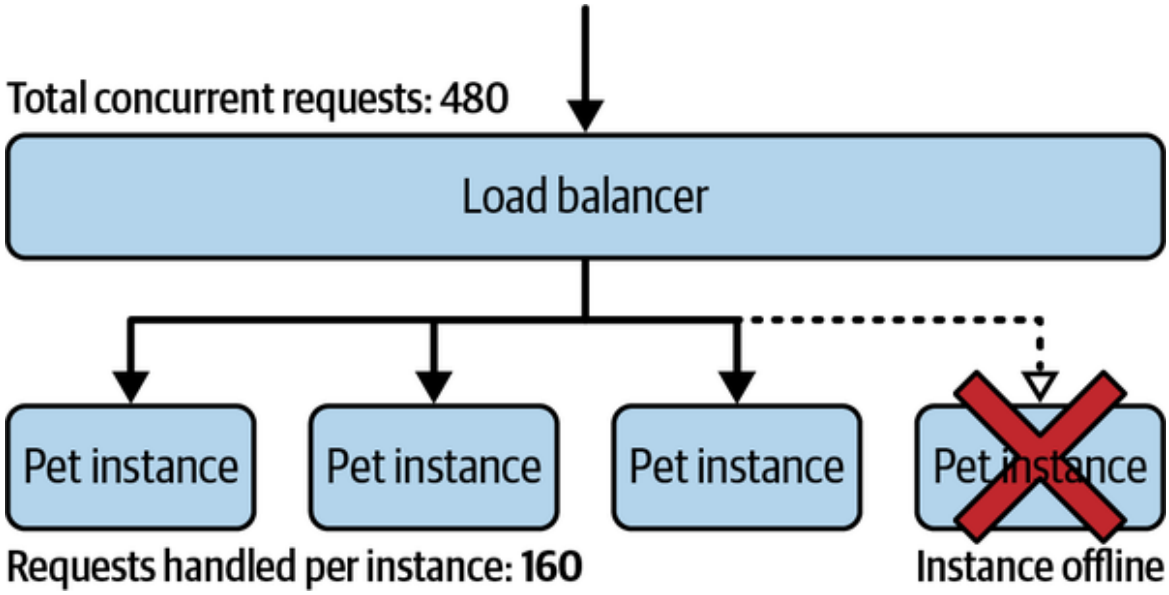


Figure 5-2. An instance failure means incoming work is distributed across the remaining three instances

Each of the remaining instances is in effect dealing with its own thundering herd—a sudden, unexpected spike in the amount of work. If you are unlucky, then this additional load pushes the remaining instances over their limit, potentially causing them to fail as well, creating a nasty domino effect: another instance fails, so more work is distributed across the remaining instances, which in turn are way over their capacity, and so they too fail.

Mitigations

The nature of this failure is particularly unpleasant. A single failure causes a subsequent cascade of failures, resulting in the service being taken offline in a fairly short space of time. Luckily, this type of issue has several potential mitigations.

Overprovision

The easiest way to avoid the `Pet` service's cascading failure is to have excess capacity on hand to ensure that even if a single instance of the `Pet` service fails, the remaining instances can handle the expected number of requests.

In the original scenario, the `Pet` service had four instances, each of which was handling 120 concurrent requests. If we increase the number of instances to five, as in Figure 5-3, each instance of our service is now doing less work (handling 96 concurrent requests, down from 120) and therefore has more headroom to handle more work in the future.

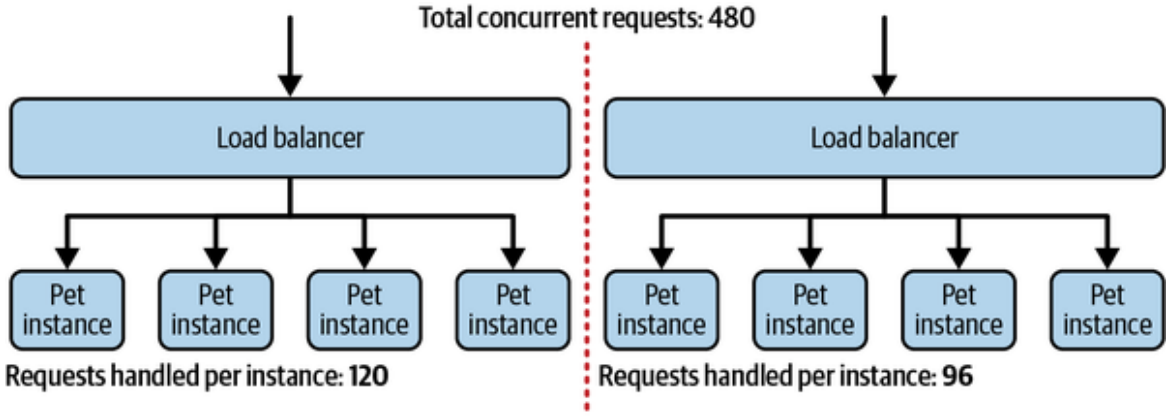


Figure 5-3. Adding an additional instance reduces the number of requests each instance handles

If a single node fails, we go from five instances to four, and each of the remaining instances are now expected to handle 120 concurrent requests, which is still within the capabilities of each instance. The excess capacity we had available is able to absorb the additional work, avoiding the cascading failure.

Of course, the question comes down to how much to overprovision. If we're trying to handle the failure of a single instance, having a single extra instance to cover that failure would be sufficient in our scenario. But what if we were worried about multiple instances failing at once? This could be a concern if you had a number of instances running in a data center (or public cloud availability zone) and wanted to hedge against the risk of all instances in that data center failing. The more you overprovision, though, the more it costs—and if you are trying to hedge against multiple instances failing at once, then the costs of having to run multiple extra copies of your service could add up quickly.

Concerns around cost aside, overprovisioning should probably be your initial go-to to hedge against this type of failure. That said, some of the

other mitigations we'll look at next could easily be used in conjunction with some degree of overprovisioning.

> **A NOTE ON LOAD BALANCERS**
>
> In the `Pet` service example, I'm assuming that the load balancer is able to perfectly balance work across the instances. In reality, while load balancers will do their best to achieve this, it's not always possible to get a perfect distribution of work.

Reduce work

Once the `Pet` instance fails, the remaining instances become crushed under the thundering herd that the now defunct instance is no longer handling. What if you could just make that extra work go away?

A viable approach to keeping systems stable is to limit the number of calls that a service will accept, a concept called *rate limiting*. In a nutshell, you specify a limit above which additional work would be rejected.

The idea is that it is better to process some work and remain stable rather than running the risk of the system failing from trying to handle all the work. If each instance of the `Pet` service was able to reject traffic over and above its limit, then the service as a whole would keep running, albeit at a reduced capacity.

There are different ways to implement rate limiting, and we'll be exploring that topic in detail in Chapter 6.

Use desired state management

Many tools exist to allow you to specify how you want your infrastructure to be configured. With a tool such as Terraform, Pulumi, or Crossplane, you specify how you want your infrastructure to look and then the tool sorts out the details. Kubernetes and public cloud autoscaling groups for virtual machines allow you to do similar things, albeit for a more narrow set of circumstances. If you'd like to explore the space of infrastructure

automation in more detail, I recommend *Infrastructure as Code*, 3rd edition, by Kief Morris.[1]

Some of these tools can go further, so if at some point after defining your desired state something changes in the actual system, such tools can detect this and carry out actions to bring the system back into line with your desired state. This is what is known as *desired state management*—the tool is continually ensuring that your system is matching your desired state as closely as possible.

We'll explore the desired state management pattern more in Chapter 8, but to give a brief example, let's imagine that we've deployed the `Pet` service onto Kubernetes. When you deploy a service, you specify how many copies of the service you want.[2] If, after deployment, one of those instances goes offline (perhaps because the underlying physical machine had an issue), Kubernetes would realize you had one fewer instance than desired and automatically spin up a replacement.

On the face of it, a tool managing the desired state of our `Pet` service instances seems to be a great solution. When deployed on Kubernetes, if one of our instances fails, Kubernetes will spin up a replacement! In addition, this could save us money if we're renting our computing resources —we won't need to pay extra money to have spare capacity sitting around.

The issue here is speed of replacement. There is going to be a period of time between our instance going offline, Kubernetes (for example) noticing, and a replacement instance being provisioned. Some of this is under your control—you could, for example, make sure that an instance of the `Pet` service can be launched and be fully functional as quickly as possible (and this is one area where container-based deployments typically do better than virtual machines). However, if the work keeps coming in while you are waiting for the replacement to be available, you might find that other instances of the `Pet` service start failing before new instances are available to shoulder the load.

One potential solution to the delay in provisioning a replacement would be to use desired state management in conjunction with some form of rate

limiting. The overall rate limit of a service will be determined by the sum of the rate limits of all instances that deliver that service (assuming even distribution of work). So if each instance could handle 20 concurrent requests, and you had five instances, the service as a whole would have a rate limit of 100 concurrent requests. If an instance goes offline, then the overall rate limit of the service would reduce to 80 concurrent requests. Reducing the rate limit being applied when an instance fails would protect the service until the replacement instance is available, at which point the rate limit would return to its previous level.

> **WARNING**
>
> If spinning up new or replacement infrastructure takes too long, it may not arrive quickly enough to help.

Reduce failure rate

The cascading failure here is caused by one of the `Pet` instances failing. So why not just stop that failure from happening?

In the case of the `Pet` service, we're worried about a computer going offline. So what could we do to reduce the chances of that happening? With a physical machine, ensuring the storage is redundant so that a single hard drive failure doesn't take the machine down is one option to consider. In some cases, redundant power supplies can make sense, especially as power supply failure is not unheard of. Ensuring that the location where your machine is running has an uninterrupted power supply could help further.

I could go on. The key thing to understand is that while many of these approaches to reducing the failure rate of an individual machine may help, they all cost money, and none of them will result in a cast-iron guarantee that the machine will not fail in some other way.

You might have a redundant power supply, but does that protect you from water leaking into the data center rack? Your machine's OS might be running from a massively redundant and reliable storage area network

(SAN), but what happens when the SAN catches fire? These examples aren't hyperbole; they are both real-world examples.[3]

Don't get me wrong—reducing the failure rate of a component in a system is often a sensible and cost-effective thing to do. But you cannot eliminate the chance that a specific component could fail, as we covered back in Chapter 1. Remember the example of the Google racks, where relatively cheap hardware was used, and components with a high failure rate were attached via Velcro? Rather than relying on expensive servers, Google accepted early on that servers could fail and built its systems with that failure in mind. Hoping a component won't fail is not a sensible approach if you are unprepared for when the worst does happen.

> **WARNING**
>
> Hope is not a strategy. Even the most robust computer can fail.

Cache Collapse

Caches are a widely used mechanism to help improve system scale and speed. By taking a value that has already been calculated and storing it in a way that means it is much faster to access in the future, the value can be retrieved more quickly, and pressure can be taken off parts of the system.

In the context of distributed systems, caches are often used to store data locally so that you don't have to fetch it from the origin. For example, when requesting a view of an order from a database, you would need to send a query over the network to the database to retrieve the data and then wait for the query to be executed and a view of the order sent back. You could then keep hold of that view of the order, storing it in memory. The upside of caching in this example is that subsequent requests to view the order will be returned much more quickly than if a round-trip to the database was still required, and in addition you may end up reducing load on the database itself. The downside is that the cached view of the order might be out-of-

date—an update to change the order in the database doesn't magically update your in-memory copy.

When caches fail, it can result in another type of thundering herd: requests previously dealt with at a cache will now head to the origin instead—potentially taking components not used to dealing with that degree of load offline.

Example: ShopCorp Catalog Cache Failure

In Figure 5-4, you can see three instances of the `Order` service, part of a wider ecommerce system for ShopCorp. One of the primary sources of information for the `Order` service is information about the items for sale, which is provided by the `Catalog` service. To speed up order processing, the `Order` instances make use of a cache to store the information we got from the `Catalog` service.

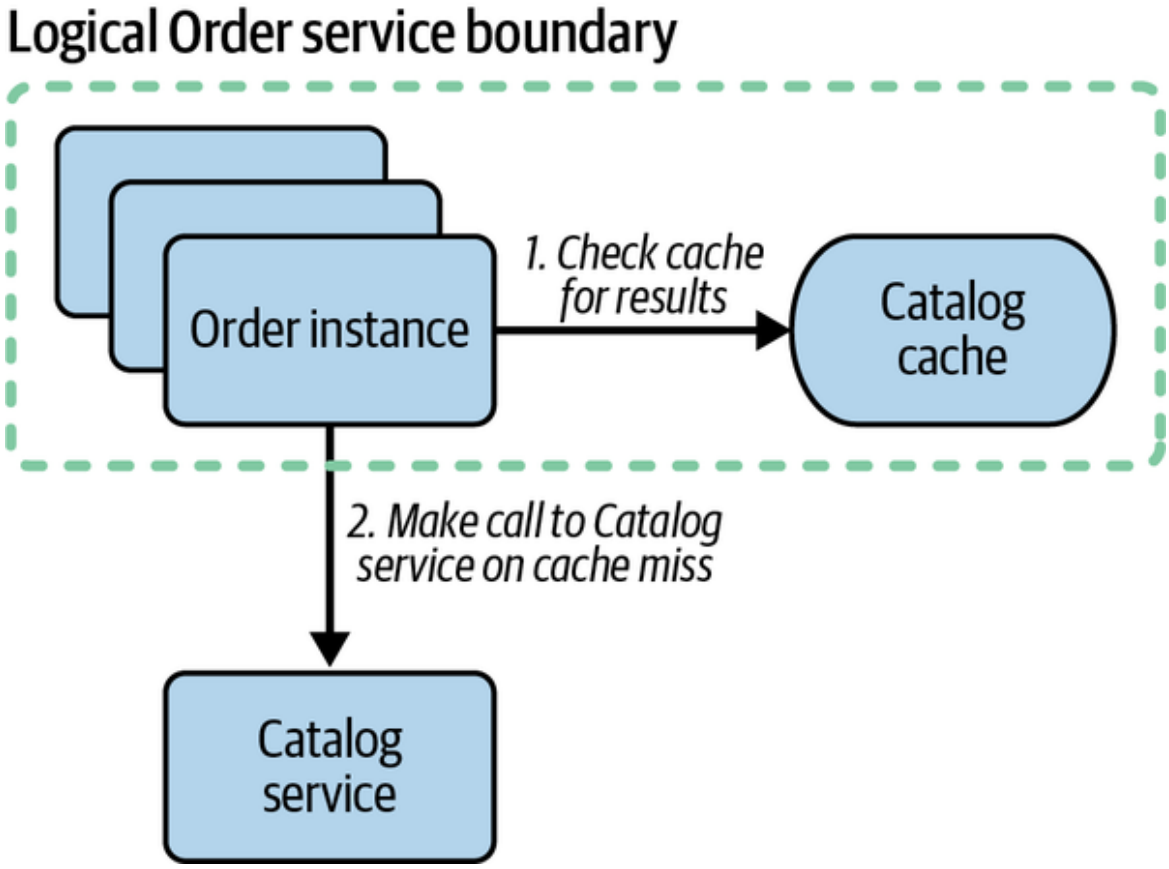


Figure 5-4. `Order` instances store information from the `Catalog` service in a dedicated cache

This cache is running as a separate process, and for the sake of this example you could assume it is running Redis, Memcached, or some other similar dedicated caching software. Given that this cache is there to support the `Order` service specifically, you could logically think of it as part of the `Order` service itself. Retrieving information from the cache should be faster than retrieving the same information from the `Catalog` service, with the added benefit that it will also reduce the number of calls the `Catalog` service has to handle.

When needing to retrieve information from `Catalog`, the `Order` instance will first check the cache. If the information is not available there, the `Order` instance will then fetch the required information from the `Catalog` service and put it into the cache. This is an example of a *cache-aside* pattern.

As ShopCorp grew and the order volume increased, the `Catalog` service was still able to support the increased order volume, as the majority of the calls to fetch catalog-related data are served by the cache. One day, though, a problem occurs—a hardware failure results in the caching node being brought down. It is quickly restarted. However, one of the problems with storing data in memory is that in the vast majority of computer systems, memory contents are not persisted between restarts. So this newly relaunched caching node has nothing in its cache. Suddenly, the requests that were previously being served from the cache are now instead thundering toward the `Catalog` service.

Mitigations

Perhaps ShopCorp is lucky, and when the cache fails, the `Catalog` service has enough capacity to serve the sudden spike in traffic. However, we know that the `Catalog` service won't be able to respond as quickly as the cache could. So ShopCorp could expect the order processing to be slowed even if it does withstand the initial herd. It's worth noting that if order processing speed degrades, assuming the same amount of work is coming in, then each `Order` instance would be handling more concurrent requests—the

relationship between concurrency and the speed of processing is something I'll come back to when we look at Little's Law in Chapter 12.

But what if ShopCorp *isn't* lucky? What if this thundering herd is sufficient to overwhelm the `Catalog` service? Let's look at some potential mitigations for this problem.

Cache in service instances

One option for ShopCorp is to remove the use of a separate caching node and have each instance of `Order` store its own cache of information from `Catalog` in memory. On the face of it, this has some nice benefits. ShopCorp would be removing the need for a separate caching node, simplifying its topology. Also, if the goal is speed, then fetching data from memory on the `Order` instance itself should be faster than fetching data from the caching node, as you avoid paying the penalty for the network calls.

This approach has a few issues, though. Firstly, if each instance of `Order` service stores its own cache, each of those caches will need to be filled up; this would increase the traffic on the `Catalog` service, as each `Order` instance has to fill up its own cache and would resort to making calls to the `Catalog` service when it experiences a cache miss. Additionally, one of the benefits of having a dedicated caching node is that the machine it runs on can be tuned for caching purposes—giving it lots more high-speed memory, for example. Ensuring that each instance of the `Catalog` service has the ability to store as much data in memory as the dedicated caching node could get expensive.

Fundamentally, though, to what extent does this approach solve the problem? You no longer depend on the caching node, but what happens if an `Order` instance is restarted? At that point, its in-memory cache is emptied, and it would need to fill it up. From a release process point of view, this can be mitigated by releasing changes to the `Order` service only via a slow, rolling update, which would involve gradually shutting down old versions of an instance and spinning up new ones in a managed way. This

could reduce the flood of calls to the `Catalog` service but would still be an added complication. And even if a rolling update process did work, that doesn't help in the case of an unplanned outage.

Use a more resilient cache

Instead of accepting the premise that the cache can fail entirely, how about we make the cache more robust? Some caching software provides the ability for you to run a cluster of nodes—the idea being that if one machine fails, other cache nodes can still provide access to the required data. We'll explore some of the implications of cluster-based software like this in more detail in Chapter 8.

Generally, machines can fail and go offline for a variety of reasons. At a fundamental level, even if this is abstracted from you, a computer runs on hardware, and hardware can fail—power supplies can stop working, network cables can get severed, floods can take out servers. For this reason, it is often highly sensible to have solutions that mean a single machine failing doesn't cause the entire service (in this case, a caching node) to fail.

This creates a new headache. We now need caching software that supports this sort of setup, which might limit your software options. In some cases, it might involve more expensive license fees to vendors, and even if that isn't the case, you will need more hardware, which is an increased cost. In addition, our system as a whole is getting more complex once again.

On the other hand, you may be able to take advantage of a managed service that supports a clustered model. AWS, Azure, and Google Cloud can provide caching services that can tolerate single machine failures, and in some cases (such as with Google Cloud's memory store or AWS ElastiCache) the service can replicate cache data across regions. Of course, while the public cloud vendors will no doubt do an excellent job of providing this for you, they'll also charge you for the privilege!

Improve recovery time

When bad things happen, recovering quickly is normally a good idea. In this scenario, the cache has failed and, as a result, the system will be under more load until the cache is once again filled up. So how about getting the cache back up and running more quickly, even in a situation where we are running only a single caching instance?

Redis and Memcached, both popular open source caching solutions, provide the ability to periodically flush a snapshot of the cache to disk. When the caching node is shut down and restarted, this snapshot can then be read, populating the in-memory cache so that it can start serving data more quickly. In our scenario, this would help greatly if the caching node was quickly restarted—there may be a small window in which requests would be hitting the origin (the `Catalog` service in our example), but once the caching node is relaunched and the snapshot file is read, the cache could once again start helping to remove load from the origin and speeding up request handling.

There are a couple of downsides here. With both Memcached and Redis, data is not typically written to disk the moment it is updated in the cache. This is by design. Caches are designed to be fast—if you waited for the updated data to also be written to disk, this would slow down cache writes. As a result, the flushing of updated data to a file is done periodically—by default Redis does this every 5 to 15 minutes, depending on how much has been updated. This means that there is potential for the data that has been written into the cache not to be persisted across restarts, which could increase cache misses in the wake of an outage. As long as you know how much data could potentially be lost, such a scenario is usually still very manageable.

Another issue here is that the data is persisted to disk. If the machine containing the cache caught on fire, the data might be lost too. So in a scenario where the caching node might need to be brought up on a different machine, you'd need to have some way for the snapshot file to be shared with other machines. Using a data volume that can be "attached" to different machines (such as Kubernetes volumes or AWS Elastic Block

Stores) might be one option, or persisting the file to a distributed filesystem like ZFS or similar could help.

Finally, you need to ensure that a replacement caching node can be brought up swiftly. If it takes you minutes or hours to bring the cache node back up, then the `Catalog` service is going to be at risk of a thundering herd for the duration. If the caching node is being run on a platform like Kubernetes, a replacement may well be spun up automatically for you—so running your cache in an environment where a desired state management solution can do the heavy lifting for you could be incredibly helpful.

These challenges aside, if your caching solution supports storing snapshots to disk, this seems, at least in the case of ShopCorp, like a somewhat straightforward thing to implement. It won't solve all problems, but it will at the very least help recovery time should the worst happen.

Disable the service

Coming back to a topic we've already touched on previously, it is better to fail fast than to fail slow. And it is better to fail in a managed way than to fail in a catastrophic way. If, when the cache fails, the resulting calls to `Catalog` would cause the system to grind to a halt, it might make sense to stop taking orders until the cache has been repopulated.

A less extreme version of this may be to implement some rate limiting at the `Catalog` service—perhaps reducing the amount of requests that will be processed until the cache is back up and running. Even a simple technique like load shedding, which we explore in Chapter 6, would help. This would allow us to still process *some* orders, which is almost certainly better than processing zero orders!

Mismatched Capacities

Sometimes you can end up generating your own thundering herd when trying to scale up parts of the system. This can especially be the case if you scale up one part of your system without taking other parts into account. It

can be like forcing more and more water into a pipe that doesn't have the capacity to handle it—eventually, something is going to burst.

Example: BankCo Pricing Grid

You may recall the story I told in Chapter 4 about working for BankCo, where I helped build a system to perform calculations across a large number of machines. One of the goals of the project was to be able to provide these calculations faster, and as a result we selected a technology called DataSynapse that allowed us to scale up the number of machines available for the calculations. Unfortunately, simply spinning up more machines didn't go to plan.

In Figure 5-5, you can see a heavily simplified version of the architecture we had built. Requests for calculations came into the queue and were in turn picked up once a calculation agent (which was running as part of a DataSynapse cluster) became available. Once the agent completed its calculation, this information was put into another queue so that the results could be picked up by the `Persistence` service and written to a database. There was one key component that we relied upon that wasn't under our control. One of the key inputs to the calculations came from the `Risk` service, which was managed by a separate team. This service provided a lot of factual information, such as details about bonds, interest rates, and the like.

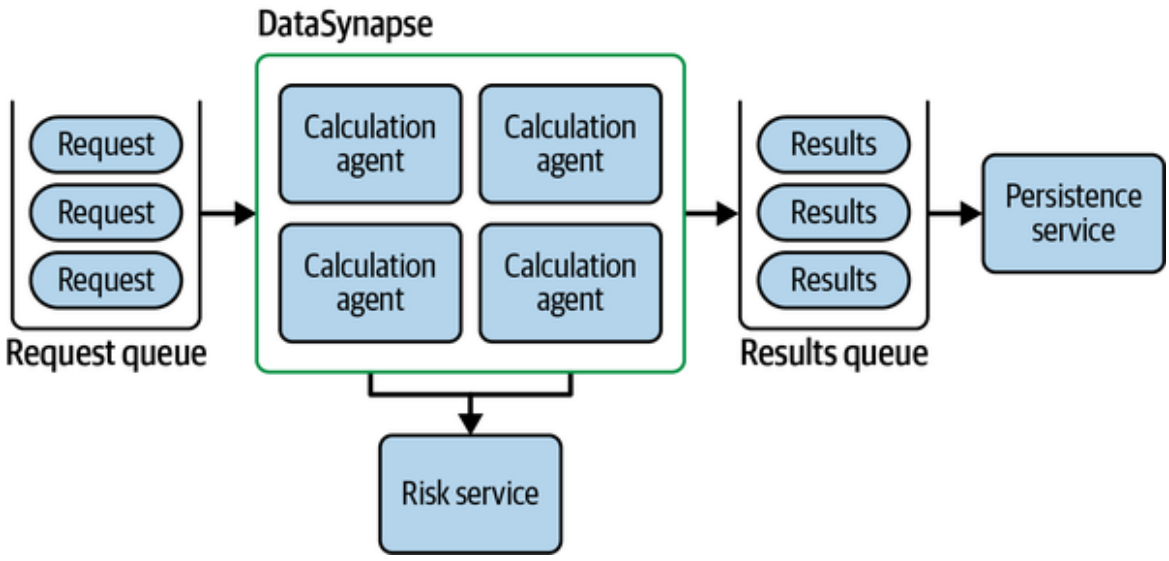


Figure 5-5. The calculation system

BankCo made use of a standby office, located outside of London. This was to ensure that if for some reason the central London location could not be used, essential bank employees would be able to move to the standby site to carry on their work. Because of the important nature of the work (or should I say, the amount of money the bank hoped this work would generate), it was essential that the standby site be operational at all times to ensure that it was always ready in the event of an emergency. As a result, it was full of desktop computers that were switched on but otherwise idle.

Scaling Up

At this point, our system had been running well, with around 25 servers acting as calculation agents, but we wanted to push it up a notch. One of the things DataSynapse allowed us to do was to run agents as screensavers. After a machine's screensaver had been triggered for a period of time, we assumed that it was idle, at which point the machine would be added to the pool of workers that could be used to carry out work. It didn't take us long to connect the dots between the availability of the DataSynapse screensaver agent and the large number of desktop machines sitting idle in the standby office location. After what proved to be an all-too-short chat, it was agreed to roll out the screensaver agent to all these terminals. We went very

quickly from having 25 calculation agents to having more than 100, as shown in Figure 5-6. This did not go well.

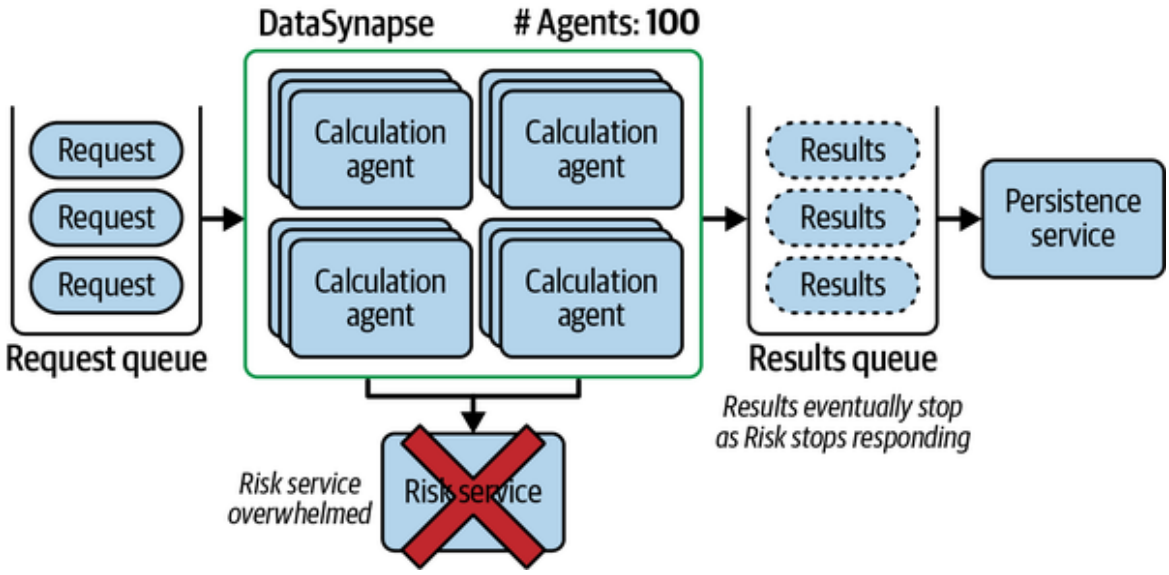


Figure 5-6. Increasing the number of calculation agents overwhelmed the `Risk` service

The problem was that we had increased the number of agents able to carry out calculations, but we hadn't scaled up any other part of the system. This meant that we created our own thundering herd of requests to the poor `Risk` service, which was not expecting this call volume. As a result, the service was effectively taken offline because of our barrage of calls. This was bad for us, as it meant we couldn't perform any calculations, but it was also bad for the other systems across the bank that also made use of the `Risk` service.

A Lucky Break

We did catch one lucky break. The service that picked up the results of the calculations wasn't crushed under the surge of results. This was partly because once the `Risk` service went offline, the calculations stopped being performed. But even as the load built up to breaking point, the `Persistence` service survived because of the way work was funneled to it. When calculations were completed, the results were sent into a queue. Luckily (and it was luck!), the message broker we were using was able to handle the increased number of calculations being sent to it. Thus the queue

in effect worked as a rate-limiting mechanism. The `Persistence` service had only so many threads that could consume from the queue at any point in time. The depth of the queue didn't increase the upper bound of how much work the `Persistence` service was able to pick up.

We theorized at the time that if we had also scaled up the number of `Persistence` service nodes, we would likely have created a different problem. Having more instances of the `Persistence` service might have allowed us to handle more of the work in the queue in parallel, but it would have increased the number of writes being sent to the database, and we could well have unleashed another thundering herd onto our poor, unsuspecting database server. Our database server infrastructure was shared with other teams, and so a problem there could have had a knock-on effect to other applications.

Mitigations

Although our attempt at scaling up our system did end in failure, it was an excellent learning experience, as is often the case with such things. Looking back, there were a number of things I took from this experience, although I know that only one or two of them actually occurred to me at the time. Hindsight is a wonderful thing—but at the same time, if I hadn't learned something in the nearly 20 years since the incident itself, then I would have to question what I've been doing with my time!

I should also highlight that there was no postmortem after this incident, so some of these issues likely would have come up back then if we'd gotten together and really dug into what happened and how things could have been improved. I'll return to the topic of postmortems and the role they can play in Chapter 14.

So, with all the benefits of hindsight, it turns out there were a number of areas where we could have done better.

Better load testing

While we had done some load testing of our pricing grid, it did not get close to the scale we planned to run at in production. In some cases, re-creating a load test environment identical or even close to the real production environment may not be practical. In our case, we didn't have access to enough infrastructure in the test environment to re-create the production environment exactly, but at the very least we should have acknowledged that from the start.

Gradual ramp-up

If we had ramped up the number of pricing agents more gradually, observing system behavior as we went, then we may well have spotted the problem we were causing for the `Risk` service before things went too far. Given that we didn't have the ability to create a production-like environment for our load testing, having a plan to gradually ramp up the number of agents would have been even more important.

Understanding what signals you will be observing during a ramp-up is vital. What parts of the system do you think will be stressed? Do you have visibility into those parts of the system? In our case, we had the ability to get information from only our own services, and thus we had no direct insight into what was happening for the `Risk` service. Ideally, we would have had access to that information, but even without it we could have involved the `Risk` team in the ramp-up exercise, giving the team the opportunity to let us know if we were pushing things too far.

Rate limiting

The `Risk` service kept trying to handle more and more traffic until it essentially stopped working. Ideally, as the `Risk` service got close to capacity, it should have started rate limiting, dropping calls that would otherwise have made the service unstable.

You can do your best to ensure that clients are well-behaved and don't send in a flood of calls that can't be handled, but as a service owner, you can't guarantee that clients will always do the right thing, Additionally, there may be a number of factors outside the control of the service owner and the

clients that can cause problems. For example, a distributed DoS attack or a network misconfiguration could result in unexpected spikes in the load. For this reason, service owners should prioritize having some defenses in place to protect against unexpected spikes in traffic.

> **TIP**
>
> A server needs to think defensively—it shouldn't rely on clients being well-behaved.

Improve communication

Looking back, our biggest failure was the lack of communication. As we explored in depth in Chapter 1, the systems we are building bring technology and people together. In our obsession with playing with the cool technology in front of us,[4] we'd forgotten about the other people in the system. If we'd stepped back and thought a bit more holistically about the other teams involved, it would have been obvious that working more closely with the team that looked after the `Risk` service would have resulted in a much better outcome. Better communication between teams before the event would also have been helpful. The `Risk` team never thought that it would need to handle the level of traffic we sent to it. The team had done its own testing and had a set of expectations about what "normal" load looked like. But that information was internal to the team and was not communicated to (or agreed to by) its consumers.

In Chapter 2, I introduced the concept of an SLO, a service level objective. If the `Risk` team had had a defined SLO that indicated what sort of traffic it was committed to handling (not that people talked about such things back in the mid-2000s), we could have used that as a target to work against. And if we had found that the SLO the `Risk` team offered didn't match what we needed to handle the growing size of our pricing grid, then that would have been a great trigger to have another conversation.

TOO MUCH OF A GOOD THING

Sometimes a technical improvement can cause more headaches. Java 19 (re)introduced the concept of virtual threads. Previously, for much of Java's life, a thread in Java mapped to an operating system thread. This provided an upper bound to how many threads a Java Virtual Machine (JVM) could handle in many situations, in turn reducing how many concurrent requests a JVM could manage. With Java 19, it was once again possible to create virtual threads (something that had been possible to do very early on with Java, although the earlier virtual thread implementation was a different beast). An existing Java client switching from native to new virtual threads could end up being able to generate much more traffic, and from anecdotal reports I'm already hearing of situations where this is resulting in thundering herds.

In general, the option to run virtual threads seems to be a good thing and in many cases should benefit the amount of work a JVM-based system can process. However, you probably want to do some testing first to ensure you don't cause system instability. Starting with server rather than client processes is probably the right choice, and all clients should be able to make use of connection pools to limit the number of calls an individual client process can make.

Roaring Success

> *Twitter is doing us a service, with its lack of stability, in illustrating the dangers of centralized systems. We do need to figure out how to build a Twitter-like system with all the advantages of centralization and none of the disadvantages.*
>
> —Dave Winer, 2008

Sites suffering from initial surges in popularity were not uncommon during the various dot-com booms. While as an industry we seem to be a bit better

at understanding how to scale systems up, spikes in popularity can still result in thundering herds that can bring our systems down.

In fact, the cycle of "grow quick" isn't just an obsession of the 2000s. Until recently, the relative abundance of "cheap" money (caused by ultralow interest rates) resulted in many VC-backed startups throwing money into scaling quickly, in the hope of one day being an actual proper profitable company. For many of these organizations, we can summarize their business model as follows:

1. Grow users
2. ?
3. Profit!

A focus on growing the user base means that startups are frequently looking to incentivize as many users signing up as possible. At the same time, the success of acquiring these new users can bring instability, which makes getting to step 3 in the business plan seem unlikely.

The rush of users to your new product may be welcome—and as an industry we have gotten better at dealing with user growth (largely because of the lessons learned during the various dot-com successes and failures). But it is still all too possible for you to create a system so attractive to potential new users that you end up inciting your own thundering herd.

Compared to some of the other thundering herds we've looked at so far in this chapter, the load generated by this type of overnight success is a little different. Firstly, you often get a bit more notice about increasing popularity, and this means that measures such as securing extra hardware can often be done ahead of time. By contrast, the types of calamity caused by a failing cache or a DoS attack are often hard to see coming, and those particular thundering herds can be on you before you have time to react.

On the other hand, a sudden increase of new users may take longer to dissipate. And while with other herds the goal may be to ride it out, perhaps degrading functionality or even shutting things down entirely to ensure you

survive, that tactic could be catastrophic if you are an early-stage startup aiming to grow a customer base.

Examples of public systems being taken down by unexpected demand are commonplace, even if the underlying lessons to be learned from each example may differ.

Example: Twitter

Back in 2007 I was fortunate enough to be invited to speak at a conference called XTech, which that year was held in Paris. The conference focused on the technologies around Web 2.0 (which at that point was in full flight), and I had somehow slipped through the net to talk about database change management.[5] During the conference, I went to a talk by some of the lead engineers at an up-and-coming new type of social network called Twitter, which was starting to get a bit of interest at the time. I actually remember signing up during the talk just to see what all the fuss was about.

I forget the exact title of the talk, but the gist of it was that the engineers were outlining how they had decided to essentially build their own message broker. In a moment of extreme smugness on my part, it appeared to me that the presenters seemed unaware of the existence of other message brokers and of what would be involved in building one. The talk was unfortunately interrupted, as at least one of the presenters had to rush offstage because Twitter had gone down. My smugness only increased.

During Twitter's early days, system outages were so common that the error page showing a whale being lifted up by a flock of orange birds became an all-too-familiar sight for users of the growing platform. The "fail whale," as the image became known, came to represent Twitter's inability to keep up with demand. It took a reasonable amount of time before the system stabilized to the point where it could be considered a reliable service—Twitter's ability to survive the US election night in 2012, five years after that conference talk, was considered a watershed moment in the system's ability to cope with its popularity.

Example: Healthcare.gov

For another real-world example of a product being too popular, we can look to Healthcare.gov. In 2010, a bill called the Affordable Care Act was signed into law in the US. The goal of the bill was, as the name suggests, to make healthcare more affordable. As a result of the act, a new online service, Heathcare.gov, was created, with the Centers for Medicare & Medicaid Services (CMS) overseeing its creation.

The need for Healthcare.gov was real. Healthcare in the US was (and is) very expensive. As of 2010, when Healthcare.gov initially went live, figures from the Organisation for Economic Co-operation and Development (OECD) showed that average healthcare spend in the US far exceeded that of all other countries, with most countries spending less than half of what the US spent per person.[6] While the reasons for the differences are far too complex to explore in this book, what is clear is that the opportunity for people to access more affordable healthcare insurance was always going to be popular.

Healthcare.gov initially went live in 2010 with a simple system to provide users visibility into the costs of public and private healthcare options available to them. This initial launch appeared to go well, but it was ahead of a much more fully fledged system that would provide a marketplace where Americans could purchase health coverage from a variety of options directly through Healthcare.gov itself. To say that the rollout of the marketplace in 2013 didn't go well is an understatement.

At the start of October 2013, the Healthcare.gov marketplace was made available to users in 36 states. While the goal of making the system available to as many people as possible was laudable, it was clear early on that the system was not able to cope with the actual loads. Demand in the early hours after launch was five times greater than expected, causing the website to crash within two hours. On the first day of its launch, as a result of the crashes and other bugs, only *six* users were actually able to sign up for a health insurance plan.

A significant amount of work was done in the wake of the initial launch issues. As of 2024, 21.3 million Americans were covered by insurance provided through Healthcare.gov. So while the initial rollout may not have gone to plan, it appears to have fared well since. That said, there is a lot to be learned from what happened and from the work required to fix it.

FURTHER READING ON HEALTHCARE.GOV

The full story behind Healthcare.gov is more than I am able to go into in this book; as with any complex system failure, many factors were at play in terms of why things didn't go to plan and what kinds of mitigations were required to fix things going forward. While many helpful overviews of what happened at Healthcare.gov are available, I'd caution against taking all of them at face value. Many of the commentators who covered the failed rollout did so from an external perspective, with their own bias coloring their assessment, while often not being in a position to assess all the available information. That said, I found the case study of the Healthcare.gov rollout by Henrico Dolfing to be a good jumping-off point for my own research, and the full report into the launch by the US Office of Inspector General is great if you want to dive into more details.

Mitigations

As the case studies from both Healthcare.gov and Twitter show, while the nature of the failures can appear similar, the causes and impacts of a thundering herd (a stampede of prospective users) can differ wildly. In the case of Healthcare.gov, the nature of the system meant that its launch failure was the subject of a detailed analysis. Aside from case studies done by independent observers, the official report by the US Office of Inspector General provides a number of great insights.

As such, we have a number of different mitigation strategies to consider.

More realistic load testing

During the development of Healthcare.gov, the system had been tested to expect 2,000 concurrent users, which was well short of the actual demand. Only a few months after launch, significant effort had to be undertaken to rework the system to handle a more realistic 25,000 concurrent users—showing that the original estimates were off by a factor of over 12. It was clear that the expected loads used during testing were completely unrealistic.

In the case of Twitter, given its relatively unique nature at the time, it was difficult to predict how it would actually get used. It was originally positioned as a microblogging site with limited interaction between users, and so initially at least it was natural to compare how it would be used with existing blogging solutions, and any load testing would have been based on that. However, once people started using Twitter, a lot of the more interactive elements emerged organically, and this pushed the system into unexpected directions.

Scaling your way out of trouble

If your product becomes successful (or at least interesting enough) to generate a surge of new users, then an obvious answer is to scale up to handle the load. Typically, when time is short, the easiest way to scale up is to throw more computing resources at the problem. Assuming your system has been built with this in mind, sometimes just adding more machines can be the answer.

You can think of this type of thundering herd very much as a wave. It builds and builds, and then it crashes and subsides. Often the early onset of success causes a flood of interest, but perhaps only a small portion of those early users stick around to become long-term customers. This means that you need to be cautious about spending money to ride out this initial wave if doing so means you're locked into an ongoing spend that may not be justifiable, or you've sunk a significant portion of your cash into infrastructure or services that you can't get back.

This is where the public cloud really shines. If your system is on the public cloud, it is significantly more viable to spin up additional resources to ride

out this initial wave of traffic and then shut down those resources when the user traffic slows to more reasonable levels.

The issue is that even if you do feel that the public cloud is the best way to help you scale your way out of trouble, moving there can be a complex undertaking if you aren't already running there. As such, a public cloud migration tends to be a longer-term play rather than something you can do quickly while dealing with scaling headaches. Healthcare.gov did eventually decide to move to AWS, but the process took years to complete. Twitter also started to make use of AWS in 2020.[7]

The public cloud also makes vertical scaling more viable. Rather than simply adding more copies of your software, it can be even easier to just swap out smaller machines for bigger, more powerful ones. The public cloud providers make it easy to change the instance types of your machine, perhaps swapping in a more powerful version that has more memory or CPU. If you were purchasing your own physical machine, this would be a sunk cost—if the load reduced to the point where a more powerful machine was now sitting largely idle, it would be difficult to recoup the money spent purchasing and installing it in the first place. On the public cloud, though, you could just switch back to an earlier instance type and start saving money straightaway.

You might think it odd that I talk about the sensible use of money here, but this book is about resiliency as a whole. If you make technical choices that result in you wasting money, then you may well have less money to spend in the future, which could easily impact your ability to maintain (or improve) the resiliency of your software. Coming back to the themes we discussed in Chapter 1, if you look at your software as a broader system, including people and technology, you start to see how everything is related.

It is worth noting that a system typically has to be designed to allow for dynamic scaling. Aside from vertical scaling, systems often have constraints that mean that throwing additional computers at a problem may not help. As a simple example, if a relational database is bottlenecked on writes, additional computers are going to help only if the data can be broken

into shards, with writes distributed across multiple machines. This is a complex undertaking, and providing more hardware is often the simplest part of it. It seems likely that both Twitter and Healthcare.gov may well have had similar internal system design challenges. Netflix designed its cloud architectures specifically to take advantage of the ability to scale horizontally; doing so required building that concept into a lot of the design principles that guided both technology choice and implementation—for example, its selection of Cassandra, a database that allows for horizontal scaling for both reads and writes.

Limiting sign-ups

To perhaps a greater extent than any of the other scenarios we look at in this chapter, generating your own thundering herd due to building something people want is at the very least a situation over which you have some form of control. This means you can put things into place that can help reduce the chances of being overwhelmed with load.

A common technique to reduce the pace of growth to an acceptable level is to gradually roll out your product to your users. Starting with a small, closed beta can be useful to iron out any egregious issues that could derail a more widespread launch.

Invite codes are a clever way of creating and managing a closed beta. You issue a number of codes to people, which they in turn can hand out to their friends. You can control how many invite codes are available, and by extension you can control how many people can sign up in the first place. As you get a handle on what load your system can handle, you can increase the number of available invite codes, before removing them entirely once you are happy to throw open your doors to the whole world.

Invite codes also have another potential benefit: they create an air of exclusivity. Lines of people outside popular venues create more buzz around that venue.[8] So you might find that the use of invite codes can actually increase the demand for your product without increasing load. The use of a waiting list for invite codes can then help you get some concrete data around your potential user growth, giving you time to plan accordingly.

While I find the idea of handing out invite codes for people desperate for cheap healthcare to be distasteful in the extreme, a more controlled rollout for Healthcare.gov may well have resulted in a more stable system, meaning people would have been able to access the healthcare they needed more quickly. Rolling out the marketplace on a state-by-state basis may have been a way forward (rather than launching it in 36 states simultaneously), though I appreciate that there could well have been some significant political ramifications of a decision like this.

Better alignment

In the immediate aftermath of the launch of the Healthcare.gov marketplace, another key change was made to break down the barriers between CMS (the federal agency tasked with creating the website) and the contractors hired to help the agency. Prior to this, coordination between CMS and its vendors had been poor, leading to miscommunication and delays in making decisions. From the testimony of Assistant Inspector General Erin Bliss at a 2016 Senate hearing:

> *Following the launch, first with the technological team and then more broadly, CMS promoted a horizontal culture that was "badgeless" and "titleless," [encouraging everyone on the Federal Marketplace project to work] as a single team regardless of employer or job title…According to CMS, this change in culture fostered a greater sense of mission and teamwork that further improved daily operations.*

While I can't speak to the culture inside Twitter during its early days, I have seen firsthand what can happen in startups when people aren't aligned. When things are stressful, it's vital that people are pulling in the same direction.

THE VITAL COMPONENT

People are your most valuable asset. People design, build, test, secure, operate, and use your software, and they need to work together. If people cannot work together effectively, the system they are trying to create will suffer, and any attempt to build a resilient system will fail. One of the biggest improvements made in the aftermath of the Healthcare.gov rollout was to have vendors and federal employees working together as a team.

From a resiliency standpoint, massively increasing the number of users/customers puts a strain not only on the computing resources available to you but also on the human operators. In the same way that our computing resources can become saturated with work to the point where they become unusable, so too can people. Humans, of course, are not computers. One of the main ways in which humans and computers differ is that if you restart a machine, it doesn't suffer the aftereffects of being totally overloaded sometime before. The effect of overwork on people, though, can be long-lasting.

Working crunch time shouldn't be normal, or normalized. If your business model relies on people working so much that they break, then your business model won't be long for this world. I am a fan of the concept of sustainable pace, which Extreme Programming popularized early on:

> *Working overtime sucks the spirit and motivation out of your team. When your team becomes tired and demoralized, they will get less work done, not more, no matter how many hours are worked. Becoming over worked today steals development progress from the future. You can't make realistic plans when your team does more work this month and less next month.*

Remember, in the wake of a thundering herd, people will be tired and burned out. Thus when times become calmer, you should take the opportunity to give people a break, if needed. It's also vital when the

energy levels are topped up that you try to learn from what happened and decide on improvements that could be made in the future; this is a topic we'll explore in more depth in Chapter 15.

Distributed Denial of Service Attacks

A distributed denial of service (DDoS) attack consists of a coordinated attack on your site from a large number of machines. The goal of the attack is to interrupt service to the point where normal operations are no longer possible. It is typically a targeted attack that may be launched as an attempt to extort money.

Far from being accidental, a DDoS attack is a maliciously generated thundering herd that has been weaponized to achieve the aims of the attacker.

Examples

Depressingly, DDoS attacks are so commonplace that there are many examples to pick from. Cloudflare's 2025 Q4 DDoS threat report shows that in 2025 the number of DDoS attacks more than doubled compared to the previous year. And it's not just the number of attacks that are on the increase—attackers are increasingly targeting higher-profile targets.

In 2024, Microsoft's Azure public cloud was targeted by a DDoS attack that affected services for customers over a 10-hour period. The fact that a large-scale public cloud vendor was impacted by such an attack is notable, because one of the reasons some people feel safer with a public cloud service is that they assume the vendor is better able to defend itself from attacks like this.

Along similar lines, in late 2025 Cloudflare was targeted with what may have been the largest DDoS attack in history. The attack had the goal of taking down not just Cloudflare but all its customers by extension.

Thankfully, Cloudflare was able to fight off the attack, which at its peak was sending 20 million HTTP requests per second.

Mitigations

The nature of DDoS attacks can vary. Broadly speaking, there are three types of DDoS attacks:

Volumetric

: Overwhelming your system with traffic and thus causing your computing resources (such as network bandwidth) to be exhausted

Protocol

: Exploiting vulnerabilities to cause general disruption

Application-layer

: Exploiting vulnerabilities in the application itself

Depending on the nature of the attack, different mitigations may help.

Keeping things patched

Vulnerabilities in commonly used networking protocols are often exploited for DDoS attacks. Examples include the ping of death, in which outsized packets are sent as part of a network ping. Many of these types of attacks have since been neutralized through improvements made to the implementations of these protocols, but older devices could still be vulnerable. Making sure that your system is updated to take advantage of newer implementations of these protocols will help reduce the impact of protocol-driven DDoS attacks.

In 2023, an exploit of the rapid reset capability of HTTP/2 emerged that allows a small number of bots to cause significant amounts of damage. Essentially by constantly opening and closing HTTP streams, an attacker's bot aims to exhaust a server's resources. Technically speaking, this attack

isn't making use of an exploit in the HTTP/2 specification—it's working as designed. Many vendors have since released patches to mitigate this form of attack.

Scale up

The goal of volumetric and some application-layer DDoS attacks is to overwhelm your computing resources. This opens up the possibility of increasing the amount of computing resources you have available to you in an attempt to provision more infrastructure than the attacker can saturate.

The problem with scaling up in the face of a DDoS attack is that it requires either an investment in hardware you keep lying around for such purposes or the ability to dynamically scale up resources. Either scenario is costing you more money, and there is no guarantee that you'd be able to scale up enough to handle the load that the DDoS attacker can bring to bear.

> **WARNING**
>
> Your system won't remain resilient for long if you've got no money left.

It's also worth noting that one of the computing resources that a DDoS attacker will most often look to saturate is network bandwidth. Provisioning additional network bandwidth is typically more difficult than firing up another virtual machine on your public cloud.

Multiple networks

Rather than having all your services in the same network segment, you can run different parts of your system on different networks. This could be done by breaking your network into a series of virtual local area networks (VLANs). Doing this typically requires a networking layer that supports such configuration, although this is very common nowadays and is widely supported. Separating out the various services into different networks could

mean that a DDoS attack against one part of your system would leave other parts of your system untouched, thereby reducing the impact of the attack.

A more aggressive approach might be to distribute your system across multiple cloud vendors or private data centers, further complicating the task of the DDoS attacker and giving you more separation. Splitting out networks in this context is an example of creating a bulkhead. In shipping, a bulkhead is a part of a ship that can be sealed off to protect the rest of the ship. If the ship springs a leak, you can close the bulkhead doors; you lose part of the ship, but the rest of it remains intact.

The idea with a bulkhead is that you reduce the blast radius of a failure. In the case of a DDoS attack, you are attempting to limit the damage such an attack can cause—the bulkhead ensures the impact is felt only within the bulkhead itself. We'll look at other examples of bulkheads throughout the book.

Degradation functionality and blackhole routing

Another potential option would be to degrade the functionality of your system so that it needs fewer computing resources to run in the first place. This might include turning off features that aren't as critical, or perhaps even switching the system into a read-only state. This is a concept called *graceful degradation*, and it's a pattern we'll explore in more detail in Chapter 11.

To give you an idea of graceful degradation in action, I can share an example of a technique from a real client. When I worked at *The Guardian*, we had a standby version of the website that was a set of static HTML pages generated by crawling the normal site. Serving a simple static website is much easier to do (computationally speaking) than serving a website that relies on backend processing to generate the content. So switching over to the statically generated website still gave the users of the site some functionality. Depending on the nature of the DDoS attack, this could have been sufficient to keep at least part of the site's functionality live.

However, even if you are able to reduce the amount of computing resources it takes to run your system during an attack, there is still the possibility that turning off noncritical functionality won't be enough, and the DDoS attack can still exhaust all available resources.

A more aggressive approach is to carry out *blackhole routing*, where traffic is silently dropped. This can be used to target the malicious traffic, but it could impact legitimate traffic as well, especially if the attack is targeting your application directly, as it may be hard or impossible to differentiate between legitimate and nonlegitimate traffic.

Rate limiting

If the DDoS attacker is attempting to overload your system with work, limiting the rate of work can help. In Chapter 6, we'll look at how to reduce the rate at which work is allowed. That chapter will focus more on rate limiting from normal traffic patterns, but one of the topics we cover in that chapter, load shedding, could be useful here. With load shedding, you throw away work above a certain threshold to keep the system stable. This can be especially useful if an attacker is performing an application-level DDoS attack in an attempt to overwhelm part of your system.

In a way, load shedding is doing something similar to blackhole routing. The main difference is that blackhole routing tends to just drop all traffic, whereas with load shedding we drop work once a certain threshold is reached; there is still an attempt to do some work with load shedding. The problem with both techniques is that they can impact both legitimate traffic and that from the DDoS attack indiscriminately.

Web application firewalls and content delivery networks

The reality is that DDoS attacks can vary greatly, not just in terms of the sophistication of the attacker but also in the amount of resources a malicious party might bring to bear. It's possible you work in an organization that is large enough to have the expertise in place to deal with the wide variety of DDoS attacks you might face, but in my experience it's

unlikely. So one of the best ways to mitigate DDoS attacks is to make use of software and services created by experts in this space.

Web application firewalls (WAFs) can be installed on the perimeter of your system and can help mitigate against some forms of DDoS attacks. The exact capabilities of these types of products can vary significantly, and if you decide to make use of one, you are going to be adding another piece of complexity to your system. Some public cloud vendors provide WAFs as a service, which can reduce the cost of managing them, although you should note that public cloud vendors rarely give you things for free.

A content delivery network (CDN) such as Fastly, Cloudflare, or Akamai provides the ability to deliver your website's content efficiently through a network of geographically close caches. Your website might be served from a computer in Mumbai, India, but the right CDN can ensure that a user in Wellington, New Zealand, actually loads your website from a set of servers local to them.

CDNs allow you to decrease the load on your own servers by acting as a cache on your behalf and also further speed up the end-user experience by ensuring that requests are routed to caches that are local to the user.

Because of their nature, CDNs are in a position to provide significant DDoS mitigation. Aside from the expertise that these companies have internally, most CDN vendors provide services that can help mitigate a DDoS attack. They can filter out malicious-looking requests at different layers of the networking stack and provide rate limiting at the edge, and in many cases they will waive the excess CDN charges you might otherwise accrue during an attack.[9]

Coming back to the example of the HTTP/2 rapid reset attack I covered earlier, both Cloudflare and Fastly were able to roll out mitigations for the attack, and they did so far more quickly than could be expected of the average corporate networking team.

In a nutshell, if the DDoS attack is small in scope or limited in sophistication, you might be able to scale your way out of trouble—or shut

down some functionality to try and keep critical functionality running. For anything more than that, you probably want a CDN.

Summary

I could detail a dozen more examples of a thundering herd problem laying waste to a system, and I suspect you could think of a few that you've experienced as well. Despite the varied types of thundering herds we've looked at, though, hopefully you saw how the same mitigations came up again and again, including:

Degrading functionality

By shutting down some features of the system, critical functionality can continue to operate.

Provisioning redundant computing resources

Extra computing resources can absorb the impact of a thundering herd.

Improving recovery time

Thundering herds caused by component failures can be mitigated if the failed components can be replaced quickly.

Reducing blast radius

By creating a bulkhead, you can ensure that component failure has a limited impact and thus avoid cascading failures.

Setting clearer requirements

Many thundering herds are caused by a lack of clarity about what is expected—with systems designed to handle only a fraction of what they will actually face.

Throttling system usage

> Reducing the amount of work being done, either by reducing sign-ups or through more general rate limiting, can keep the herds down to manageable levels.

The scenarios in this chapter are far from exhaustive, but as you've seen, not only can the impact of thundering herds be mitigated, but it might be possible to stop them altogether.

Many of the mitigations I shared in this chapter warrant further exploration, and with that in mind, in Chapter 6 we'll be doing a deep dive into the topic of rate limiting.

---

1 Kief Morris, *Infrastructure as Code*, 3rd edition (O'Reilly, 2025).

2 I'm simplifying here—you actually specify how many instances of a pod you want and separately state how pods map to your service definition.

3 In the case of the SAN failure, for complex (internal political) reasons it was almost a week before we were told that our system failure was caused by part of the SAN catching fire.

4 And don't get me wrong—in the mid-2000s, software like DataSynapse was *very* cool, especially for those of us who had yet to learn about map reduce.

5 Yes, I'm that cool.

6 Unfortunately, although the gap isn't quite as large as it was back in 2010, as of 2025 the US still spends in excess of 45% more per person than the next closest OECD country.

7 Twitter signed a five-year deal with AWS in 2020, and it also made use of Google Cloud. For some reason, shortly after Twitter went private under new ownership, it stopped paying some of its bills from both vendors.

8 Berlin techno club Berghain is legendary for its huge lines and its hard-to-fathom door policy, to the point that the internet is littered with articles on how long the lines can be and how you can increase your chances of getting in.

9 This is the case for Fastly, at least, although I am unsure whether this is commonplace.

Chapter 6. Rate Limiting

Much of what I've covered so far has to do with the things you can do on the client side to deal with problems getting an appropriate response from a server. Often when a client is struggling to get a response, it's because the server it's talking to has too much work to do and is overwhelmed.

In this chapter, I'll be looking at approaches to reducing the amount of work being processed, a technique called *rate limiting*. As you'll see, rate limiting can be implemented in numerous ways, and I'll be sharing some concrete patterns that can help you implement rate limiting in your own system.

Before I get into the details, though, let's step back a bit and consider what your options are when a server has too much work.

TERMINOLOGY

It's probably worth clearing up some of the terminology I use in this chapter. When I talk about a *server*, I am referring to a single machine that serves up functionality over a network connection. If you are working on a service-oriented architecture (like a microservices architecture), then you are more likely accustomed to thinking in terms of services. A *service* is going to have one or more instances, each of which runs on its own single server.

Ways to Handle Having Too Much Work

When a server has so much work to do that it is in danger of falling over as a result, you have five options to consider:

1. Just let it fall over.

2. Throw away some of the work (load shedding).
3. Reduce the amount of work being sent (backpressure).
4. Queue up the work.
5. Provision more computing resources dynamically.

I'm going to briefly explore each of these ideas before drilling down into more detail on load shedding and backpressure.

Just Let It Fall Over

If a server is being hammered with requests, you could just let the server fall over. Now, I suspect this is unlikely to be desirable behavior for you, dear reader. You've made a decision to read this book likely because you don't want this sort of thing to happen.

The problem with just allowing a server to fall over is that it is uncontrolled. Requests will be terminated halfway through being processed, and clients will be left confused about what state their work is in. In the worst cases, an uncontrolled failure could result in data loss, and perhaps a lengthy time recovery time.

In fact, if you were in a situation where you thought a server was about to fail, it might actually be better to shut it down yourself in a controlled fashion rather than let it fall over in an uncontrolled state.

But we can, of course, do better than this.

Throw Away Some of the Work (Load Shedding)

If you have too much work coming in, one answer is to just throw away some of it—this is a pattern known as *load shedding*. Some work will get rejected, but other work will be done, and the system will remain up and functional (at least in some capacity). The trade-off here is about prioritizing the health of a system over attempting to serve an individual request. By throwing away some work, you may reduce contention on the

server while it still has a good chance of surviving some sort of onslaught. Remember: resilience engineering is about trying to make sure that as many things go right as possible; it's not about trying to make *everything* work.

I'll explore the mechanics of load shedding later in this chapter.

Reduce the Work Being Sent (Backpressure)

If a server has too much work, it would be ideal if its clients could reduce the work being sent to it. This pattern, called *backpressure*, allows us to limit the work flowing through a system to keep the system stable. So in a situation where a server is being overwhelmed by a large number of requests, if these requests can be "pushed back," it reduces pressure on the server itself.

Backpressure can be implemented in a number of ways. Later in this chapter, you'll see how *circuit breakers* can be used on the client side to stop requests being sent, and I'll show how adding information to the client-server protocol can make backpressure even smarter. I'll also share the *leaky bucket rate-limiting* pattern, which can be very effective in making load from a client more consistent.

Queue Up the Work

Rather than throwing work away or reducing the amount of work you can accept, a server can decide to queue up the work, with the aim of working through the queue as quickly as possible while keeping itself stable.

The queue can be thought of as a throttling mechanism. Clients can keep sending in work, and this work builds up in the queue. The server can limit how much work it can process. The trade-off here is that while this queue is building up, the latency of the work increases.

I explore the concept of queuing in more detail in Chapter 9, where we'll look at how message brokers can help.

Dynamically Provision More Resources

Often the issue with having too much work to do is that the amount of computing resources available is constrained—your server has run out of CPU, memory, I/O, storage, or something else. I'm sure many of you are running on infrastructure that allows for computing resources to be dynamically launched, which then opens the possibility of spinning up more computing resources to handle the increased amount of work.

So now you've seen some of the options for dealing with too much work; for the rest of this chapter, we'll focus on two of them—load shedding and backpressure.

Pattern: Load Shedding

An operating mode in which a server starts throwing away requested work to keep itself operating within acceptable parameters.

If a server attempts to handle everything, it can collapse under the load and end up serving nothing. A sensible alternative is to throw away some of the work; this is known as *load shedding*. With load shedding, once a threshold is reached, a server starts rejecting work to keep itself operating within safe parameters.

If you can drop some of the work you are being asked to do, you may still be able to get the rest of the work done. If there is a choice between processing *zero* work items and processing *some* work items, the answer starts looking pretty clear.

> **TIP**
>
> It's better for a system to reject some work in favor of keeping the system healthy, rather than trying to process all work at the risk of bringing everything down.

How It Works

When implementing load shedding, we have to look at a few specific details. Firstly, we have to decide what is going to be tracked to determine whether a server is approaching its capacity. After that, we need to decide how to set a safe threshold. Next, we have to decide where the load shedding will be triggered. Then, when work is rejected, we need to explore how to communicate this with the client.

Choosing what to track

With load shedding, you will track one or more metrics and set a "safe" threshold, above which load shedding kicks in. One of the simplest mechanisms is to find a way to track the total number of work items the server is currently processing. For example, if your service exposed an HTTP API, you could track the number of concurrent requests. You would then set an upper boundary, and any requests received beyond that limit would be rejected. If work is being processed from a queue, then the queue depth could be used as a trigger for load shedding.

CONNECTIONS VERSUS REQUESTS

It can be easier to monitor the number of active connections than to monitor the number of requests. The problem with monitoring connections is that some protocols will keep connections between client and server open even if they're idle. In addition, protocols like HTTP/2 can send multiple requests over a single connection.

Arguably, tracking something like the number of concurrent requests is in part a proxy measure for the underlying contention on resources. The thinking goes that as a server does more work, more computing resources are being tied up, so you get closer to the point where the server is overwhelmed.

But if the state of the computing resources is really what you care about, then why not track the underlying resources instead? This would help in situations where the load generated by different types of requests can vary.

The challenge of using more sophisticated metrics for triggering load shedding is that it can be difficult to obtain the underlying metrics of the server within the context of handling a request. Inside the process running your service, getting a request count isn't too difficult. But are you able to access the underlying machine's CPU or memory use?

If you are configuring load shedding at a proxy layer, such as a load balancer, does the proxy have access to the load characteristics of the machines it is talking to?

Moreover, you go from having one simple measure to use as a trigger for your load shedding—the number of items of work—to having multiple potential metrics to track. Do you set a threshold for memory? CPU? Network I/O? If you change the type of server you run your code on (which is easy to do with virtualized compute), all these thresholds would likely need to change. But if you're tracking the number of requests, switching to a bigger underlying machine would just require changing a single threshold.

So while I have seen the state of underlying computing resources used to trigger load-shedding, it's not common. The most common thing to track as part of load shedding decisions is the number of concurrent requests. It's important to note that you will still want to capture metrics associated with the underlying computing resources, as this information can be vital in understanding failure modes, capacity planning, and more, as I outlined in Chapter 2.

Setting thresholds

Once you've decided what you are going to use to track the current amount of work being processed, the tricky part is identifying the upper bound. Is the limit for a server 100 concurrent requests? 50? 25? Typically, working out the acceptable level for your server will involve observing it under different load patterns. Gathering information from production can be incredibly helpful, but a load test can be very useful too, as it can help to run break tests. You can generate enough load to the point where the system fails in order to understand where to draw the line.

The threshold you set should line up with any specific service level objectives (SLOs) you might have. For example, if your SLO indicates that your service should be able to accommodate up to 500 orders a minute, then you need to make sure that any load-shedding thresholds are set to take this into account. The SLO is a commitment—your load-shedding thresholds should be kicking in only if the load level starts exceeding this commitment.

Communicating load shedding to clients

Once the server has decided to shed load, the question becomes what you should tell the client. As I discussed in Chapter 4, when a client gets an error from a server, it's important that the client is able to distinguish between errors that should be retried and those that shouldn't.

If using HTTP, for example, you might be tempted to respond with a generic `503 Service Unavailable` response. However, this is a somewhat generic error, and it may not be clear that you want the client to back off. It would be detrimental for us to reject a request as we are load shedding, only to have the client just retry the request.

In fact, a dedicated HTTP status code, `429 Too Many Requests`, was added explicitly as an indicator that the service was load shedding. The use of a `4XX` error code for this is interesting—remember that these response codes are broadly considered client-side errors and are used when an HTTP server is telling a client that the client has done something wrong. In this situation, it seems to make sense; the client is being admonished that too many requests are being sent to the server. Additionally, as with other `4XX` error codes, the server wouldn't expect the client to retry the request (although actually you might—I'll come back to the use of the `429` later in this chapter when we look at backpressure).

Of course, telling a client "this work was dropped—please don't retry it!" is problematic for things that really do need to happen. Some work is important and needs to be carried out. In situations where we just reject the

work, we then leave the client with a puzzle about how to make forward progress.

If you can take the concept of better client communication one step further and give more information to the client, then it's possible to help in situations like this. Later in this chapter, I'll cover the concept of accord-based backpressure, which requires the client and server to work together.

Setting where to trigger

In some cases, you may mostly be concerned with clients speaking to a single server. In such a situation, this single server would be the point at which the amount of work is tracked but also where load shedding would be applied. However, I suspect many (if not most) of you are working in a services environment, in which a client talks to a logical service, which could in fact be deployed onto multiple servers. In Figure 6-1, you can see a service with multiple instances, with the client making calls to the service's API via a load balancer.

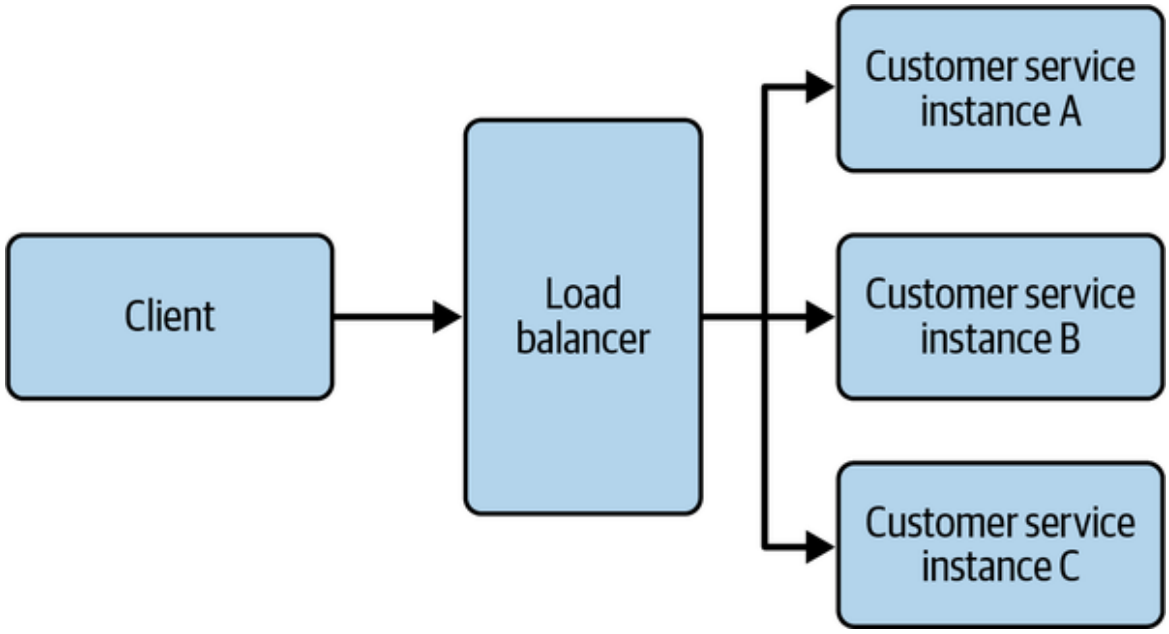


Figure 6-1. In a services environment, a service is often deployed on multiple instances

This raises the question of where you should be tracking your thresholds for load shedding. Do you set limits for the service as a whole? For example, you might say that the entire service can handle only three hundred

concurrent requests, at which point load shedding should commence. This creates the opportunity for load shedding to be handled by the mechanism that distributes load in the first place. Many load balancers allow you to set an upper limit on the number of requests that can be allowed. If you were using a message broker for load distribution, setting a maximum queue size would have a similar effect.

If you do try to manage load shedding on the service instance level of a service, then you need to understand the implications. For example, in a load balancer setup, if one of the instances starts load shedding, these errors may be detected by the load balancer itself, which may potentially result in the instance being removed from the load balancer pool. This could result in a significant loss of capacity for the service as a whole and could even lead to a cascading failure, like the one we explored in Chapter 5. We'll look at load balancers in more detail in Chapter 8.

Handling inconsistent load from work

Tracking a total number of work items can be problematic in situations where the load generated by different work items varies drastically. Imagine, for example, that a client can send two different calls to a `Customer` service instance. One of these calls is to create a new customer, which represents a pretty key piece of functionality in the system. The `Customer` service also handles *subject access requests* (SARs), which are required for systems that operate under the auspices of the GDPR.

A SAR represents a request from a user for the information we hold on them. Creating a new customer doesn't generate too much load, but processing a SAR might be more intensive, especially if it involves interaction with a number of other services. In such a situation, an instance of your `Customer` service might be quite happy handling 100 customer creations but might struggle with handling 50 SARs.

In Figure 6-2, we can see that although each instance is handling the same number of requests, one of the instances is suffering more, as it just happens to have more SAR work items to process.

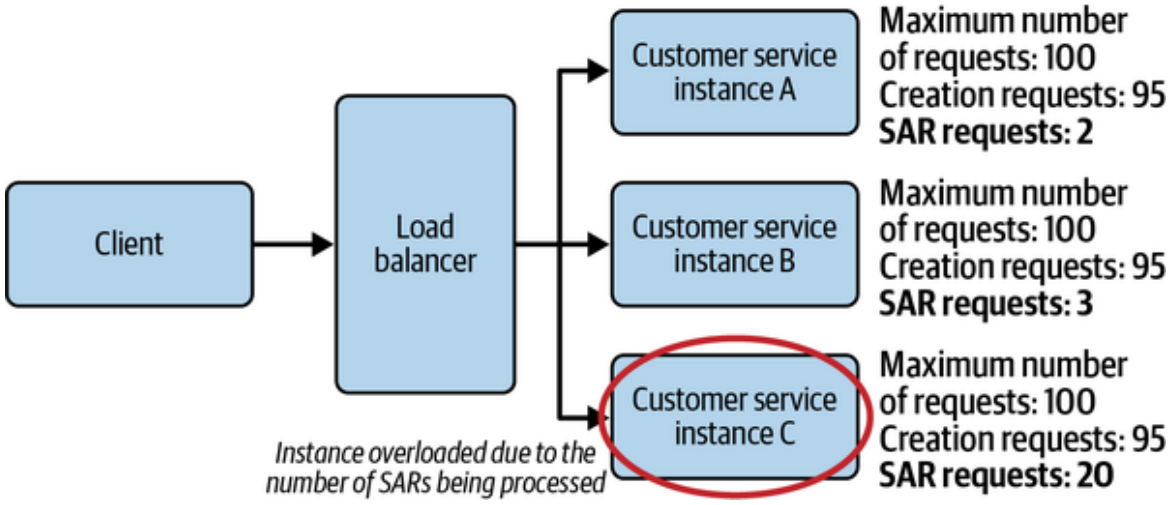


Figure 6-2. The number of requests might be evenly balanced, but some requests are more expensive than others

So, in this scenario, tracking a general number of concurrent requests is going to be tricky. This raises the prospect of load shedding using actual underlying performance metrics, with all the caveats we explored earlier. It may also cause you to push load shedding down to a service instance level, where the given instance could better understand how much work it is doing. However, there may be better approaches to consider.

Prioritizing work

In situations where different work may cause different amounts of load, or where some work is more important than other work, you may want to have some mechanism for prioritizing how workload gets shed.

Coming back to our scenario with the `Customer` service, I showed you that two types of requests need to be dealt with—creating a new customer, and dealing with SARs. Creating a new customer is a key business process that is critical to the system. Getting a new customer created is important to grow the business, and you'd want to do this quickly, so that your new customer can actually start using your software (and maybe paying you money). Just rejecting new customer sign-ups might not make a good first impression.

On the other hand, while you might have a regulatory requirement to process a SAR, you have a long timescale in which to do so—in fact, you

have up to a month. So in a situation where `Customer` is under load, having a client still allow new customer creations while throttling SARs may make a lot of sense, assuming you have a mechanism to ensure that those SARs get processed later.

In Figure 6-3, we see the `Customer` service with two REST resources exposed over HTTP for the different operations. To create a new customer, a client sends an HTTP POST to `/customer/`. When a SAR is required, the client POSTs a request to `/sar/`. In this case, you can clearly trace the entry points of the two types of requests, and this could allow you to tune different load shedding for each entry point.

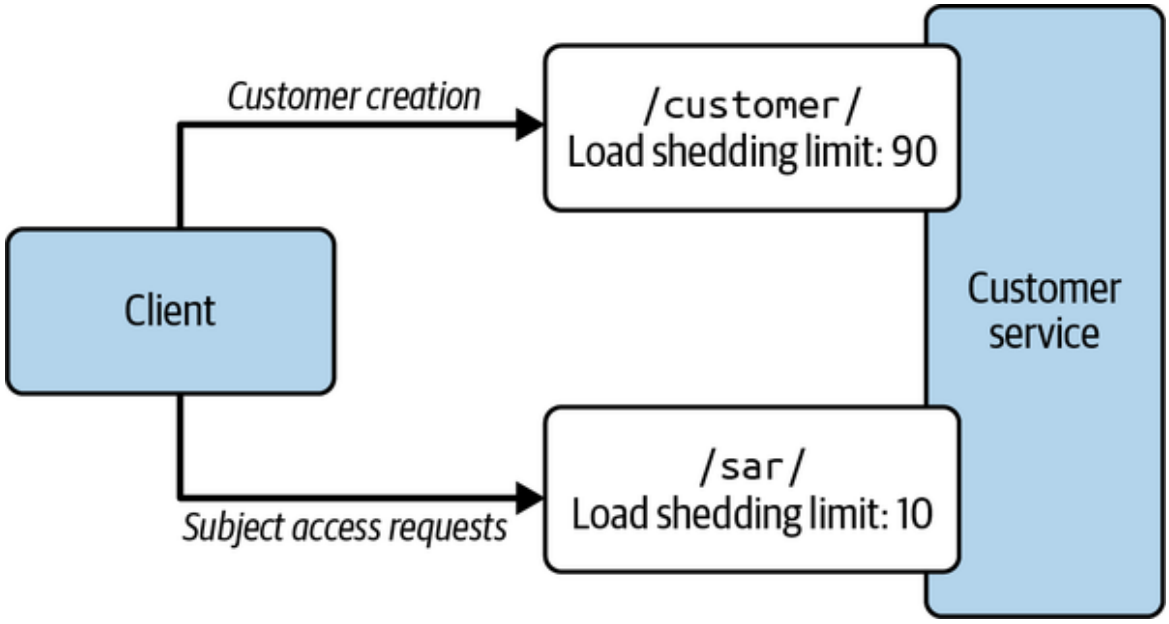


Figure 6-3. Setting different load-shedding thresholds for different endpoints

Another approach would be to have two pools of `Customer` service, one for each type of work, and route the calls appropriately.

If your service was consuming work from a queue instead, you could have two queues for the different types of work. That would allow you to specify queue depth limits for each type of work in a very transparent way.

In practice, though, I see this as working around a more fundamental issue that may need to be addressed: is it OK for one service to be doing two very different types of work? The more different types of things a service does,

the more challenging it can be to manage the service, especially at scale. So when you have a service carrying out a mix of work with different types of load profile and complexity, this could be a sign that you should consider splitting the service apart.

For example, if you split out the SAR functionality from the `Customer` service, you'd end up with two services with different load profiles and differing levels of criticality. This could simplify a number of things—for example, our availability SLOs for the new `Subject Access Request` service might be much lower than those for the `Customer` service.

The existing `Customer` service could remain focused on dealing quickly with new customer requests over HTTP. On the other hand, the SAR processor could follow a model in which the SARs are sent to a queue, which is then processed in order at a defined rate, as we see in Figure 6-4. (I'll explore why queues might be a good fit for this type of problem space in Chapter 9.)

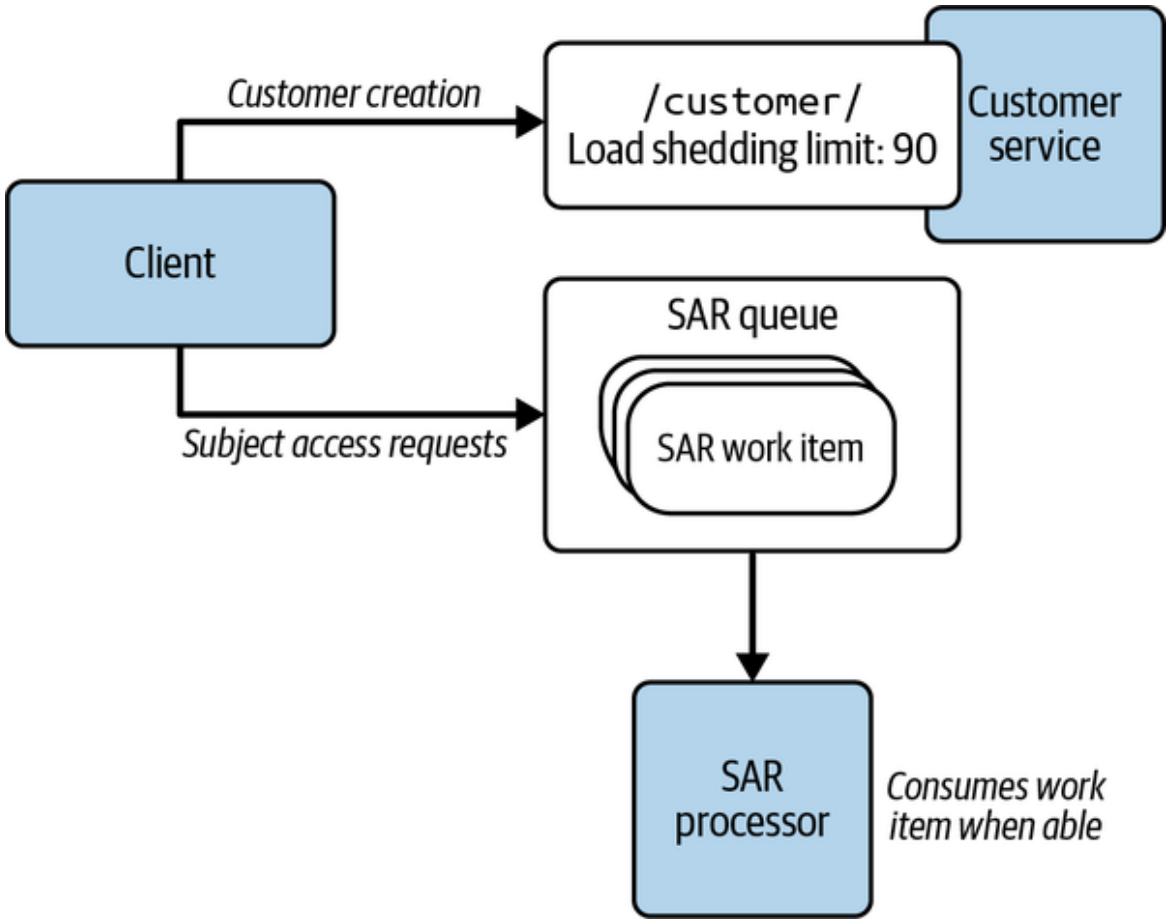


Figure 6-4. Splitting the `Customer` service due to differing load profiles

Once the service is split, you also have the ability to run the different services on different infrastructure as appropriate to their requirements.

The decision to create a new service comes with a cost, of course. This is another component that needs to be maintained, and it adds further complexity to the system. This brings us back to the paradox we explored in Chapter 1: sometimes the solutions we use to improve the resiliency of a system can increase complexity—and increased complexity can lead to new sources of failure.

The decision about when to split a service isn't always clear-cut, but if you want to explore this in more detail, I have a book on the topic.[1]

When to Use It

Load shedding should be used in any situation where you believe that your service may be overloaded, and an uncontrolled shutdown caused by an overload would be unacceptable.

To keep things simple with load shedding, start with an upper bound on the number of concurrent requests that you are happy for the service to handle and track that. If possible, implement load shedding at the load balancer or similar, when it is generally easier to do so, and clearly differentiate load-shedding errors from other error types.

LOAD SHEDDING AND KUBERNETES

The use of Kubernetes raises some interesting questions when it comes to load shedding, specifically around where the load shedding is done. Traffic that is routed externally into a Kubernetes cluster has to go via an ingress controller. There are a variety of ingress controllers available, many of which provide load-shedding capability and other rate-limiting controls. However, this allows for the load shedding to be done only at the main entry point into a Kubernetes cluster and does nothing for communication between services inside the cluster.

Internally, Kubernetes allows you to configure network policies that describe how interpod communication is done. This includes how traffic is distributed across nodes in a service. However, these traffic policies do not extend to providing any rate-limiting capability.

This means that if you want to provide service-level rate limiting, Kubernetes out of the box doesn't do anything to help you. You could of course make use of a sidecar proxy such as Envoy (perhaps as part of a service mesh) to apply rate limiting, or else implement the appropriate logic inside your service implementation itself. Using a sidecar proxy may actually be the way to go if making use of HTTP/2 or above, as tools like Envoy are able to see application-level data in the network traffic, meaning they can better account for multiple request-response flows operating over the same HTTP connection.

Another option would be to consider using a Kubernetes overlay network, such as Cilium or Calico, both of which allow you to configure rate limiting for services running inside Kubernetes itself.

Backpressure

In a situation where a service is overloaded to the point that it is struggling and perhaps is even starting to shed load, we ideally should have some way to reduce or even stop the flow of work. More generally, the term

backpressure is used to describe applying some sort of resistance to the flow of fluid through pipes. This concept maps over quite nicely to distributed systems, where the term refers to our ability to reduce the flow of work.

Backpressure can be implemented in a number of ways. A client can decide to trigger backpressure arbitrarily, in reaction to increased error rates—I'll refer to this as *client-only* backpressure. The client can also choose to implement backpressure as a result of information being received from the server itself; this approach can be more effective, but it must be built into the protocol used for the client and server to communicate. I describe this as *accord-based* backpressure. Let's look at client-only backpressure first.

Client-Only Backpressure

With client-only backpressure, a client decides to apply backpressure based on its own information. This decision could be based on a manual operator's intervention, or on observation of the success (or failure) of calls the client has been sending to the server. A threshold is reached, and the client stops sending traffic. A circuit breaker is a common pattern used to implement client-only backpressure, and I'll explore that more shortly.

Aside from the generic benefits that backpressure brings in terms of improving system stability, client-only backpressure is often a popular choice because it can be somewhat easily retrofitted into existing interprocess communication. The protocol between client and server doesn't need to change.

The main downside of client-only backpressure is that backpressure is being applied locally. In a service-based architecture, it's common for each server to have multiple clients. If one client decides to stop sending calls to apply backpressure, that doesn't mean other clients will have reached the same conclusion (unless you have some mechanism for clients to share this information). As a result, you may end up relieving less pressure on the server than you would have hoped.

For client-only backpressure to trigger, the client must already be seeing errors. This means that client-only backpressure be applied only reactively rather than proactively.

Another issue with client-only backpressure is that you lack information on how much backpressure to apply. Decisions such as how much to reduce calls by or how long a circuit breaker should remain open are ones the client is reaching by itself. The client is relying on local information rather than on any wider understanding of what is happening at the server. This, of course, is where accord-based backpressure comes in.

Accord-Based Backpressure

With accord-based backpressure, the server provides information about the backpressure that is required, which the client then acts on. Put another way, the server is agreeing to send some information asking the client to back off, and the client is agreeing to listen.

For accord-based backpressure to work, we need to build this information into the client-server communication protocol. A good example of a real-world use case would be the HTTP `429 Too Many Requests` status code, which I touched on previously. When sending back a `429`, the server can include an optional `Retry-After` header, which tells the client how long it should wait before sending additional requests.

gRPC's equivalent error code would be `GRPC_STATUS_RESOURCE_EXHAUSTED`. Unlike in the case of HTTP, there is no defined equivalent of the `Retry-After` field in the gRPC specification. As a result, you'd need to define for yourself what information needs to be relayed back to the client to trigger the backpressure. In addition, gRPC also supports custom backend metrics; this allows for more detailed information to be sent either within the context of the call or out of band and is primarily used to help gRPC clients make load-balancing decisions. I'll look at load balancing in more detail in Chapter 8.

In situations where multiple clients are involved, a server's ability to provide backpressure hints means that we are likely to reduce backpressure more quickly, as we're not waiting for all the clients to reach their own thresholds. This means the backpressure can be applied before a client even starts seeing an increase in error rates.

Note that with accord-based backpressure, a server could decide to start applying backpressure for some types of work but not others. Returning to our previous example of the `Customer` service, if you were to prioritize creation of new customers over SAR-related work, `Customer` might start sending back `429`s for SARs first, before it considers doing the same for customer creation work.

Once a client has been informed that it should start applying backpressure, there is the question of how the client should actually implement it. One of the most common patterns for implementing backpressure in a client is the circuit breaker, which we'll look at next.

QUOTA LIMITS VERSUS LOAD SHEDDING/BACKPRESSURE

Both gRPC and HTTP define standard error codes to be used when load shedding or backpressure is applied (`GRPC_STATUS_RESOURCE_EXHAUSTED` and `429 Too Many Requests`, respectively). However, these codes can also be used when an individual client has reached a quota limit.

Especially when exposing APIs to third parties, it is common to limit the number of calls that a specific client can make. This ensures that each client is getting a fair share of the server's time—or, in situations where API access is monetized, that the client is allowed to make the number of calls it paid for and no more.

From a client's point of view, at the moment the error is received, it doesn't much matter why the error was sent back—if you get a `429`, you're being told you sent too many requests and should stop.

From a service's point of view, a single client being rate limited for exceeding its quota is quite different from a service being overloaded. If one client breaches its quota, then that client gets rate limited. But if a service is close to its breaking point, then all clients will end up getting rate limited. However, per-client quotas can make a lot of sense for dealing with badly behaved clients who could end up becoming a "noisy neighbor"—a participant in the system whose usage of the system ends up adversely impacting others.

Because of this, one client getting a `429` cannot be extrapolated to the service itself having an issue. This means that if you were trying to diagnose a failure on the client side, it might be difficult to understand why a specific client got rate limited. A simple message in the `429` response body could help make things clearer to a human operator wanting to understand what has happened. If you wanted a client to programmatically handle a generic load-shedding scenario versus a quota issue differently, then having an easily parsable field in the

response (perhaps a custom header) could also be an option. Such a field could also then make its way into your telemetry, allowing you to distinguish easily between quota limits and more general rate limiting in your observability tooling.

Pattern: Circuit Breaker

Breaks the connection from a client to a server when an error condition is identified in the server, allowing subsequent calls from the client to fail fast.

In your own home, circuit breakers exist to protect your electrical devices from spikes in power. If a spike occurs, the circuit breaker switches into an open state, breaking the circuit and therefore protecting your expensive home appliances. You can also manually open a circuit breaker to cut the power to part of your home, allowing you to work safely on the electrical wiring. In a pattern I first learned about from Mike Nygard's book *Release It!*, 2nd edition,[2] we can implement a similar mechanism in our client-side software as a way of implementing client-only backpressure.

How It Works

With a circuit breaker, after a certain number of requests to the downstream resource have failed, the circuit breaker is switched into an "open" state. Any requests routed to a circuit breaker that is in an open state will fail fast, as shown in Figure 6-5. The terminology of an "open" breaker, meaning requests can't flow, can be confusing, but it comes from electrical circuits: when a breaker is "open," the circuit is broken and current cannot flow; closing the breaker allows the circuit to be completed, and for current to flow once again.

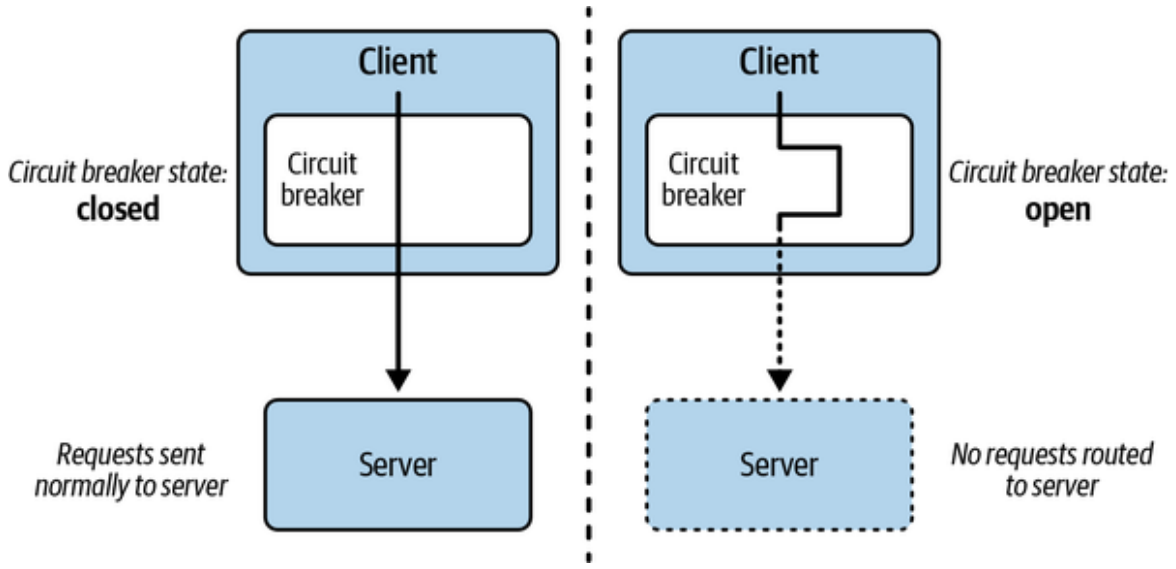


Figure 6-5. Circuit breakers

Once a circuit breaker has switched into an open state, we need a way to "close" it so that work can start flowing again. This can be done in a variety of ways, but a common approach is to still allow a few requests through even if the circuit breaker is open, and to close it if those requests succeed at an acceptable rate. Another option is to monitor some sort of health check endpoint on the remote server. When the server is considered healthy again, the circuit breaker is closed.

Getting the settings right can be a little tricky. You don't want to open the circuit breaker too readily, nor do you want to take too long for it to switch closed. Likewise, you really want to make sure that the service is healthy again before sending traffic.

Once you have a circuit breaker mechanism in place (as with the circuit breakers in your home), you can use them manually to make it safer to carry out maintenance work. For example, if you had to shut down a service for a period of time, you could manually open all the circuit breakers of the service's consumers so they fail fast while the service is offline. Once the service is back, you can close the circuit breakers, and everything should go back to normal. This could all be automated as part of a deployment process—although in general, moving to a model where services can be updated without causing a loss of availability is preferable.

Case study: AdvertCorp

In Chapter 3, I introduced the real-world example of AdvertCorp, which had suffered a major outage. As you may recall, we had a problem on that project with the Turnip application responding very slowly before eventually returning an error. One of the issues was that the timeouts were too generous, leading to resource contention, so fixing those was a priority. Once we did that, though, we realized we still had an issue. Even if we got the timeouts right, we'd be waiting for the timeout threshold to be reached before we received the error. During the failure mode we saw, the `Turnip` service had a fundamental issue—but we kept sending requests. It would have made much more sense to fail fast rather than failing slow.

To deal with this issue, we decided to wrap calls to all the legacy applications with circuit breakers, as Figure 6-6 shows. When these circuit breakers blew into the open state, we programmatically updated the website to show that we couldn't currently show ads for, say, turnips. We kept the rest of the website working, and clearly communicated to the customers that there was an issue restricted to one part of our product, all in a fully automated way.

We were able to scope our circuit breakers so that we had one for each of the downstream legacy systems. This lined up well with our decision to have different request worker pools for each service we were talking to. Our use of circuit breakers predated any available open source implementation,[3] so we just hand-rolled some code around our use of the HTTP connection library. If we'd thought to have open sourced our implementation, we could have been pioneers!

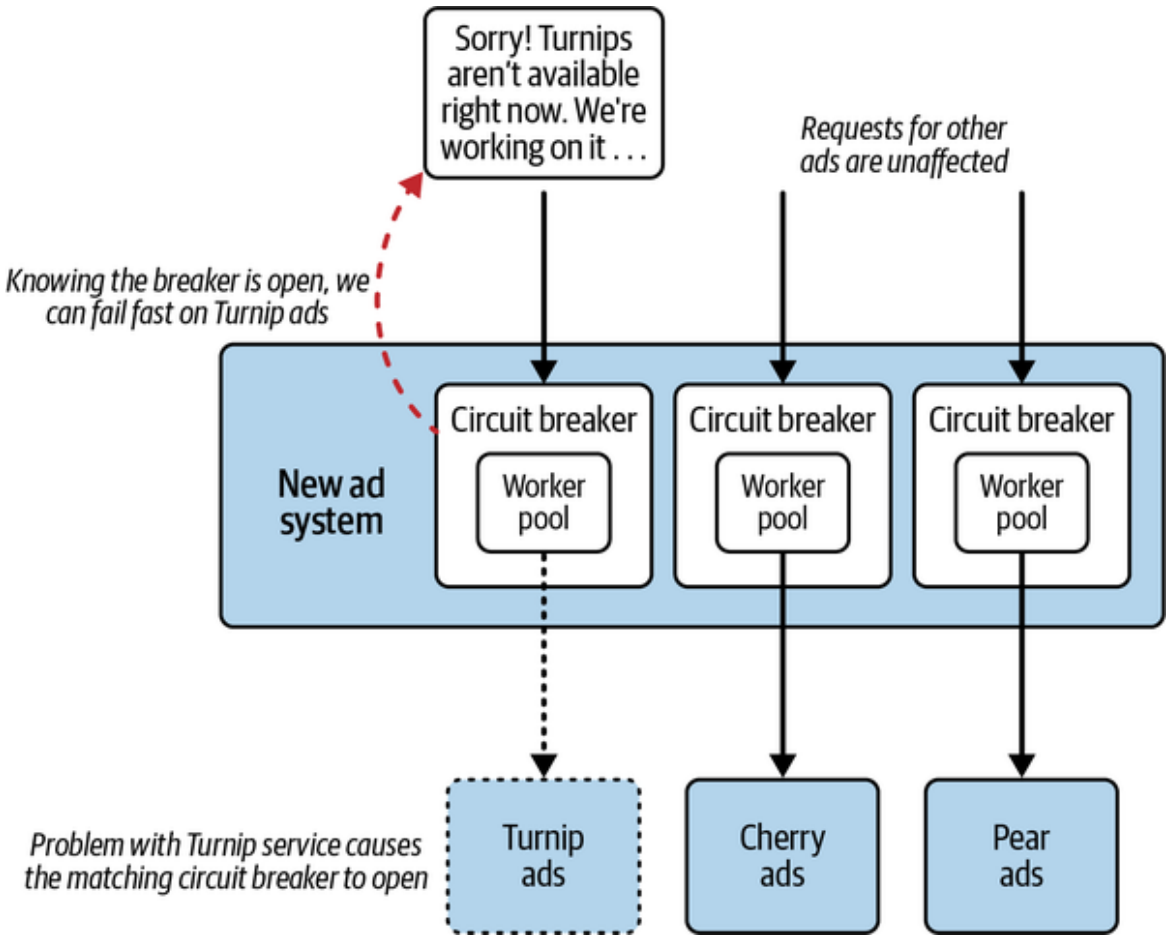


Figure 6-6. Adding circuit breakers to AdvertCorp

For client-only or accord-based backpressure

A typical circuit breaker implementation would provide client-only backpressure. On observing a certain number of call failures, a client would decide to open the breaker. This is the implementation of a circuit breaker that you'll typically see in connection libraries.

Theoretically, though, you can put a circuit breaker into an open state based on information from the server. Rather than waiting for a certain number of failures, you could implement a circuit breaker to trigger on a single `429`, using the `Retry-After` field to determine when the circuit breaker would reset to its closed state.

Issues with circuit breakers

On the face of it, the circuit breaker pattern seems like a straightforward and relatively simple way of implementing client-only backpressure. If you currently don't have any backpressure in your system, it's certainly a good place to start. However, you need to be aware of some downsides.

Too late to the party

When relying entirely on locally available information to set a circuit breaker state, circuit breakers have the same downside as any client-only backpressure mechanism: by the time the backpressure is triggered and the circuit breaker is opened, the server we're talking to is already having a problem. As such, our backpressure may not trigger quickly enough to maintain an acceptable degree of system stability.

Boom and bust

Each circuit breaker is an all-or-nothing affair. Either it's letting requests flow or it isn't. This can create something of a boom and bust pattern. Consider Figure 6-7, which shows the number of requests being sent to a server over a period of time. As the requests build up, the server is struggling to deal with them, causing the failure rate of requests to increase (perhaps via client-side timeouts and/or load shedding). This then causes the circuit breakers in the clients to open, resulting in no requests being sent.

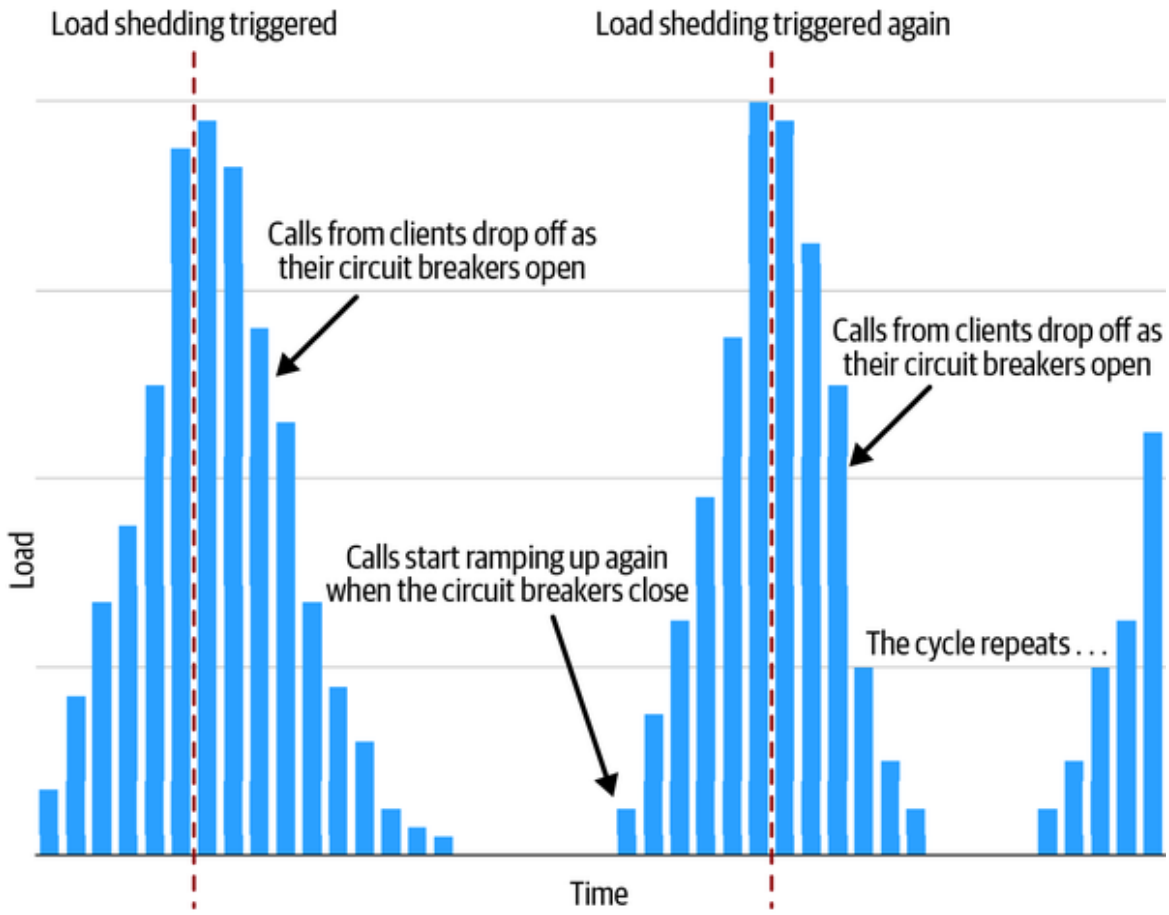


Figure 6-7. Circuit breakers opening and closing together can create a boom and bust cycle

After a period of time, the server recovers, the circuit breakers close, and the calls come flooding back in. In this example, the peaks are too great for the server, but the server is then idle during the troughs—not a great use of our computing resources. Things would be much better if our resource consumption were much more consistent.

Circuit breakers give us a great way to fail fast, which is always preferable to failing slowly. It means you free up computing resources more quickly, which helps reduce load on potentially stressed components and allows the system to carry out mitigating actions as soon as possible. But there is no nuance with a circuit breaker—either the client sends no traffic or it sends all the traffic.

Partial failure

Another problem is how well circuit breakers deal with a server that is partially failing, resulting in a situation where some types of work seem to be processed without issue but other types of work get rejected. Coming back to our example of the `Customer` service, imagine that you've decided to prioritize customer creation requests over SARs. The SAR limit is reached, so you start load shedding those requests.

In Figure 6-8, we see that a client sending calls to both customer creation and subject access requests has a single circuit breaker for the `Customer` service as a whole. If enough SARs fail, this could cause our circuit breaker to open—even if all the calls to create customers are working fine. In effect, the circuit breaker has made a partial failure worse.

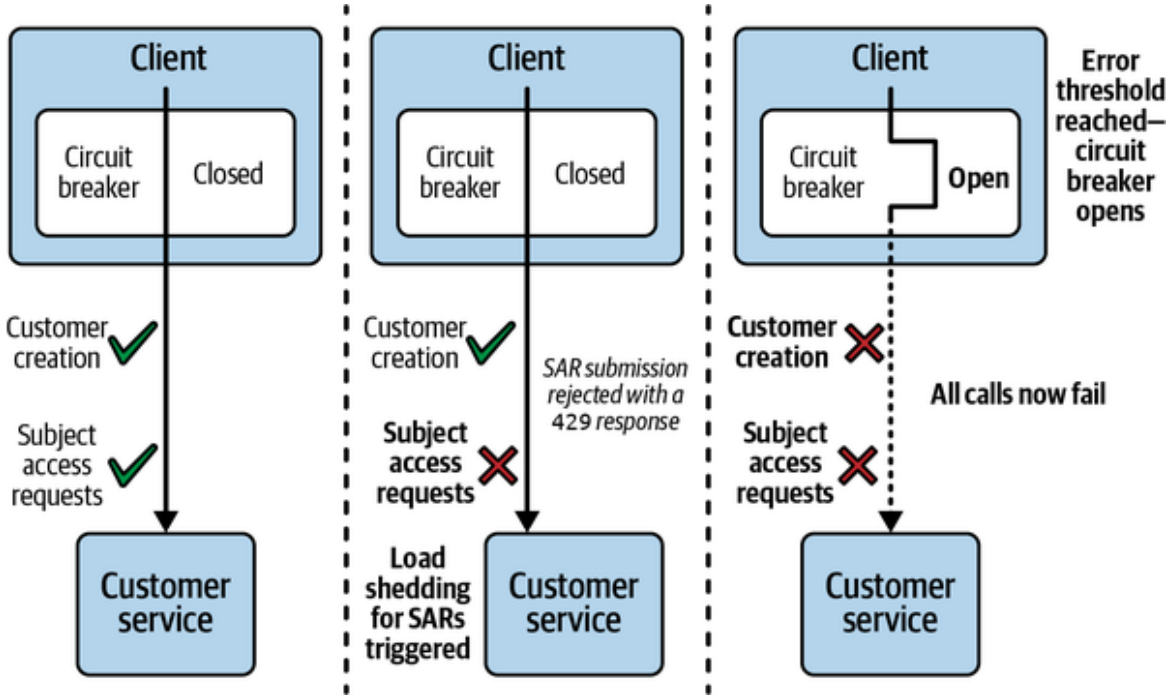


Figure 6-8. A partial failure of a service can cause the circuit breaker to trigger

There are a couple of ways to solve this. The first is to allow each initial request to go straight to the server and route only retries to the circuit breaker. This means you are always sending some requests to the server, and you can use the success (or failure) of the initial attempt as a way to determine the state of the circuit breaker itself. This approach is more of a partial mitigation than a total fix—it ensures that at least some customer

creation calls get through, but if one of them needs to be retried and the circuit breaker is open, the retry will be rejected.

Coming back to the original analogy, in an electrical system a circuit breaker reacts to an issue in a specific circuit and breaks that circuit. In the situation where some types of work are being processed successfully but other types of work are not, it's worth considering if we should actually have two circuits.

This leads to the second solution, which would be for your client to have two circuit breakers for talking to the same service—one for creation requests and the other for subject access requests, as we see in Figure 6-9.

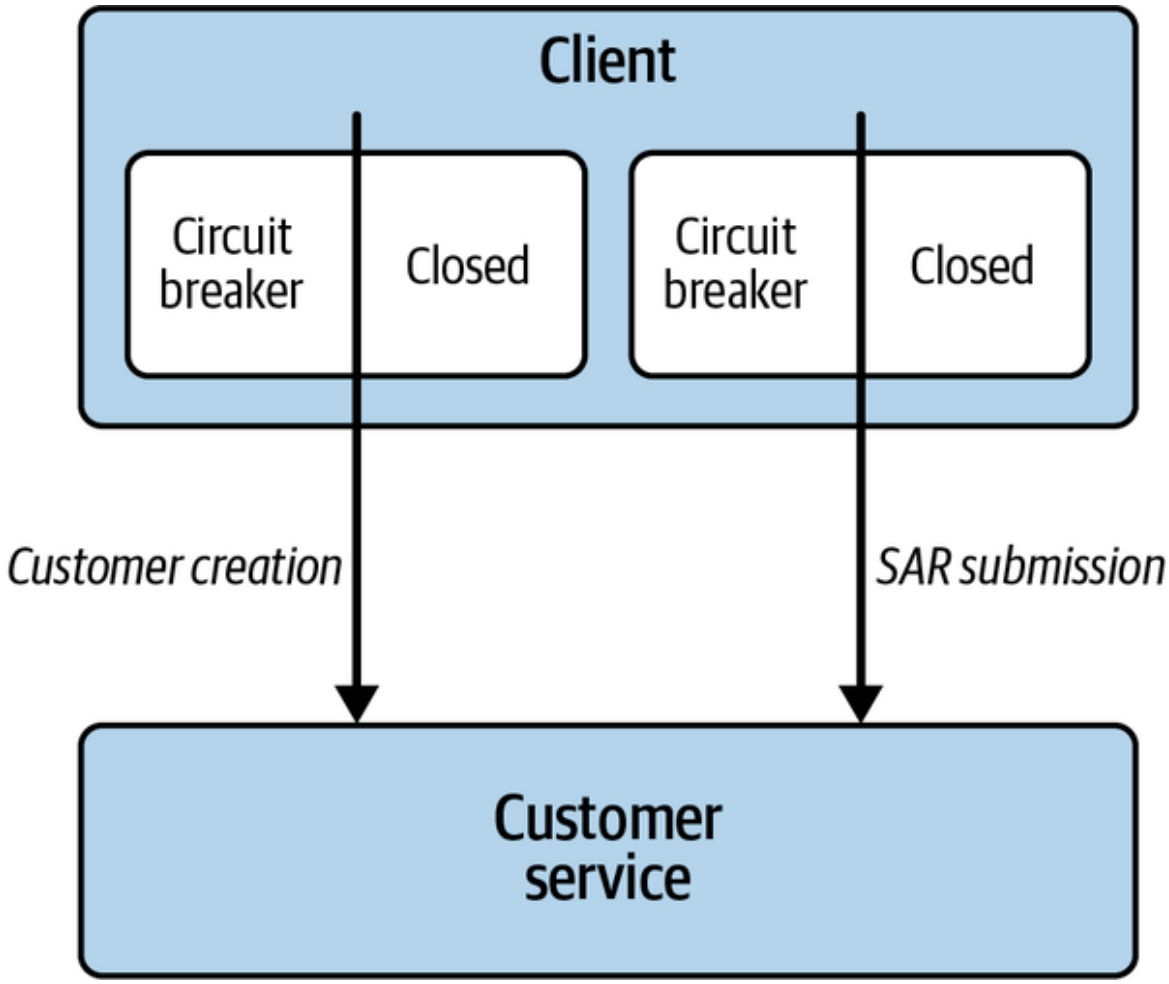


Figure 6-9. Splitting the different types of requests across two circuit breakers

If this solution looks appealing to you, I'd urge you to check what the underlying issues are here. Why is one set of requests failing, with others

working fine? It's possible that the types of work are so divergent that splitting the service apart might be a more sensible approach.

When to Use It

As you've seen in the preceding section, circuit breakers come with a host of downsides. As a backpressure mechanism, they are a blunt instrument. But they are a blunt instrument that is fairly simple to implement and can be a straightforward approach to allowing clients to fail fast. As a result, despite the drawbacks, I still think they are generally better than just using a simple retry policy by itself.

I would generally recommend against using a service mesh to implement circuit breaker logic. As I discussed previously in Chapter 4 when discussing where retries should be implemented, I feel that the developer should be very much aware of the health of the endpoints that their service depends on. This is partly due to the fact that your service may want to change its behavior based on the state of the circuit breaker (and this is difficult to do if the service mesh is handling this), but I also feel that the attempt to use the service mesh as a way of hiding the complexity of network-based communications is an abstraction too far.

When it comes to communication between services over message brokers, I am somewhat ambivalent about their use. Remember that in this context, all a circuit breaker is useful for is keeping track of the health of the communication between the client and the broker itself; it does nothing to help us regarding whether the message we gave to the broker ends up where we expect. Message broker client libraries already contain a host of smarts around connections to the broker, and assuming the broker is configured in such a way as to provide high availability (see Chapter 9 for more), I have not in the past felt the need for circuit breakers in that scenario. I might consider a circuit breaker when communicating with a message broker that had proved to be unreliable; however, at that point I would likely focus my efforts on understanding why I was having to rely on an unreliable message broker.

Reducing Versus Stopping Traffic

So far, we've mostly looked at traffic from the client being stopped entirely when the backpressure is applied—and this is exactly what a circuit breaker does for us. Just stopping all work being sent from the client is a simple and perhaps overly simplistic reaction to backpressure being applied. Let's look at some of the downsides of this approach and explore some alternatives.

Boom and Bust Cycle

If all clients simply stop all calls in relation to backpressure, this will have an immediate impact, but we have the issue I discussed previously with respect to circuit breakers. In Figure 6-10, we revisit our previous example showing the load over time for a service. When the load hits its threshold, backpressure is applied, which in this case results in clients ceasing all work.

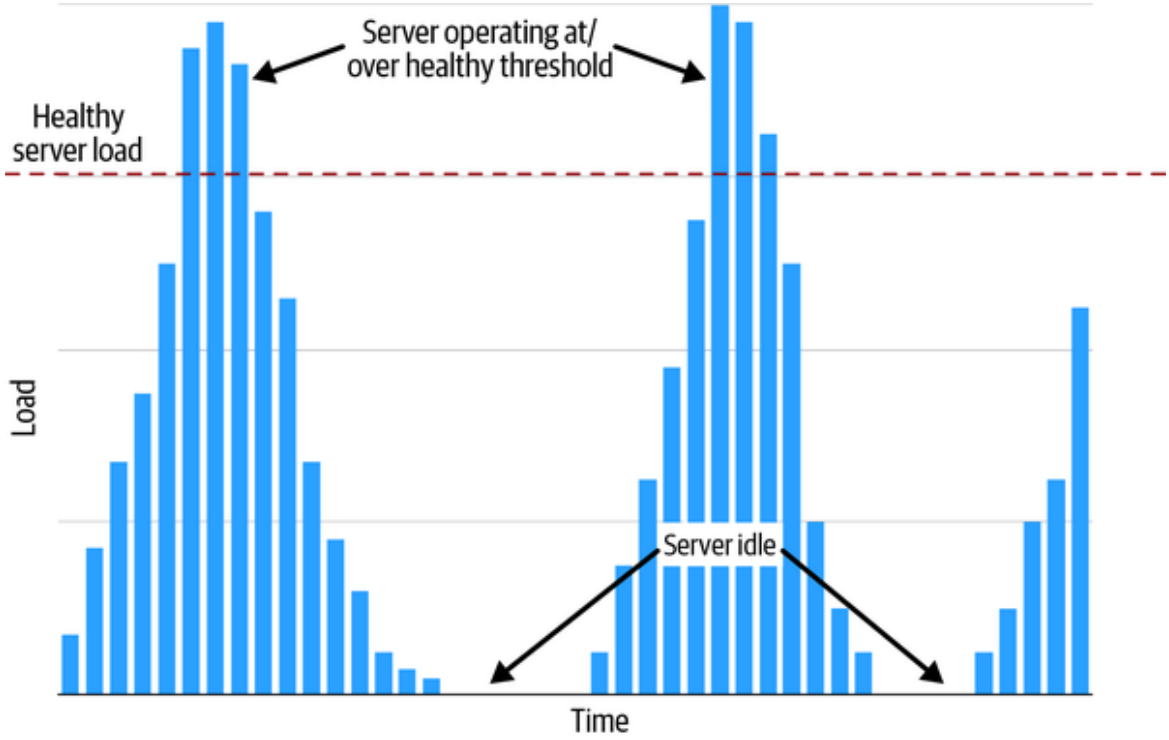


Figure 6-10. Totally stopping calls from clients can lead to large idle periods, followed by peaks when the backpressure is removed and the calls can flow easily again

Once backpressure is applied, you can see the load plummet. The service is then idle until the clients decide to start sending calls again. At this point, it's possible that there is still a lot of work the clients want to get done, so the calls come back, and the cycle is repeated.

In such a situation, throttling rather than stopping client-generated work could be beneficial. Rather than the server stopping all work, if some can be processed then there will be less of a backlog to handle when the backpressure reduces.

Prioritizing some work over other work can help resolve the issue of large peaks and troughs when backpressure is applied. For example, stopping all SARs might reduce the load on the server enough that normal customer creation can continue, as we see in Figure 6-11.

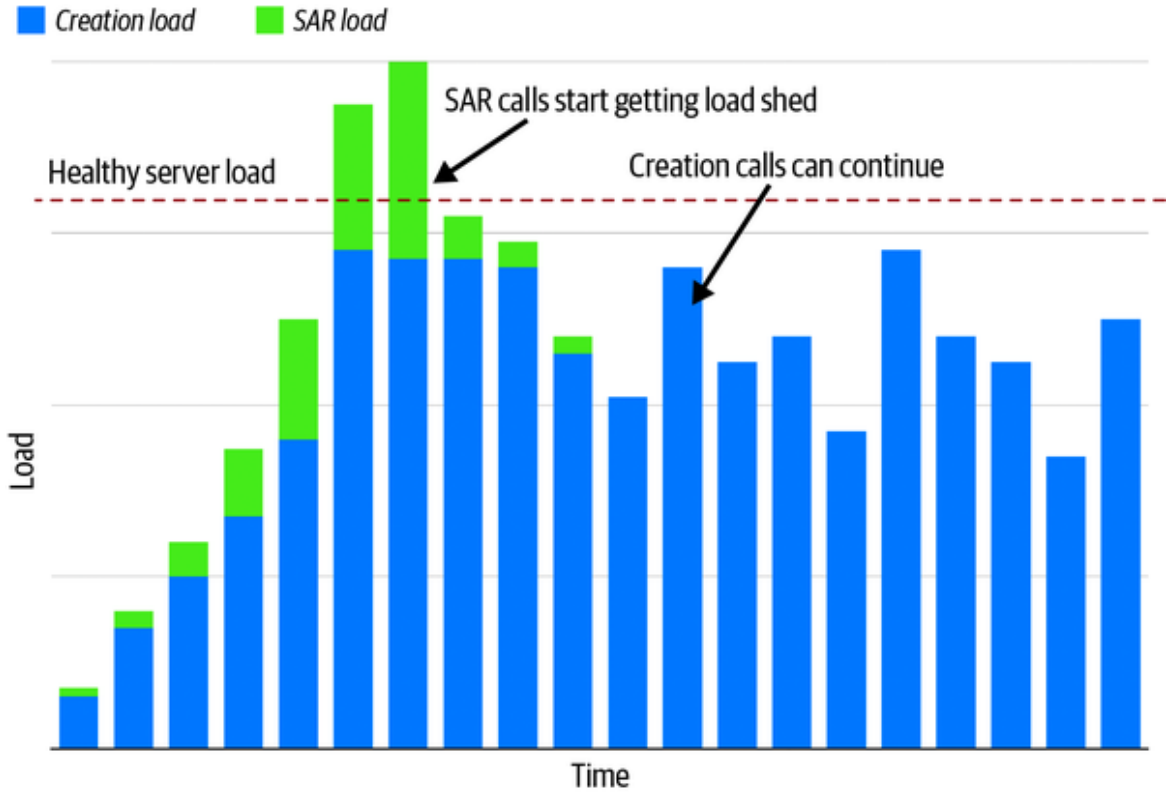


Figure 6-11. Dropping SAR calls may allow customer creation calls to continue to be processed

A Delicate Balance

One of the problems with reducing traffic rather than stopping it entirely is that if you still allow some work through, then in high-load situations you may not be reducing traffic enough to make a difference. Imagine that you have dropped the SAR-related work but are still allowing new customer creation. If at that point you are still over capacity, you'll have to start rate limiting customer creation calls as well.

The issue then is that it takes more time for the service to get to the point of equilibrium. Rate limiting more aggressively earlier would have made the service healthier earlier.

As you can see, it's not always clear which approach makes sense; this is why understanding your traffic patterns is so key to working out which rate-limiting technique is most appropriate.

Pattern: Leaky Bucket Rate Limiting

Ensures that the rate of calls from a client to a server can remain consistent to avoid overloading the server.

Rather than enduring the "Boom and Bust Cycle" we saw earlier in this chapter, it would have been preferable to reduce the service's traffic to the point where it was within healthy bounds, while also reducing or eliminating periods when the server sat idle. A smarter client can help here, with the help of a bucket.

Picture a bucket with a hole in the bottom. At a regular, predictable rate, water drips out of the bottom of the bucket. From time to time, you need to add water to the bucket. If the bucket has room, you can add more water. If it doesn't have room, you have to wait until enough water drains out to make space for the new water you want to add.

If you decrease the rate at which water is draining from the bucket, you decrease the rate at which water can be added. With leaky bucket rate limiting, we take this concept and apply it to the number of outbound calls our client can make.

How It Works

In Figure 6-12, when a client wants to make a request, it needs to add a token to the bucket. The bucket has a fixed amount of space for tokens, and these tokens leave the bucket at a steady rate. If there is room in the bucket, you can add the token. If there is no room, you have to either wait until there is or else just reject the request.

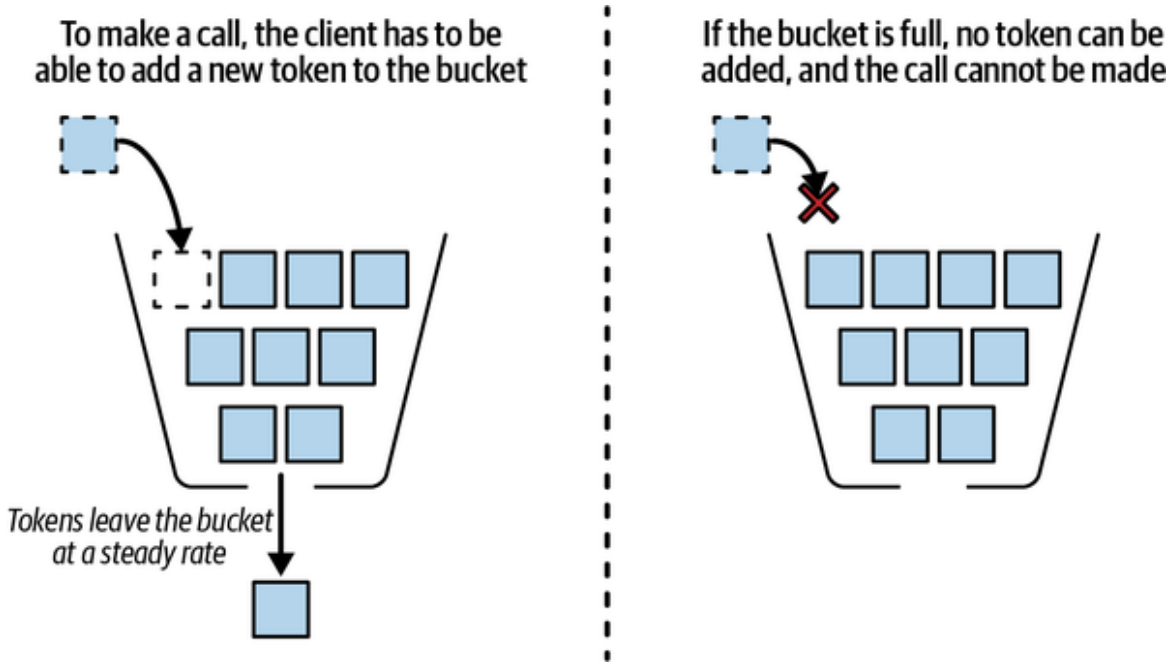


Figure 6-12. A client using leaky bucket rate limiting

This ensures that requests being made from your client will be set at a steady rate. By controlling the rate at which tokens drain from the bucket, you control the rate at which calls can be made.

Challenges

One of the complexities with a leaky or token bucket implementation is getting the configuration right. With a leaky bucket, you have two basic configuration values: the size of the bucket, and the rate at which tokens leave the bucket. Working out the most appropriate mix of values is going to take some careful work. If you have the ability to performance test your service, then that can help you play around with different values.

You can also use an appropriate simulator, such as Stability Sim, to help you try out different configurations, which I'll cover in more detail shortly.

Variations

A related algorithm is the *token bucket*, which is like the mirror image of the leaky bucket. To make a request, you have to remove a token from the bucket, and tokens are added at a regular rate. If a token isn't available, you can't make a call.

Personally, I prefer the imagery of a leaky bucket, so that's the one I've chosen to focus on for this book. Fundamentally, though, these two variations have a similar degree of complexity to implement, and they deliver the same results, so pick whichever variant you prefer.

Both the standard leaky bucket and token bucket algorithms provide static rate limiting from the client. The net result of this approach is a predictable upper bound in terms of the calls that can be made, with a simple mechanism to dial that number of requests up or down as appropriate.

Marc Brooker from AWS has discussed an adaptive retry mechanism using a token bucket. When a client makes the initial request, it is sent as normal. If the request succeeds, a partial token (e.g., 0.1 of a token) is added to the bucket. If a retry is required, that needs a full token from the bucket.

Brooker's simulations seem to show that this mechanism performs better at lower failure rates than a circuit breaker approach, with the potential downside of still creating some load when failure rates are high. Amazon uses adaptive retries extensively and has found that it reduces the extent to which retries can amplify load.

Existing implementations

There are a number of variations of the token/leaky bucket to be found. Resilience4j implements a version of the token bucket in its RateLimiter, and .NET 8 provides an implementation in `TokenBucketRateLimiter`. Variations of this algorithm are also found in the Transmission Control Protocol (TCP) and in message brokers as well.[4]

When to Use It

The leaky bucket rate-limiting pattern is a nice, elegant mechanism for making the flow of work more consistent from a client, and it avoids many of the pitfalls of the circuit breaker. However, it requires more work to tune properly.

In terms of a simple, easy-to-understand client-only backpressure mechanism, I can see a lot of value in starting with the circuit breaker and then looking at the use of a leaky bucket, either instead of or in conjunction with a circuit breaker, as you get more sophisticated.

Simulating the Use of a Circuit Breaker and a Token/Leaky Bucket

If you want a comparison of a circuit breaker and a token bucket in a simulated environment, Stability Sim lets you switch between token bucket, circuit breaker, and simple fixed retry semantics for clients. So you can set up a scenario and play around with some different options before deciding which one to use in your actual system.

One of the built-in examples, retry storm, is a good place to start working out how to use the tool; you can see a screenshot of this simulation in Figure 6-13. Give it a go yourself, and try switching the retry strategy of the web client to see how this quite significantly changes what happens when a server fails.

You can use this tool to simulate your own setup, and this might help you see whether a circuit breaker or a token bucket would be more suitable for your scenario. You could also use this tool to rapidly run through some different configuration options and zero in on a set that you want to explore further. A full run on Stability Sim can take a minute or less, letting you do a lot more experiments more quickly before moving on to a more time- and resource-intensive performance test, for example.



Figure 6-13. Stability Sim showing the simulation of a retry storm after a server failure

Config, Not Code

Reiterating a point from previous chapters, it's important that the configuration for your rate limiting is not hardcoded, but is stored in such a way that it can be changed without requiring a redeploy. This should apply for your load-shedding thresholds, token bucket and circuit breaker configurations, retry policies, and timeout settings.

You will often want to tweak these settings during both test and production. Having to do a full rebuild and redeploy to roll out a simple change in configuration will make tuning your system painful and will make dealing with production issues a nightmare!

> **TIP**
>
> Make sure your settings for retries, timeouts, circuit breakers, and token buckets are in configuration, not code!

Stopping the Load in the First Place

Much of what we've talked about so far covers either delaying load (by reducing the rate at which load arrives) or dropping load entirely (through load shedding). Delaying the work our system is doing works well in situations where load is peaky—so we have high peaks of load where our system looks like it is getting overwhelmed, interspersed with calmer periods. Delaying the load can help even out the work our system is doing, keeping resource saturation, latency, and error rate within acceptable bounds. Fundamentally, though, if we have long periods of load that strain our system, delaying or dropping load is going to be only a short-term fix.

As we discussed in Chapter 4, retries can be useful for making sure that operations you really want to happen actually happen. But an overly eager retry policy can end up being the source of high and sustained load. So make sure you check what portion of your load is first attempts versus second or subsequent attempts (retries). If a large portion is retries, this could be down to an overly eager client retry policy, or perhaps an increasing error rate in your service. Both of these would warrant further investigation.

If, however, the bulk of your load comes from first attempts, and delaying the load isn't helping, then you might need to look at other options. If the volume of work just represents work you actually want to process but can't, then scaling up the system would be a natural next step. Caching can often help here as well, as it can reduce the need to make a call in the first place.

But are there some nontechnical options for reducing the rate at which load arrives? This book is attempting to look holistically at a distributed system. Through that lens we should also consider whether other things can be done to reduce the load in the first place.

As I discussed in Chapter 5, if the large growth in user numbers is causing you load issues, consider techniques to control how quickly those numbers can grow, such as using invite-only betas or invite codes more generally.

If specific features are at risk of causing load issues, then consider rolling those features out to only small sets of your user base initially, to help you get a grip on how you can handle this new load pattern. This approach

would require you to design features in such a way that they can be enabled or disabled for specific user cohorts, but that is fairly standard practice nowadays.

It can also be useful to track load by user. If some users are generating a lot more load on the system than others, consider a fair use model in which high-load-generating users get their usage of your system throttled above a certain threshold.

Finally, consider pricing features based on their load profile. If certain features generate significant load on your system, then you would expect this to be reflected in what you charge your customers. Perhaps more load-heavy features are limited to more expensive pricing tiers, or perhaps those customers that use your system more are charged more proportionally to how they use the system.

Matching customer spend to actual resource usage will help in two ways. Firstly, the customers themselves may end up moderating their usage to control their own cost. Secondly, you can ensure that those customers who do use the system more will make more money for you, enabling you to then spend that additional income on more scale for your system.

Summary

Everything starts with knowing the limits of your server. At what point do you start having issues? How many items of work can you try to handle at once before the quality of service starts degrading? Without this information, it becomes difficult to know where to set the thresholds for things like load shedding or backpressure.

Your goal is almost always going to be about protecting the system as a whole rather than trying to handle every bit of work. With that in mind, load shedding is an excellent pattern to use to protect servers. Even if you are more interested in backpressure, it makes sense to start with load shedding first.

In situations where you can't control the client, or the clients cannot be given the smarts to apply backpressure, load shedding will be vital. Consider a public-facing website, where you're exposed to traffic from the wider internet—or a situation in which your clients are actually Internet of Things (IoT) devices, where rolling out changes in the way they work is not practical.

If you already have simple server-side load shedding in place and have the ability to change the behavior of both client and server, a sensible next step is to change clients to implement proper accord-based backpressure in order to allow them to apply smarter backpressure in concert with information coming from the server.

On the other hand, if you are unable to change the server but can change the client, then implementing client-only backpressure via either a circuit breaker or leaky/token bucket patterns would be the way to go.

Of course, if you can find ways to reduce the amount of load you are expecting to handle in the first place, that can be a good idea. Load shedding and backpressure just delay when the load has to be dealt with.

In Chapter 7, we're going to look at how AI can help—or hinder—our push for a more resilient system.

---

1 Sam Newman, *Monolith to Microservices* (O'Reilly, 2019).

2 Michael T. Nygard, *Release It!*, 2nd edition (Pragmatic Bookshelf, 2018).

3 Yes, I am that old.

4 RabbitMQ calls this "credit based flow control." It's an interesting variation, as the "credits" actually propagate across multiple parties.

Chapter 7. Resiliency and AI

Nobody knows anything.

—William Goldman

Throughout this book, I outline where AI can help (or hinder) your quest to build more resilient systems. In this context, I thought it would be useful to discuss some of the broader themes around the current state of AI. My goal with this chapter is to highlight those areas in which the technology we currently have available to us can deliver useful benefits, while also indicating areas of concern with respect to resiliency.

I also pull back to look more broadly at the current state of AI. With the huge changes that are impacting our industry right now, I wanted to look at some of the bigger-picture issues that are being hotly debated at the moment and provide some concrete guidance as to how you can attempt to safely navigate a very uncertain AI-enabled future.

IT'S NOT ALL LLMS

People have been working on AI in various forms for decades. Despite this, the majority of the current hype is focused specifically around one subset of the wider field—large language models (LLMs). It is worth remembering that LLMs aren't all that AI has to offer.

Neural networks are an established technique designed to model the human brain and are used in a broad range of fields, such as medical imaging, manufacturing, and banking. They have proved to be incredibly useful, and their use predates LLMs by many years. Unlike LLMs, neural nets, once trained, are deterministic. Neural networks are, like LLMs themselves, just one subset of the larger field of machine learning (ML).

I'm highlighting this not to downplay the benefits of LLMs but more to draw your attention to the fact that the current discussion around AI has become very reductive. LLMs have a specific set of trade-offs associated with them that are quite different from those associated with more general ML-based solutions like neural nets. While the hype is strong around LLMs and around generative AI (GenAI) more broadly, it's important to understand that a much wider body of knowledge will absolutely be part of the future direction of this field.

Using AI for Software Delivery and Operations

There are some pretty wild claims out there regarding what AI can do for us, and even more wild claims from vendors about what AI will be able to do for us "any day now—please just give us some more money for data centers!" I want this book to be practical, so I've attempted to look at a set of activities that can be important for building resilient systems and highlight where the AI available today can actually help.

Software Development

Let's start with the obvious. For technology that was created to generate text, generating code fits the LLM capabilities quite nicely. In fact, the main source of the angst that AI has brought to the IT industry more broadly has been its implications for the role of the software developer. Should AI write all our code? Do we need junior developers anymore? What happens when we lose all those skills? *Won't somebody think of the children?*

But the vast majority of software developers I know now use AI coding assistants as part of their routine software delivery and credit it for making them much more productive. Compared to the claims made by tool vendors, though, initial reports like those from Google's DORA group and DX show only modest improvement in code actually shipped to production—one theory being that the writing of code is quite a small part of what is involved in getting something into production. As the tools get better, and as our patterns for using these tools begin to improve, I'm sure we'll start to see more of an uplift in terms of delivered code.

With all that being said, how can AI help you write code that might in turn help you build more resilient systems? Well, firstly, you can provide context files to your coding agents to direct them to make use of certain libraries and patterns. These context files can be reused across a team or organization. For example, you could create an AI skill that defines that all HTTP communication needs to be wrapped with timeouts that can be configured in a certain way, and perhaps use a circuit breaker for all retry attempts. This would at least get some sensible defaults in place. Pushing this further, if your coding agent has access to the right data, you could have the configuration of timeout thresholds set to be based on the published SLO of the service you were talking to.

You could also have your coding agent create load tests to check resource use and behavior over load, or perhaps tests that simulate network jitter to check retry behavior. Rather than your having to search online for example code samples that may or may not fit your use case, with an AI coding

assistant at your beck and call, this can all be done for you, and it can be tailored to your needs as well.

Understanding Existing Code

AI can be incredibly useful in helping us understand existing code. Given a piece of code, an AI coding assistant can do a great job of explaining what the code is doing. More than once, I've found some cron script that is being run that seems important but also is pretty opaque as to what it actually does. I also vividly remember joining a project on which two of the most important methods were called `doDestinationZuluLand` and `doOperationTheWonderStuff`.[1] To the team that initially created that code, it was a fun inside joke that made sense, and they knew what these methods did. To a newcomer, it was baffling.

It's worth highlighting here that code that is easy for a human to read is likely also going to be easier for an AI agent to understand. AI will benefit hugely from having more human-readable code to work with. When you're instructing AI to make a change to an existing codebase, the vocabulary you use in your prompts will be used to make changes to the code; if there is a big disconnect between the terms you use and the terms used in the code, the AI is more likely to struggle to interpret what you want. Also be aware that your AI coding agent is likely going to copy the styles and practices of the codebase you have it working on. If said codebase is full of examples of practices you don't want to repeat, then your agent may need some additional guidance.

When you are trying to learn a new programming language, this ability to understand code is really helpful too. Taking useful snippets of code in some new programming language and getting your AI coding assistant to describe what is going on can be quite beneficial.

Interacting with Powerful Tools

Tools that have very powerful capabilities can sometimes be daunting to first-time users. As a result, I'm seeing more and more vendors integrating

AI to help users learn how to use their tools. As an example, observability tool Honeycomb has a powerful language for querying data. As a power user, I find that this can be a great way to slice and dice my data in order to visualize the information I want in many different forms.

Rather than having to learn the language, you can use a natural language query like "Show me the latency of all the `200` status codes in the last hour for the `Pet` service," and Honeycomb will generate the query that it thinks you should run. You can then run that query as is or tweak the parameters—all the while learning more about how to use the tool.

Automated Anomaly Detection

Fully automated anomaly detection for our production distributed systems has long been a holy grail. The idea is that an automated system could spot anomalous behavior in your production system, determine that something is wrong, and let you know that action is needed (or perhaps trigger some automated remediation mechanism).

Now to be clear, good old-fashioned (boring? conventional?) non-LLM ML techniques can absolutely be used to help analyze anomalies. With Honeycomb, for example, you can select some data that looks odd, and Honeycomb will automatically correlate this with lots of other data to help you understand what is going on. You might see a huge spike in traffic, select that period, and then be able to see that this is correlated with an increase in sales—in which case the traffic increase might be justified. The reality, though, is that as useful as these solutions are, they are primarily used as debugging tools. Once an operator has detected that something is wrong, they can use these tools to better understand what the data is indicating.

The dream is a fully automated anomaly detection system that is good enough to spot problematic anomalous patterns without any intervention and then do something about them. As discussed in Chapter 2, you can use observable data to define custom alerts. The ultimate goal of automated

anomaly detection is that you wouldn't need to do this, and the system itself would magically know what is wrong.

On the face of it, technology that is good at spotting things that look similar should also be good at spotting anomalies, and fundamentally LLMs are good at spotting similarities. The issue is that models need to be trained to see similarities. When training a model, vast amounts of information are used (books, web pages, images, perhaps video) to create training data. This data is key in training the model. It's only once we have that trained model that we can present new information to it to get answers (this process is called *inference*).

Now obviously, we (hopefully) have a lot of data available to us from our observability platform—information about traffic patterns, latency, error rate, saturation levels, and so on. Theoretically at least, it should be possible to train a model based on that data. Then, as new observability data is made available, we could run it against our trained model and find anomalies.

The problem with this idea is that it implies we are doing the training ourselves. This wouldn't be a turnkey vendor solution—someone isn't going to deliver a model pretrained on data that would be relevant to our system. And training models is a complex and potentially expensive undertaking. It would also require some high-quality data; if our system has poor observability, then we'll likely be feeding poor data into our training process. It also implies a constant training loop. The newly arriving production data that we will initially be using for inference will also need to become part of the training cycle.

LLMS—A BAD CHOICE?

Although LLMs might seem like a good choice for automated anomaly detection, their lack of ability to learn on the job and the challenges in training one's own model lead me to believe that by themselves they are fundamentally unsuitable for this particular problem.

Incident Response

One area in which AI-infused tools are already proving useful is around incident response and post-incident analysis. A number of vendors, including Rootly, Resolve.ai, and incident.io, now provide AI assistants to help operators understand production issues as they arise, and these assistants can also aid in the creation of postmortem documentation.

For example, incident.io's AI SRE agent can pull together a bunch of useful information when an incident is triggered.[2] The agent will join a dedicated channel for the incident, provide information around the alert, and start gathering potentially useful related information. It will look for recent commits and mentions of this error in other channels and will also help you dive deeper into your system telemetry to try and understand what has happened.

For tools like incident.io's AI SRE to work, they need access to information. This means that they will need broad access to a lot of valuable data if you want them to be effective. If the agent can't see your telemetry data, then it can't slice and dice that data for you. If it can't look at your source code repository, then it can't spot recent commits that may have caused the issue. This does mean that you need a high degree of trust in the vendor creating and running the agent.

For tools in this space, it becomes vital that the human operator can get to the raw information used by the agent to summarize the situation and be able to understand the reasoning behind any recommended actions. This becomes important not only from the point of view of learning to trust the agent but also in being able to spot where poor data might be leading the agent to deliver suboptimal suggestions.

These tools can also be useful in the wake of an incident. Post-incident response can be a lot of work, and much of it should be done by humans. The process of understanding why a problem occurs helps the operators of a system learn and in turn better understand what changes might need to be made to avoid such a problem in the future. Having discussions with users of the system and communicating clearly to impacted stakeholders again benefits hugely from the human touch. And in the wake of a nasty incident,

there can be an element of catharsis—closing the loop on a stressful situation, and moving on with renewed confidence.

However, there is a lot of legwork involved in post-incident response, especially when it comes to pulling together the information required for a postmortem. Manually building out a detailed timeline of "what happened when" could take hours, whereas the right sort of AI agent, such as the ones that incident.io and Rootly provide, could do this for you in minutes. Quickly building an initial draft of a postmortem that the team can then refine before publishing the results can lead to much faster response. It's worth noting, though, that while LLM-powered AI is great at generating text, that text is worthless if it's never read. Make sure your incident response–assisting AI helper isn't generating reams of documentation that never get looked at!

Resiliency of AI-Powered Software

We've looked at how AI is changing how we build and operate our software. But we are also integrating AI into the products we build, using it to hopefully improve our software for our users. What new challenges does this create for ensuring our systems are resilient?

Model Provider Resiliency

Foundation model providers OpenAI and Anthropic don't have a great track record in terms of providing high levels of availability for their services, at least when compared to other managed service providers, like public cloud vendors. While I'm sure this will improve over time, having a potentially flaky third party in the mix of your critical functionality can be an issue.

A solution here is to consider ensuring that your software can still operate even if the model is unavailable. This could be achieved by limiting functionality in the case of the model being unavailable (or you getting rate limited or running out of tokens!)—this is an example of the graceful degradation pattern, which we'll look at more in Chapter 11.

Another option is to have backup vendors for a given model to hedge against your normal model provider having availability issues. For example, you can use OpenAI models on Azure, a handy alternative if accessing the models via OpenAI's own services proves to be too unstable, and in a similar way Anthropic's models can also be accessed via Google's Gemini Enterprise Agent Platform. When using backup vendors like this, there may be a lag in terms of how quickly the latest models may be available, but given that you may not always want to be on the bleeding edge of the models you use to embed in your software (something we'll discuss in just a moment), this is probably fine.

Model Changes

Consistency of behavior plays a big role in helping us ensure that our system is behaving appropriately. While dealing with the unexpected is a key concept in resiliency (or "graceful extensibility," as David Woods put it), that doesn't mean we want to build software where surprise is constant. Quite aside from the fact that LLMs are nondeterministic, small changes in models can also lead to quite different behavior.

I've seen reports where a model has started behaving differently, without any major change in behavior being flagged by the vendor. Additionally, moving to a newer version of a model can fundamentally change how your prompts may be handled.

It is therefore vital to have a good set of tests to validate that the AI portions of your code are behaving appropriately—and this should be a mix of traditional functional tests, together with the use of AI evaluation tools (evals). You want to make sure that when the unexpected occurs, you are ones who get surprised first, rather than your users.

I can recommend the second edition of Gayathri Mohan's *Full Stack Testing*,[3] for those of you wanting to explore testing in general, including verification of AI-based systems. For a deeper dive specifically into evals, I'd suggest *Evals for AI Engineers* by Shreya Shankar and Hamel Husain.[4]

Observability

The telemetry we receive from our own software and from third-party services alike is key to making sure our applications are working the way that we expect. As we discussed back in Chapter 3, we prefer a small variation of latency rather than a broad variation of latency. If the time it takes to successfully complete an operation varies wildly, it can be difficult to know where we should start triggering timeouts, for example. It also becomes more difficult to define latency-related SLOs.

Once you send a prompt to an AI model or instruct an agent to do something for you, the latency for the operation can vary greatly, and in ways that can be difficult to predict from the outset. Some of this can be put down to variability in the performance of the provider at a given point in time, but it can also be the result of rate limiting by the provider, or your prompt could just be taking longer to process.

In situations where the prompts sent to models come directly from your users, then you will likely have even less certainty about how long it should take for a model to come back to you. Has a call hung, or is it expecting a large output?

Some techniques to solve this can include dynamically setting timeouts based on the number of tokens being generated, or monitoring the streaming output from a model so you can track when the response starts coming back rather than waiting for the full response.

AI Challenges

> *There is no such uncertainty as a sure thing.*
>
> —Robert Burns

It's not just our AI chatbots that like to speak with often unfounded confidence. We live in a time when ever more people feel the need to speak with absolute certainty about what they think is right. Having strong opinions, weakly held, seems to be very much out of fashion.

The reality is that while the emergence of GenAI promises to be transformative, we can't yet work out what, exactly, everything is *transforming into*. If we really are living through a change that will have impact on the world for generations to come, then we have to remember that it was only in the years following similar shifts (such as the emergence of movable type, the invention of the telegram, the Industrial Revolution, or the invention of the internet) that the true implications of what was happening really became clear.

Uncertainty about the role of AI in our lives in fact exists across multiple axes, many of which may impact our ability to build resilient systems both now and in the future. But the technology that is promising to change our world also has a number of issues that can often go overlooked. To better understand the role that AI can play in improving your system's resilience, you need to have a firm grasp on its shortcomings and on the wider uncertainty in this space.

We've already touched on a few specific issues, but I wanted to look at some of the more general issues around AI that might give us pause for thought. So let's look at the various sources of uncertainty, confusion, and risk in the AI landscape before we look at some things you can do to navigate your way through this chaotic time.

Nondeterminism

We are used to our programs being predictable. Given the same input, we expect the same output. Computers are supposed to be good at that, right? Unfortunately, because of the nature of LLMs, they are nondeterministic. This means that giving LLMs the exact same input doesn't mean you always get the same output.

This is problematic in situations where you want to test an LLM-driven solution. When writing automated functional tests, we expect to be able to give a specific input and then the test passes if we get a specific output. LLMs don't work like that, so testing them becomes problematic.

This is also an issue in situations where we need predictability. So much of understanding whether or not our system has an issue is knowing what "normal" looks like. We want our system to have predictable behavior, as it makes the system easier to reason about and makes it easier for us to react when something is wrong.

We also want our software to be built and deployed in a predictable way. The steps involved in taking code from source control, compiling and packaging it, and creating a deployable artifact need to be repeatable.

When we deploy a service, we want it to be deployed in the same way each time. We want the automation of our infrastructure to be repeatable—reconfiguring our cloud resources isn't when we want to be injecting the drama of uncertainty into the proceedings.

For this reason, you should keep LLM-driven AI solutions away from decision making around *how* your software is built or *how* your infrastructure should be configured. That is the place for deterministic tools such as Terraform or continuous integration (CI) tools.

I did make the argument back in Chapter 2 that the sociotechnical nature of our distributed system means that we cannot consider the system as a whole to be deterministic, even before we consider the use of AI. A distributed system has a lot of components (hardware, people, environment, and so on) that impact how the system performs, and many of those components are not in and of themselves deterministic. If anything, though, that increases the importance of making parts of our system deterministic where we can.

Correctness

LLMs are trained on large volumes of tokens, which in turn help them generate plausible output based on a given input. The term *plausible* here is important—an output might look good, but it might not be correct. When people refer to LLM hallucinations, what they typically mean is that an LLM has given them output that they know to be incorrect.

Given that LLMs emerged from translation software, this limitation is understandable. Getting a translation roughly in the right area is often good enough, and there are often multiple translations that could work. However, we now want to use LLMs in places where we need more confidence than we'd get from an answer being *plausibly* right.

A hallucination I've seen many times is an LLM-driven chatbot thinking it's the wrong year. As humans, we know what year it is, so we know the LLM is wrong in these circumstances. It's a hallucination! But how do we know the year is what it is? It's because we have lots of different ways of sensing the world around us, and we use them to work out what is true and what isn't. LLMs don't have that luxury. They are trained on a large body of tokens and have no ability to sense the world around them; they have no understanding of the physical world and how it works.

This explains why so many AI chatbots are terrible at maths. They have no fundamental concept of mathematics (which is somewhat ironic given that LLMs are routed in statistical models). I had an issue with a client recently who was complaining that ChatGPT was doing a terrible job with some calculations. The client had just realized that the calculations performed on tables of data they had been providing to ChatGPT were wrong. The irony is that these tables of data had been copied from Excel, which despite what other problems it might have is actually quite good at maths.

As humans, we tend to anthropomorphize nonhuman entities that seem to show some signs of intelligence. We project human qualities and human thought patterns onto them, even though the entities in question might have completely different cognitive systems—or in the case of LLMs, no cognition at all. This is problematic, as it can lead us to misunderstand what LLMs can (or can't) do. We see them able to do in 10 minutes what we might take all day to do, and we make assumptions like "well, they must be at least as good at maths as I am!"

This misplaced confidence is further complicated by the language that chatbots use to talk to us. Answers are given confidently, stated firmly.

There is no equivocation. “This is the answer!” they say, and often we trust them even when we shouldn’t.

Again, my goal is not to suggest that this undermines the benefits of LLMs more broadly. However, it’s vital to ensure that in situations where you need the answer to be right, you have some way of checking LLM output. When an LLM generates code for me, I can check it is correct by making sure the code compiles and testing the resulting system. But if you can’t find some way to validate that the LLM is giving you the right answer in situations where it’s essential that the right answer is given, then you may need to look elsewhere for a solution.

All is not lost. As I mentioned earlier, LLMs are just one facet of the wider world of AI. World models are a concept to help AI systems understand the environment in which they operate and make predictions about what might happen to that environment based on some sort of input. If I asked you to picture an egg sitting on the edge of a high wall, and then I asked you what would happen if I pushed that egg off the wall onto the concrete floor ten feet below, you would reason that the egg would break.

An LLM has no ability to make predictions like a world model can. If you asked an LLM-powered AI chatbot what happens if you push an egg off a wall, it’s going to give you an answer based only on the data it’s trained on. It might give the right answer if the training set contained enough egg- and wall-related data. However, the appropriate world model with a fundamental understanding of physics will be able to reason about things in a very different way. It would know that eggs are things that are fragile, that gravity is a thing that exists, and that concrete is hard and could infer from all of that that the egg is likely doomed. World models, by the way, are far from pie in the sky. They are already widely used in games, for example.

I have hope that the integration of world models into our AI-infused software will help our systems give better, more correct answers, but at present their use in the general-purpose AI solutions we are using seems fairly limited.

Infrastructure Availability

The current surge of GenAI use is straining our available computing resources. It's not just the GPUs we have associated with AI workloads that are becoming more expensive and scarce. Memory has spiked significantly in the last year, with some types of memory prices increasing by over 200%, and with SSD prices also impacted.

If you aren't running AI workloads, you might think this won't impact you, but already I'm seeing examples of client non-AI workloads being impacted by lack of availability of infrastructure in the public cloud, with costs and lead times for on-prem infrastructure also increasing. This in turn has a potential impact for systems reliant on dynamic scaling on the public cloud, and also for some types of disaster recovery (DR) strategies. I explore the topic of DR and the impact of infrastructure availability in more detail in Chapter 10.

Model Changes

While the huge and dramatic changes we saw in model capabilities have flatlined somewhat recently, each new generation of model comes with promises of improvements in terms of cost, speed, quality of results, or some combination thereof. The challenge is that the improvements from one version of a model to another aren't always linear. This makes it even more important that the solutions you are using allow for models to be switched easily to take advantage of these changes.

It's not the case that each new generation of model will be better than the previous version on all axes. Some newer models might focus on better reasoning while delivering results more slowly or at greater cost. Sometimes these variations are clear and upfront, but other model behavioral changes can be harder to spot initially.

For example, when ChatGPT 5 was released, there was a lot of backlash against the perceived "personality" differences. This led to OpenAPI having to extend access to ChatGPT 4 while it worked out these kinks. This

example underscores the importance of testing before switching implementations.

Vendor Instability

To a great extent, the current AI vendor market is a layered, interwoven affair. Initially, OpenAI and Anthropic largely avoided building their own applications (with ChatGPT being a significant outlier). Instead, they provided access to their models via APIs, allowing other companies to build their own customer-facing products on them. For example, AI coding assistant Cursor was, for many months (an age in the world of AI!), the most popular tool in this space. Its dependency on Anthropic became problematic, however, when Anthropic decided to compete and changed the prices that Cursor had to pay to access Anthropic's models.

In other spaces, application developers are finding that the very platforms they depend on are now their competitors too. Lots of startups build quickly and burn out fast, and companies like Windsurf hit the big time before being fought over by much bigger firms (with their customers suffering in the meantime). And just days before I wrote this, a service initially launched to much fanfare, OpenAI's Sora product, was shut down entirely. At the time it was shut down, according to estimates by Forbes, Sora was costing OpenAI over $15 million a day to run, with no obvious path to a revenue stream.

Looking back at previous similar technology shifts, it's common that the early companies that drive such changes don't survive. During the period of railway mania that happened in the UK and Ireland in the 1840s, hundreds of railway companies came into existence and then faded quickly after it became clear that they weren't financially viable. In the first dot-com boom, many firms came and went. Given where we are in the growth of AI, we should fully expect firms we rely on today to go under, and new companies to come to the fore.

Cost

At present it seems clear that our use of AI products from the larger firms is being significantly subsidized. Transparency around actual revenue and costs from firms like Anthropic and OpenAI is somewhat lacking, so it's hard to be exact, but it seems telling that despite significant recent price increases, these firms are still all hemorrhaging money. Now, if the reason for their losses is investment in infrastructure, that's one thing (we'll explore that more in the next section). However, it seems likely, just taking into account the cost of serving existing traffic, that the real cost of providing the service is much higher than the costs we are currently being asked to pay.

To take one example, at the time of writing, Anthropic charges a flat monthly subscription fee for Claude Code that appears to greatly mask the real costs of providing the service. One analysis stated that even at the conservative end, a power user for Claude might be paying only $2,400 for an annual subscription, against an actual cost to Anthropic in excess of $90,000. While we need to take some of these figures with a pinch of salt (due to the aforementioned challenges in drawing actual conclusions from Anthropic's publicly available financials), I've seen multiple analyses of Claude and of other AI firms that highlight similar discrepancies.

Of course, startups subsidizing real user cost is nothing new. A firm might raise money to build out its offering and then use some of that raised capital to provide free or cheap services to entice people in. At some point down the line, that firm needs to find a way to balance the books. So either people need to start paying what it actually costs to run the service, the cost of delivering the service has to decrease, or else the firm needs to find other as lines of revenue. Already we've seen many of the big firms increase their pricing model, and at the time of writing OpenAI is openly discussing whether it can continue to justify flat- fee subscriptions in the future. Microsoft has already dropped support for its flat-fee subscription for Copilot in favor of a pay-per-token model.

This uncertainty about pricing creates a challenge when you start to rely on products. If the price suddenly changes, you may have to curtail your use of the software or restrict who has access. Pay-per-use products often lead to

limited access in corporate environments, or to limited access to the tool in general, which can have a knock-on effect in terms of being able to work as a team. It becomes harder to plan into the future if you are unsure about potential cost spikes later on.

Yet Another Financial Bubble?

A lot of money is being invested in, and spent by, AI companies. A vast sum. It probably represents the largest deployment of capital in recorded human history. But a statement like that doesn't really make clear what is going on. The figure that jumped out at me came from a 2025 Bain report, which calculated that, to justify the investment in AI, the sector will need to be generating $2 trillion in revenue per year by 2030. Meanwhile, the most optimistic forecasts of revenue growth appear to show a significant shortfall of around $800 billion. So even if you take the most bullish view of where this sector is going, the current spend doesn't seem justified. And whenever a sector's investment and valuation seem out of step to such a huge degree with what the market can actually support, fears of a financial bubble grow.

As I've mentioned, previous technological shifts saw many of the pioneering companies behind changes such as the construction of the railways or the development of the early commercial internet. One argument used by those who are less concerned about whether there is a bubble is that the firms that crashed and burned left behind infrastructure that we were able to make use of for many years afterward. For example, the telecom companies that were overvalued during the first dot-com boom laid the fiber that we later came to rely on. So even if we are in a bubble right now, we'll still end up better off over the long term, right?

My counter to that is twofold. Firstly, it's not immediately clear that the amount of money being spent will result in infrastructure being constructed that we can actually use. The liquid-cooled data centers of the AI age are quite different from the primarily air-cooled data centers we are used to. The cabling, cooling, and power generation requirements are fundamentally different. And the servers being installed are often designed specifically for

specialist AI workloads. If that market dries up, it's not clear how general non-AI workloads can make use of this newly built infrastructure.

The second issue I have with the lack of concern over the potential for a bubble is the amount of contagion that a collapse would cause. The first dot-com bust was limited in its impact. But now we have a lot more exposure among both governments and institutional investors to the fate of AI companies. This is compounded by some of the circular financing that goes on—for example, NVIDIA investing in an AI company so that said AI company can buy NVIDIA chips. The problem with circular financing is that if things go wrong, more firms can be brought down by a small number of failing entities.

I have a more general concern, and that is that rather than allowing the larger firms to go bust, governments will instead step in to bail them out, leaving taxpayers around the world to foot the bill. The idea of companies asking for government bailouts sits very uneasily with me, as fundamentally it looks like a form of socialist capitalism. Which is to say that whatever your views are on socialism or capitalism, it seems like neither is being done well when failing corporations are constantly being bailed out.

So what do the wider concerns about an AI bubble have to do with building resilient applications? Well, to be honest, this is my book, and I largely wanted a space to rant. But I also wanted to outline that the general uncertainty about the role of AI in both our industry and the wider world exists on many different levels. So whatever you might think the future holds, you should very much expect to be surprised. That said, I don't want to leave you all depressed, so let's look next at how you can navigate this uncertainty.

How to Handle Uncertainty

In the previous sections, I've outlined why the world of AI is so uncertain right now. However, I want this book to focus on practical, actionable advice. With that in mind, I want to outline some key steps you can take to make sure that you're prepared for the uncertainty ahead.

Multimodel, Multivendor

With model behavior, speed, and quality changing from version to version, it's vital that you build your software to support changing models. Not only do you want the ability to change models to make use of the latest and greatest, but if costs start to increase from one vendor, you want to make sure the barrier to finding another vendor isn't too great.

> **KEEP YOUR OPTIONS OPEN**
>
> Make sure you aren't wholly dependent on a single model or AI platform. If model quality degrades, or prices from OpenAI or Anthropic go up, you want to make sure that you have the flexibility to go elsewhere.

It is worth highlighting that mixing models from different vendors in a single flow can exacerbate issues. For example, a flow that involved models from Anthropic, OpenAI, and Google's Gemini would be impacted if any of those models had availability issues. All those models have to be available and functioning correctly at the time the flow is executed. Also, prompts that work against one model may not be entirely portable to another—a prompt tuned against one model will have to be reevaluated when moving to a different model, even if you stay within the same vendor ecosystem.

Taking a multimodel approach extends to your vendors too. If a vendor you are dependent on is in turn dependent on a single vendor's model or API, you and your vendor become vulnerable to volatility in that model or to being impacted by a sudden price increase. Cursor was forced to pass on a significant cost increase to its customers in 2025, as it relied on the use of Anthropic's services, and Anthropic changed its pricing structure, which led to Cursor having to pay much more. Coincidentally, Anthropic's Claude Code is a direct competitor of Cursor. If Cursor had been able to use AI models from another firm, it might have been less vulnerable to this issue.

Cost Transparency

Given concerns about rising costs from vendors, and given that pay-as-you-go pricing models can lead to surprising and unsustainable bills, having the ability to see on a regular basis how much money you are spending, and where, can be really important. This doesn't need to be about limiting spend necessarily; it can be about understanding how you spend money and understanding what value that brings.

James Stanier, CTO at digital healthcare firm Nordhealth, has advocated for being able to see spend on a per-person basis. Rather than this being about shaming people who spend too much money, he makes the case that this visibility drives better understanding of how the software is being used and helps drive the adoption of tools that are proving to be valuable. In his 2025 talk at LeadDev Berlin, "LLMs: An Operator's View", Stanier explains that if you see a person spending a lot of money, this shouldn't be seen as a bad thing, as typically it shows someone trying to get more done. He cites an example of one of his QA engineers blowing through €1,000 in a single week on Claude Code, which was a much higher spend than normal. After pairing with the engineer in question to understand where this money was going, Stanier was able to see that this spend allowed the engineer in question to more than double his productivity.

Having the information about spend available to you, and being able to talk to people about how they use these tools and the benefits (or problems) they cause, will help you all make much better and more informed decisions.

It can also help you spot egregious overspend. Uber recently revealed that it had burned through its entire annual AI budget in just the first four months of the year, and Microsoft announced it was pulling Claude licenses from its employees, due in part to cost, apparently.[5] These stories might cause one or two raised eyebrows, but they pale in comparison to the recent (admittedly unsubstantiated) report from Axios that an unnamed enterprise company spent $500 million on Claude licenses after failing to put cost controls into place.

> **TIP**
>
> Make your AI spend visible! It can help you understand how people are using it and also help you spot when the bills start climbing precipitously.

More Frequent Vendor Analysis

Vendor management is a common activity that many firms carry out. It typically consists of a regular review of the main vendors that the organization makes use of to help it make better decisions in the future. For example, seeing that a large amount of money is being spent with a particular vendor despite the quality of service from said vendor degrading might cause an escalation to either get the price reduced or get the quality of service improved—or it might kick off a process to find a replacement.

In my last role at Thoughtworks, I helped lead our internal technology operations. Although my main job was to help provide support to internal delivery teams, I was also involved from time to time with our own vendor management activities. As part of these activities, we would regularly assess the general health of the vendors themselves. During the last review I was involved with, we were actively concerned about the health of one of our main technology suppliers; the supplier was in a very competitive market, and its market share was dropping. Although we were happy with the quality of service it provided (and with the cost), our fear was that wider market forces would drive it out of business. As a result, we decided to get to know the vendor much better but to also prepare a list of alternative solutions that we could select if the vendor in question didn't survive.

Given the degree of uncertainty around the AI sector in general, I'd recommend regular vendor management activities. If you already have a normal vendor analysis cycle, consider increasing its frequency for AI vendors. If possible, find out if any vendors are in turn dependent on another provider, as in the case of Cursor being dependent on Anthropic's cloud services. I'd be more concerned about a vendor that is in turn wholly dependent on a single company than I would be about a vendor that has

more flexibility about whose services it uses. If you've never done vendor management before, consider instituting a lightweight process so you can build up a better understanding of how exposed you are to risk in this space.

With the concerns around the potential security vulnerabilities that come from the use of AI, I'd also recommend keeping your threat model up-to-date with regular threat assessments. As with vendor reviews, consider increasing the frequency of your normal threat assessments in areas with heavy use of AI, as this technology and how we use it is changing very quickly.

> **REVIEW YOUR AI VENDORS REGULARLY!**
>
> Given the volatility in the space, make sure you review your AI vendors much more frequently than normal vendors.

Summary

I realize that this chapter may have ended on somewhat of a negative note. But I wanted to be clear that we have real, practical use cases for AI in helping us build more resilient distributed systems. Most of the use cases I outlined in this chapter are being used daily by people all over the world. The key thing is that the solutions I have shared here still put humans at the heart of decision making. It's clear that, due to the challenges I have covered here as well, we are a long way from a truly autonomous AI SRE.

Despite the obvious benefits of using AI, we do need to be fully aware of the downsides at both the micro and macro level. At the micro level, LLM-powered AI requires us to be OK with nondeterminism, whereas there are many activities we perform in the context of building resilient systems that need to be fully repeatable and deterministic. At the macro level, concerns about the AI sector as a whole mean we are dealing with an ever-changing (and hopefully) improving set of products built on very shaky financial foundations. It doesn't matter how good the tooling is if it's too expensive to use (or the company making it goes bust).

As is the nature of things, this uncertainty around AI's role in our industry and society as a whole will fade as we learn more and as the market settles down. But be warned: I think we are still very much in the middle of the storm, and the calm waters may be several years away yet.

A lot of this chapter was about uncertainty, and one thing that can make the resiliency of a system quite uncertain is what happens when machines fail. So in Chapter 8, I'll explore how to make sure our systems still work even when machines let us down.

---

1 Bonus points if you get either reference—let's just say these method names relate to music that was around at that time, which very much dates this anecdote.

2 To be transparent, Martha Lambert, who is one of the book's reviewers, also works at incident.io—but she didn't ask me to write this! She did, however, make sure I didn't get any of this wrong.

3 Gayathri Mohan, *Full Stack Testing*, 2nd edition (O'Reilly, 2026).

4 Shreya Shankar and Hamel Husain, *Evals for AI Engineers* (O'Reilly, 2026).

5 Although it's possible this announcement was just as much about promoting the use of its own competing Copilot tooling.

Part II. Dealing with Failure

In the second part of the book, we dive into more complex technical topics and look at how to handle failure of everything from individual computers to entire data centers. You'll also see how a system needs to adapt when these failures occur.

Chapter 8. Handling Machine Failure

Machines fail, and there are diminishing returns in reducing the failure rate of individual machines. And sometimes, even if a machine is operating correctly, you might not be able to reach it due to a networking issue. But machines can come and go as part of normal operations too. If we broaden our definition of a machine beyond just the physical computer and also think about virtual machines or containers, normal deployment activities can involve machines shutting down and new machines, potentially located in different places, spinning up in their place.

We have to accept that machines can come and go, so we need to work out how to deal with machine failure when it happens. A theme running through this book is "making sure that as many things go right as possible," so with that in mind, we should turn our attention to how we make sure things go right even if a machine becomes unreachable.

In this chapter, we'll explore how clients can find servers, as this can be key in recovering from machines being unavailable. We'll also look at the topic of load balancing as a mechanism for reducing the impact of a single server instance failing. Finally, we'll look at the challenges that stateful processing causes when also trying to deal with failure.

Service Discovery

If the machine a client is talking to suddenly becomes unavailable, the client may need to be pointed to another machine. With that in mind, let's take a look at some of the options around service discovery.

Pattern: Hardcoded IP Resolution

IP address for a networked endpoint is hardcoded in the client.

The address of the server is hardcoded in some form of configuration in the client. If the server address changes, the configuration has to be updated manually.

How it works

This is about as straightforward an implementation as you can think of. Just have a client hardcode an IP address for the server it wants to talk to. You should put the address in a configuration file rather than in code, so as to make your life slightly less painful if the IP address needs to change. Don't get me wrong—having to manually update lots of configuration files when a machine location changes will still be painful; it just won't be as terrible as having to make a similar change in code.

When to use it

There is a simple answer to the question of when to use this pattern: very rarely. Hardcoding IP addresses doesn't tend to end well when you need to update those IP addresses. For a public-facing service, this approach is entirely unsuitable. Note that hardcoding a Domain Name System (DNS) entry is fine, as DNS is inherently dynamic. The IP address given as a result of a lookup for a DNS address can change.

Even in my home network I make use of a local DNS server for home devices to avoid having to hardcode IP addresses for lookup purposes. In fact, the only time I would ever hardcode an IP address is when I need to point to a specific DNS server, and even then I would provide a list of DNS servers in my network configuration in case the first DNS server fails.

Pattern: Dynamic Service Discovery

Clients can receive new addresses for servers as they change.

With a dynamic service discovery mechanism, if the network location of a server changes, this change is pushed out to the clients through a variety of mechanisms. An example of dynamic service discovery can be seen in

Figure 8-1, with the newly launched `Pet` instance registering itself with the service registry.

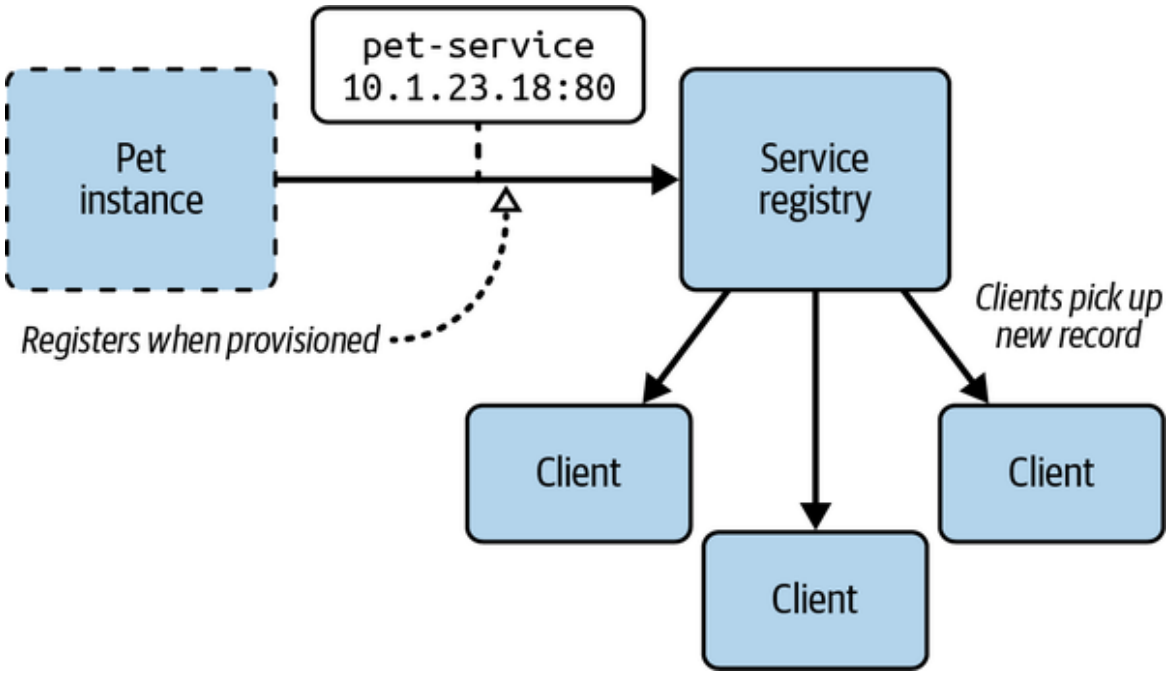


Figure 8-1. The entries for the `Pet` service are dynamically updated when new instances are launched

This is incredibly useful when a machine fails and needs to be replaced. Dynamic service discovery implementations also commonly support listing multiple IP addresses for a single record and can therefore support client-side load balancing.

How it works

Dynamic service discovery requires two key parts. Firstly, you need a source of truth for the server addresses. Secondly, you need a way to get these updated addresses to the clients. There are a few ways to implement this pattern, starting with one of the foundational mechanisms of the web.

Domain Name System (DNS)

Let's start with something that is likely very familiar to you and that you use probably hundreds of times a day without realizing it. DNS lets us associate a name with the IP address of one or more machines. That is a bit of a simplification, of course, as you can do a lot more than just that with

DNS, although a full discussion of its capabilities are outside the scope of this book.

DNS updates are applied to a primary nameserver. You can then run one or more secondary nameservers that receive updates from the primary. This helps balance the load of DNS read requests and also provides the ability to still serve reads if the primary node fails. Clients can provide a list of DNS nameservers to use. This could then be a list of both primary and secondary nameservers, or you could decide to run your primary and secondary nameservers behind a load balancer and provide the clients with the IP address of the load balancer itself. You could of course include links to multiple primary nameservers in a client, although you'd need to fully understand the implications if these primary nameservers could end up disagreeing about the address of a machine.

DNS has a host of advantages, the main one being that, because it is such a well-understood and well-used standard, almost any technology stack will support it. Unfortunately, while a number of services exist for managing DNS inside a system, few of them seem designed for an environment in which machines are being decommissioned and provisioned frequently.

DNS entries for domain names have a *time to live* (TTL). This is how long a client can consider the entry fresh. When we want to change the IP to which the domain name refers, we update that entry, but we have to assume that clients will be holding on to the old IP for *at least* as long as the TTL states. DNS entries can get cached in multiple places (even a Java Virtual Machine will cache DNS entries unless you tell it not to), and the more places they are cached in, the more stale the entry can be.

An obvious option is to use short TTLs to ensure that changes to records propagate more quickly. According to the specification for DNS, RFC 1035, a TTL as low as zero is allowed, in which case no caching is performed. However, the DNS server implementation you are using may or may not support TTLs that low. Another option for dealing with the volatility of the IP addresses is to have the domain name entry for your

service point to a load balancer, which in turn points to the instances of your service.

A final consideration for the use of DNS is that DNS servers typically do not attempt to validate that the addresses point to healthy machines. As long as an entry is in a valid format, it would be allowed. This means you could accidentally insert entries to incorrect IP addresses—or in the case that a machine was working but subsequently failed, the pointer to the now unhealthy machine would still be in the DNS server.

Public cloud providers at least give you access to an easy-to-automate DNS server, such as Amazon's Route 53. What seems to be lacking is a range of good general-purpose self-hosted DNS servers that are well suited to a more dynamic environment. Some dedicated service discovery tools, such as Consul, can provide limited ability to support DNS lookups, and we'll look at those next.

Consul, etcd, et al

Whereas DNS was created primarily to support lookup of machines on the web, other tools have been created to support the more dynamic lookup needs within systems themselves. Consul and etcd are two examples of those tools. They are fundamentally quite similar, providing a distributed key-value store with specific support to act as a service registry.

To avoid being a single point of failure, both Consul and etcd run on multiple nodes. Both tools implement a version of the Raft consensus algorithm, which performs a leader election when a node fails. As per normal recommendations when running Raft, you'll want an odd number of nodes—at least three, unless you are willing to tolerate a single point of failure.[1]

Unlike DNS, both Consul and etcd support health checking for nodes in their registry. This makes them much more useful as part of a client-side load balancing scenario (something we'll explore later in this chapter). Each also provides an API that allows clients to be told when entries change,

meaning that updated addresses can reach the clients much more quickly than occurs with the TTL mechanism provided by DNS.

For many years, Consul has also provided the ability to work as a DNS server for some types of DNS records. You can also achieve this by running something like the CoreDNS server on etcd. This means that legacy software that can't be updated to use the APIs of your fancy new service discovery tool can use DNS instead, using the same source of truth.

Consul does have an add-on tool that can help bridge the gap between hardcoded and dynamic service discovery. Consul-template provides a way to update text files based on entries in Consul. At first glance, this doesn't seem that interesting, until you consider the fact that with consul-template you can now change a value in Consul—perhaps the location of a service, or a configuration value—and have configuration files all across your system dynamically updated. Suddenly, any program that reads its configuration from a text file can have its text files updated dynamically without needing to know anything about Consul.

It's worth highlighting that etcd is baked into Kubernetes, and it's used for service discovery within Kubernetes itself. So if you've already got Kubernetes, you're already using etcd for dynamic service discovery within the Kubernetes cluster.

When to use it

Start with DNS if you can, use etcd within Kubernetes, and consider the use of Consul if you can justify the value of the other tools in the Consul ecosystem (for example, its excellent support for the Vault secret storage system or for tools like consul-template).

The more dynamic your environment is—the more machines come and go—the more vital a dynamic service discovery mechanism is. This is especially the case in a container-based environment, when the "machines" in question are frequently spun up and shut down as part of normal software upgrades.

If your system runs entirely in Kubernetes, then internal service discovery is obviously already handled by etcd without any work needed from you. If your environment is more static, or in situations where you plan to use fault-tolerant load balancers for addresses, DNS can be a perfectly fine solution, assuming you have the ability to easily automate DNS updates and you have confidence in the fault-tolerance of the DNS nameservers themselves.

If you're running outside a container platform like Kubernetes (or are perhaps running in a hybrid environment with a mix of container and non-container workloads), running a dedicated dynamic service discovery tool such as etcd or Consul may well make sense, especially as these tools also deliver some great solutions for distributing general configuration information. In a mix of legacy and more modern workloads, Consul seems especially well suited to the task because of its mix of native support for some DNS record lookup and for tools like consul-template.

Load Balancing

Load balancing across a number of machines has been a commonly used approach for both scaling systems and improving resiliency for decades. Historically, load balancers were hardware-based devices that sat between client and server, with requests being transparently diverted to their destination.

With the advent of software-defined networking, software-based load balancers have become more common. The use of HTTP/2 for interprocess communication has further pushed attention to client-side solutions. Server-side load balancing and client-side load balancing have a different set of trade-offs around resiliency, which we'll explore in this section.

Whichever approach you go for, load balancers can help in terms of both scale and robustness. You can add more machines to a load balancer pool to add more computing resources to your service, allowing you to handle more load. From a resiliency standpoint, even if a machine fails, traffic can be diverted to the remaining healthy nodes.

Load balancers give you a great mechanism for distributing calls when making direct point-to-point network calls using protocols like HTTP. Message brokers are another mechanism that we can use for distributing load, and we'll look at those in more detail in Chapter 9.

LOAD BALANCING VERSUS LOAD SHARING

Technically speaking, when splitting work across a number of workers, there is a distinction between *load balancing* and *load sharing*. Load sharing describes load being shared across multiple workers. With load balancing, you are actively aiming to evenly distribute work so that all workers are handling the same load. The distinction here can be important—a load-sharing approach that has no concept of the load a worker is actually handling can result in workers being overloaded.

In practice, load balancing ends up being used as a catchall term to cover *both* load balancing and load sharing, which is what I have done here. I have called out situations, though, where uneven distribution of load can be a problem.

Pattern: Server-Side Load Balancing

Load sent to a server is transparently distributed across a number of nodes.

With server-side load balancing, a client sends a request to the endpoint of a load balancer, which in turn routes the call to an available instance, as shown in Figure 8-2.

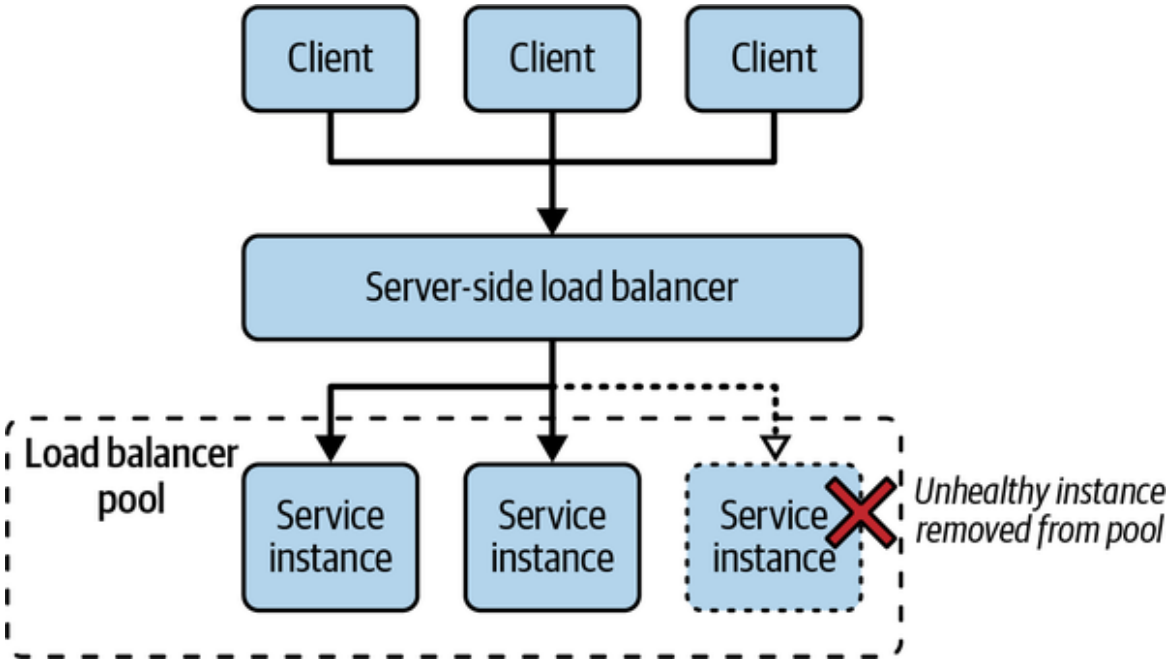


Figure 8-2. A server-side load balancer

The goal of the load balancer is to distribute work evenly so that none of the instances in the load balancer pool are overburdened. Additionally, the load balancer will remove instances from the load balancer pool if they become unhealthy and allows new nodes to be added to its load balancer pool.

How it works

Traditionally, server-side load balancers were hardware devices, but they are increasingly implemented as software-based proxy solutions such as HAProxy, `nginx`, or the load balancing provided to services running inside Kubernetes (which in turn is underpinned by etcd). Fundamentally, though, with whatever solution you pick there are three key aspects of the implementation that we should explore in more detail: how the worker pool is managed; how load is distributed; and finally, how to ensure that the load balancer itself isn't a single point of failure.

Managing the worker pool

A server-side load balancer needs to be aware of what instances are available for its worker pool. This will normally involve each instance being registered with the load balancer, combined with the load balancer

itself keeping an eye on whether the instances are considered to be healthy enough to have work distributed to them.

This means that in a more dynamic environment in which new instances are being added (or removed) frequently from the load balancer pool, automating the process of adding or removing instances becomes very important. Kubernetes handles this for you as you add or remove pods in a service, and cloud-based load balancers such as AWS's Elastic Load Balancing provide APIs to allow you to easily automate adding and removing nodes.

A common approach to detecting unhealthy nodes in the load balancer pool is to monitor the responses of the load balancer pool instances; if any individual instance breaches error thresholds, it can be removed from the pool temporarily. Exactly how this is done varies from implementation to implementation.

Distributing load

When a call is sent to a server-side load balancer, the load balancer needs to work out which instance to route the call to. Load balancers can vary greatly in terms of how sophisticated their balancing logic is. Some simple load balancers will distribute calls based on the number of concurrent requests that each node in the worker pool is handling. This is normally better than a simple round-robin approach, but it does have some downsides.

If load balancing is based on numbers of connections, you can have an issue if some connections are more expensive than others; this could still result in overloaded nodes. If your service has some operations that are much more resource intensive than others, then basing load balancing on numbers of connections is more likely to cause you a problem.

Secondly, most server-side load balancers are fairly "dumb" layer 4 (L4) load balancers—that is, they operate at the "transport" layer of the OSI networking model. They just see the TCP connection and can route it to an instance. If you're using a protocol like HTTP/1.1, this isn't too problematic. In Figure 8-3, you can see three requests from a single client

that are each made via a dedicated connection, and the load balancer quite happily distributes these connections evenly across the available instances.

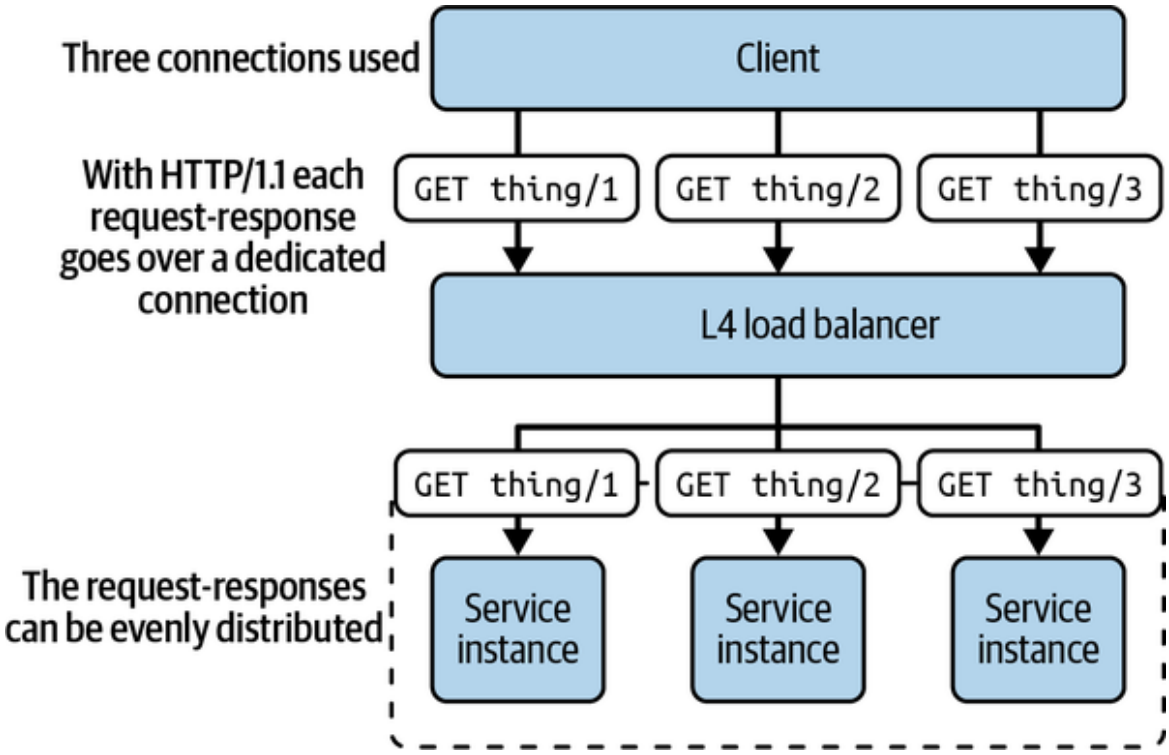


Figure 8-3. Distribution by connection with HTTP/1.1

The L4 load balancer doesn't need to have any knowledge of what is happening inside the connection. This keeps the load balancer nice and simple, but it can be problematic with some protocols. For example, HTTP/2 and above allow multiple request-response flows to be run in parallel over a single connection. This approach is more efficient, as it can keep down the number of connections that need to be established, but an L4 load balancer can't "see" the individual requests. As a result, all three of these requests will end up getting sent to a single instance, as shown in Figure 8-4.

A way around this would be to make use of a layer 7 (L7) load balancer, which operates at the application layer. This would give the load balancer visibility of the actual requests and responses flowing over the connection. L7 load balancers are more likely to be available on software-based solutions, in my experience.

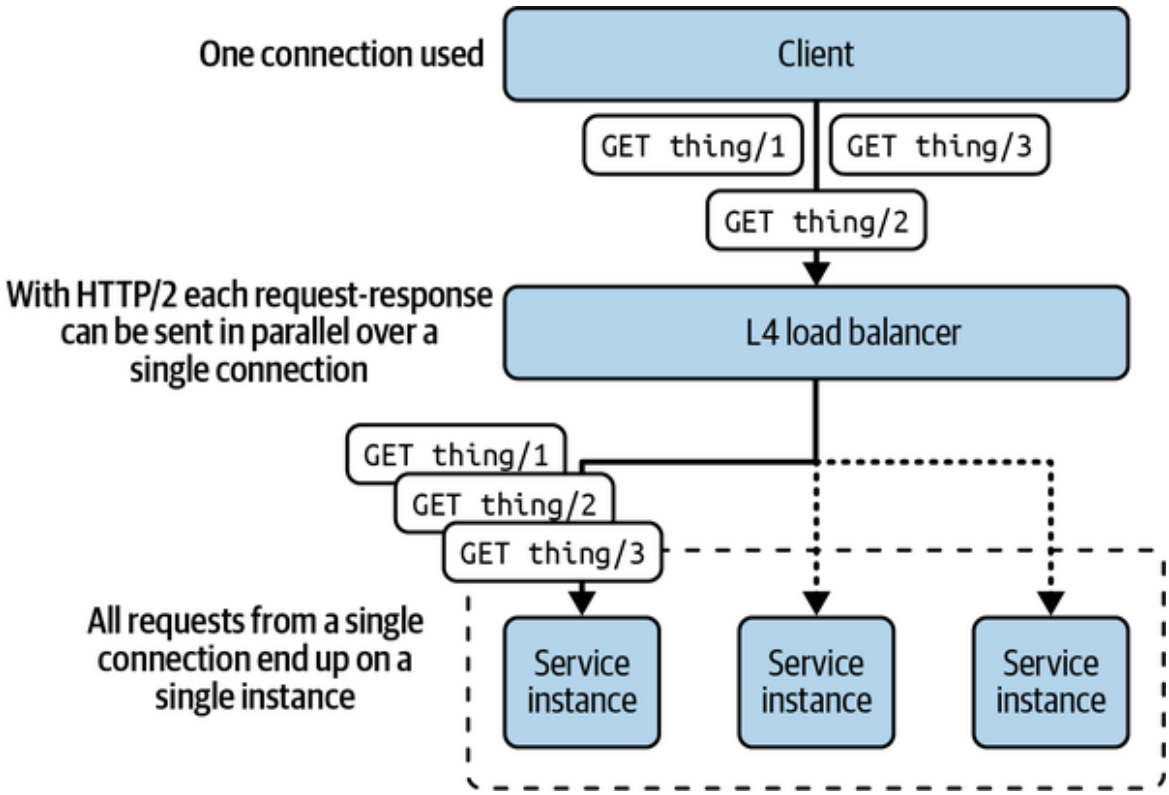


Figure 8-4. Distribution by connection can be problematic with HTTP/2 and above

If the load balancer is able to get more detailed information around the actual load of the instances in the load balancer pool (for example, actual CPU and memory use), then of course smarter decisions can be made regarding which instance should receive the next call. The flip side is the increased complexity involved in the configuration of the load balancer; as I hope you've already learned from this book, increased complexity can often bring more resiliency headaches with it!

Fault tolerance

Server-side load balancers can of course become their own single point of failure. To work around this, you can run multiple load balancers, perhaps using internal DNS to provide failover between them.

A more advanced solution would be to make use of the Border Gateway Protocol (BGP). BGP is primarily used to help route calls effectively over the internet. Calls across the internet have a number of different choices as far as how to get from point A to B; there could be many potential routes to take. A deep dive on BGP is well outside the scope of this book, but

essentially BGP lets a client know what routes are available and which ones are preferable. You can use BGP within a network (typically referred to as Internal BGP, or IGBP) to establish paths via one of a number of load balancers. When one of those routes ceases to be available (perhaps because it includes a load balancer that is no longer functioning), one of the alternative BGP paths via one of the other load balancers could be used instead.

When to use it

Server-side load balancing benefits all clients of the service, and as such this is typically my default go-to for load balancing. With a client-side solution, you rely on each client properly implementing its load-balancing approach. In addition, cloud-based solutions for server-side load balancing are plentiful.

If you are making use of the benefits of HTTP/2, server-side load balancers that operate at layer 4 are likely not a great fit. If your server-based solution can support layer 7 load balancing, then great. Otherwise, you might want to consider a client-side approach, as your client code will be aware of what calls are being sent over the shared connection, making implementation of layer 7 load balancing much easier.

A potential downside of a server-side load balancer compared to a client-side option is that the use of a server-side load balancer requires additional network hops: the request from the client to a server instance has to go via the load balancer. In situations where you have very tight tolerances around latency, or where the network you are working with has inherently high latency, adding hops may not be desirable.

I do want to highlight that the use of centrally controlled load balancers can be problematic. If you have a big, expensive hardware-based load balancer that everyone is expected to use, it often ends up getting locked down. Being able to add and remove nodes to a worker pool or tweak load balancer configuration can be vital to getting your service to run the way you want. In such a situation, I'd suggest looking at running your own load

balancer behind the centrally managed one. The downside of adding another network hop may well be worth it if it puts you back in control.

Pattern: Client-Side Load Balancing

A client is made aware of the available server nodes and makes its own determination on which node to send work to.

With client-side load balancing, the client itself decides which instance to route calls to. There are a number of implementations for this, ranging from the simple to the complex. What they share, however, is a need for the client to have knowledge about what instances are available to handle calls, as shown in Figure 8-5.

Client-side load balancing gives the client much more control over how the load balancing is done and can help you avoid the concerns around server-side load balancers being single points of failure.

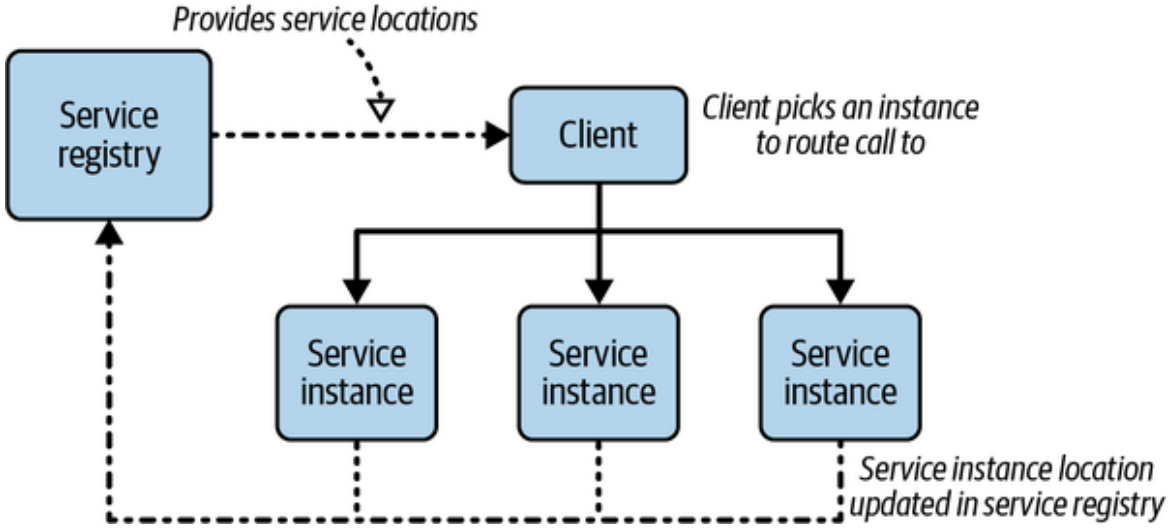


Figure 8-5. Client-side load balancing

How it works

Client-side load balancing is implemented in code, either directly in the client itself or in a machine-local proxy, potentially as part of a service mesh. There are two main aspects to implementing client-side load balancing: discovering the service instances and deciding how to distribute the load, which we'll look at next. Service meshes can arguably give us a

hybrid of client-side and server-side load balancing, so we'll look at those in more detail as well.

Finding server endpoints

Clients need to receive information about which instances are available for a given service. This is typically done out-of-band. Simple solutions such as listing instances in a client config file struggle to deal with dynamically changing service instances. On the other hand, more dynamic solutions, such as those powered by service discovery systems, add complexity to the system, as they require more technology, such as a distributed reliable key-value store like etcd or Consul, which we touched on earlier.

One of the simplest client-side load balancing options is to use a DNS server that supports *round-robin DNS*. With round-robin DNS, the DNS server has a list of IP addresses mapped to a given domain name. When a request is made for the IP address for that domain name, the DNS server rotates through which address it hands out. Subsequent requests for the same record would return the next IP available in the list.

However, DNS has no concept of a server being healthy, so you could end up routing calls to machines that aren't going to respond. Also remember that with this approach a client will send traffic to a different node only after the TTL for the DNS record expires.

With server-side load balancing, the load balancer is on the active path of the request. If the load balancer is unavailable when you make a call, the call will fail. With client-side load balancing, the information about available instances is often sent separately (out-of-band). In practice, this means the client caches the server instance information, avoiding the need to make a call to the service registry prior to calling the service. This has the benefit of reducing load on the service registry, while also reducing the overall latency.

With server-side load balancing, the load balancer itself becomes a potential single point of failure. Client-side load balancing sidesteps this problem neatly, but the concerns around the resiliency of client-side load balancing then shift to whichever service discovery implementation you picked. So

the robustness of your client-side load balancing will likely be only as good as the robustness of the service discovery mechanism.

Distributing load

When it comes to load distribution, the client typically has less information to go on regarding which service instances it should send the next call to. Without additional information, a client lacks any wider context about the load being handled by the service instances. It might decide to send a call to a given service instance just because that service is next on the list, but it might not know that that instance happens to already have more load than any other instance.

Client-side load balancing can still make decisions to remove unhealthy instances based on its own local experiences, of course, but without additional information, its ability to truly distribute load is limited.

Some protocols allow for information to be sent back to the client, giving it the ability to make smarter decisions about where to send work. The gRPC protocol, for example, provides support for custom backend metrics, either in-band (using the ORCA format) or out-of-band (using the same format as Envoy's load reporting service).

Using a service mesh

With a service mesh, communication between processes on different machines is actually done via machine-local proxies, as shown in Figure 8-6. These proxies can be hooked into the available service discovery mechanisms to perform lookup and client-side load balancing in a way that is then transparent from the point of view of the code in the client itself.

The use of machine-local proxies on both the client and server side means that the penalty of additional network hops is sidestepped (local network calls are implemented quite differently when compared to remote network calls).

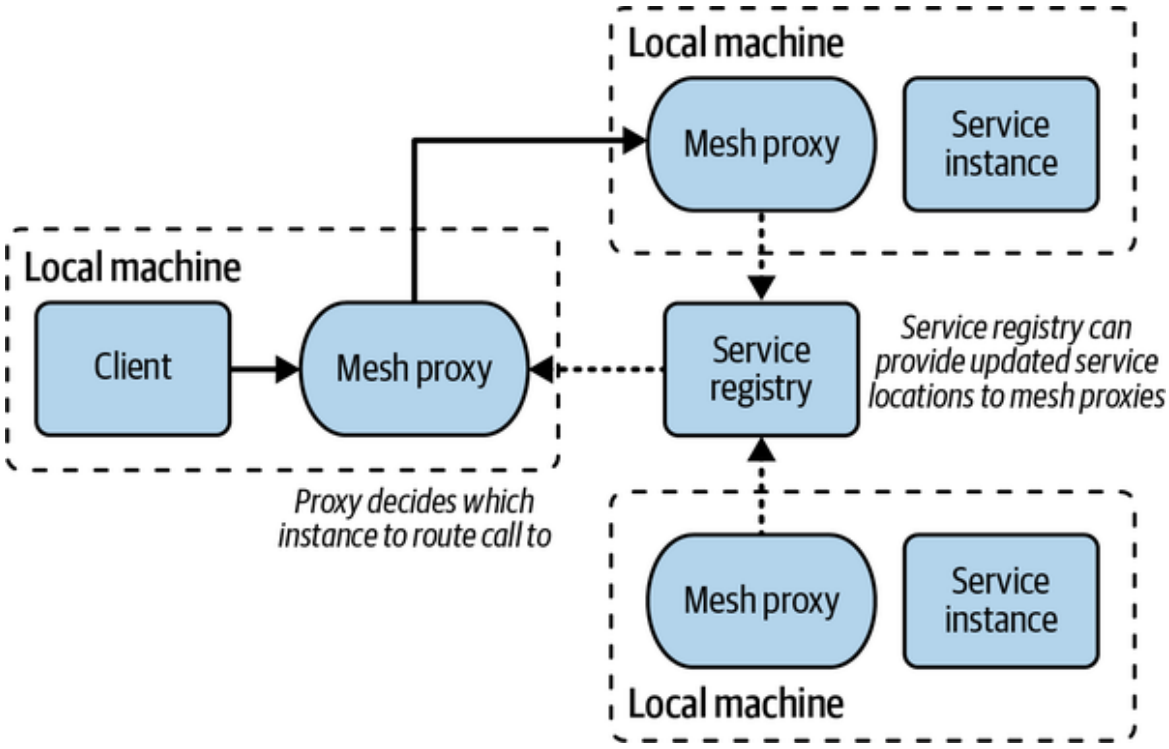


Figure 8-6. A service mesh

A service mesh can be a neat way to deliver client-side load balancing without the need for every client to implement its own logic. Service meshes can also deliver other benefits—for example, implementing things like mutual TLS becomes much easier. On the other hand, a service mesh is going to add more complexity to your system.

When to use it

One of the main benefits of client-side load balancing is the removal of extra network hops. The client is able to communicate directly with the target server without having to go via a networking intermediary like a proxy or load balancer. The exact amount of time saved by reducing the number of hops may be minimal, and thus it may not have much of an impact on end-user experience when looked at in isolation, though you should remember that reducing the number of hops also reduces network utilization in general. In a system handling a lot of load, reducing interprocess latency and network use could be significantly beneficial in allowing the system to handle more load and reducing resource contention generally.

Client-side load balancing also opens up the possibility of "smarter" load balancing. If the client-side load balancing is done inside a service instance implementation, then L7 load balancing becomes much easier to achieve. This would allow a client to balance request-response interactions over separate HTTP/2 connections if required (although if you just end up opening lots more HTTP/2 connections as a result, you may be undoing one of the key benefits of the HTTP/2 protocol in the first place!).

Server-side load balancing presents a simple endpoint for clients. But with client-side load balancing, the client has to be more aware of the server topology. If making use of sharding or cell-based architecture (which we'll look at in Chapter 11), this information may need to be communicated with the client as well. For example, if certain service instances handled the load for specific tenants, then the client-side load balancer implementation would need to implement that behavior too, unless you delegated that task to a proxy between the client and the server.

Another issue with client-side load balancing is the fact that the client needs to be trusted to perform the load balancing appropriately. For example, if a client decides to send all its traffic to a single instance of a service rather than distributing its calls, you could end up with some service instances being overloaded. The use of shared libraries or service meshes to counter that potential problem is common for client-side load balancing, because it allows different clients to share the same load balancing logic rather than expecting each client maintainer to implement things correctly.

It's also important to understand that the service discovery component of your system becomes essential to the efficacy of your client-side load balancing implementation. I've mostly seen client-side load balancing used in conjunction with more sophisticated service discovery solutions like etcd or Consul.

STICK WITH SERVER-SIDE?

Because of the complexity of implementation, I've rarely seen situations in which client-side load balancing was felt to be worthwhile. For most people, I'd suggest trying server-side load balancing first, then looking at what service-mesh based load balancing could provide (if the complexity of a service mesh can be justified), before exploring the world of client-side load balancing.

When a machine fails and a load balancer loses an instance from its load balancer pool, wouldn't it be nice if a replacement was automatically provisioned? We'll look at that next.

Pattern: Desired State Management

Continually adapting a system to ensure that its current state matches the desired state.

When a machine fails, you often want a replacement. If you have lots of computers to keep track of, manually checking the health of each one and provisioning a replacement when needed can quickly get painful.

With desired state management, you define how you want the system to look, and those requirements are maintained without manual intervention. If the running system changes in such a way that your desired state is no longer maintained, the underlying platform takes the required steps to bring the system back into the desired state.

How It Works

Defining the expected state of your infrastructure isn't new. Infrastructure automation toolchains like Ansible, Puppet, Chef, and Terraform have allowed you to do this for many years. What they all share is that the desired state is defined in a declarative way. Platforms like Kubernetes go beyond simple infrastructure definition by continually monitoring the actual state of the infrastructure and making the changes required to bring the current system state in line with the definition.

To give you an idea of what these definitions look like, in Example 8-1 you can see a Kubernetes specification stating that we expect there to be three instances of our `Pet` service.[2]

Example 8-1. A snippet of a Kubernetes definition stating that we want three copies of the `Pet` service to always be running

```
spec:
  replicas: 3
    spec:
      containers:
      - name: pet
        image: pet:3.04.5
        ports:
        - containerPort: 80
```

Internally, Kubernetes keeps track of these instances, ensuring that if one of them fails (perhaps due to a problem with the underlying hardware), a replacement pod is spun up automatically.

When to Use It

This pattern requires that the underlying infrastructure and/or the deployment of your software can be fully automated. It excels in situations where maintaining a known set of infrastructure is essential, even when you may have less control over the underlying hardware (such as on the cloud).

This pattern is also useful when you want to scale up (or scale down) the number of instances being used for a service. You can update the required number of machines (e.g., the replica count in Example 8-1), and the required number of instances will be added or removed accordingly.

For some platforms, this pattern is built in and is a core part of what the platform itself does. Kubernetes is a good example of this. If you use Kubernetes, you are implicitly adopting desired state management.

Once again, though, we do have to address one of the fundamental conundrums in resiliency: adding a system that performs desired state management increases the complexity of the resulting system—that in and of itself can introduce new sources of instability.

TECHNOLOGY CHOICES

You need to be able to trust that the system that carries out the desired state management will do its job and do it well. So when selecting tooling in this space, I'd certainly be much more conservative in my technology choices, gravitating toward tried and tested technology with a wide range of support.

Stateful Versus Stateless Processing

How you handle machine failure can be further complicated by whether you decided to pick stateful rather than stateless processing.

With *stateless processing*, the request or message received contains all the context required for the work to be processed. No state is required to be stored on a server instance across multiple client-server calls. The upshot is that any instance of a service is equally well equipped to handle the work received. *Stateful processing*, on the other hand, maintains state in memory between calls. For a number of reasons, stateless processing should definitely be preferred, but it's worth exploring both approaches so you can understand the trade-offs in more detail, and also how these different approaches impact our ability to make our services more fault tolerant.

Stateless Processing

In Figure 8-7, we see an example of stateless processing being done for the process of placing an order. The request being sent contains the contents of the shopping basket and a user ID. This is sufficient for any instance of an `Order` service to process the request.

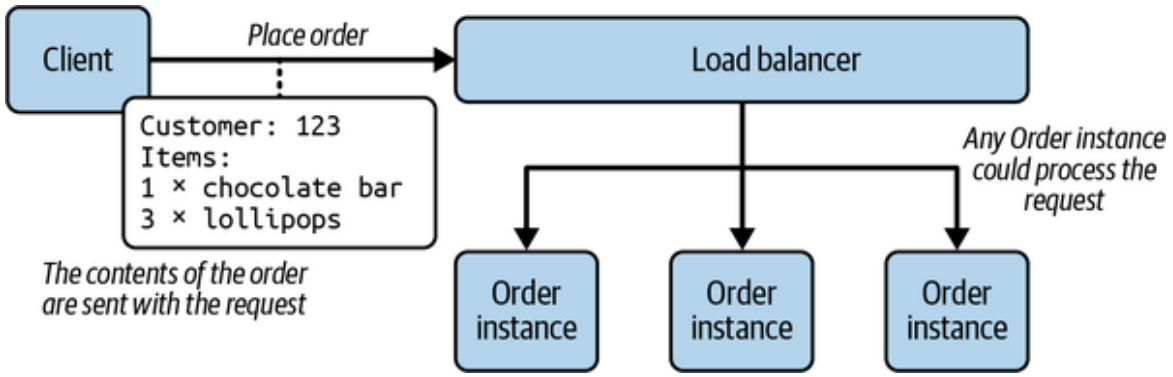


Figure 8-7. An order being processed in a stateless fashion, with all context provided in the request

Note that in the course of processing the work, a server is free to fetch required state from another location—such as by looking up data in a database. An instance is also free to cache data locally to speed up processing. The important point is that the state itself is not *required* to be available on an instance in order for processing to be completed.

Any instance of this service could handle the request. This means that load balancing is free to distribute the call to any available machine.

Stateful Processing

With stateful processing, some state is stored locally to a service instance between interactions. The processing of work requires accessing the local state that has built up throughout the interactions. Certain types of problems are well suited to stateful processing—examples include some types of online games, such as first-person shooters, and real-time analytics systems. If you are making use of the distributed actor model as provided by frameworks like Akka, then you are also making use of stateful processing.

As state is held in memory on a machine, the work has to be routed to instances where this on-instance state is available. In Figure 8-8, we see the order scenario from before, but now the request to place an order includes only a user ID. The shopping basket contents for that user are stored on the instance itself. As such, when placing an order, the request needs to be routed to instance C in this case. If the call was routed to instance A or B, the order could not be placed.

This creates two problems. Firstly, how is work routed to the correct instance? Secondly, what happens if instance C goes offline and the state local to the instance is therefore lost?

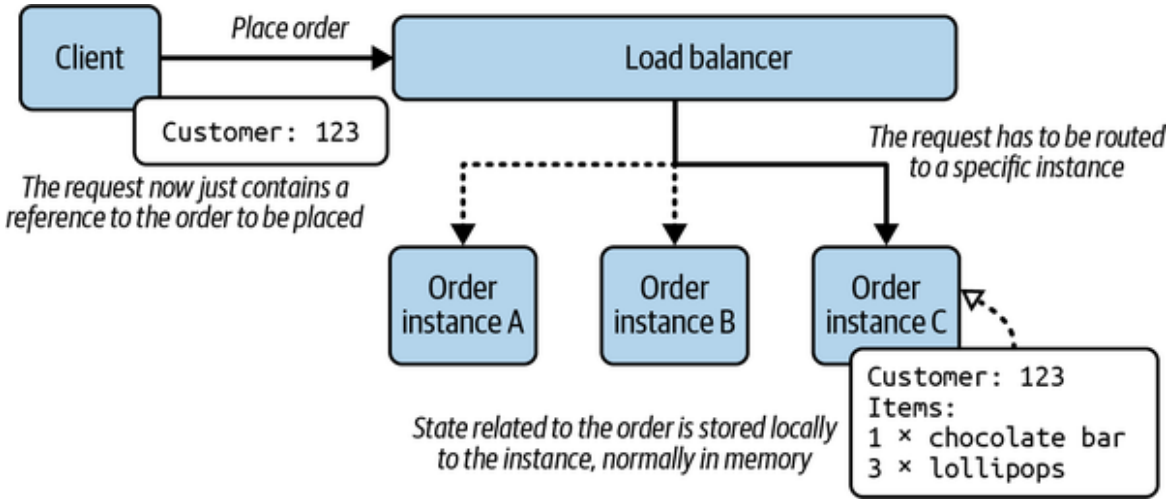


Figure 8-8. Order processing that is stateful, with the order context being stored in memory on the machine

Work routing

With a load balancer, one of the most straightforward work routing options for stateful processing is achieved using what are known as *sticky sessions*. When the original interaction is made, some form of token is sent to the client. Subsequent calls from the client will provide the token, and the load balancer will be able to use the token to route the request to the appropriate node.

If using a message broker for routing work, you'll need similar capabilities. You may find yourself having to instruct the broker to deliver a message to a specific consumer instance, which can be nontrivial depending on which message broker implementation you are using.

The increased complexity required to distribute work will at the very least complicate the configuration and testing of your work distribution mechanism, and it could also reduce the list of technology you are able to make use of. But the bigger issue is what happens if the instance with the required state is unavailable.

Handling instance failure

Coming back to the example in Figure 8-8, imagine what would happen if the service instance that held the customer basket crashed before the customer checked out. If the basket contents are held only locally on that machine, then the machine failure could result in those basket contents being lost.

This problem has two solutions. The first is to change this to stateless processing by moving the state out of the instance. A simple way to achieve this would be to move the basket contents to a database instead. Even if the service instance failed, the basket contents could still be retrieved as long as the database is still available.

If the move to stateless processing is not practical, perhaps because there are some other benefits of stateful processing that you want to take advantage of, then you need to have some mechanism to ensure that in-instance state is replicated across one or more other instances. This means that in the case of an instance failure, an alternative instance could pick up the work.

On the face of it, this seems like a great solution. However, we have to deal with the twin realities of latency and increasing complexity that come from replicating state. In Chapter 9, we'll look at this more in terms of how message brokers replicate messages internally to show you some of the challenges involved, but fundamentally, when replicating state across nodes, you have to either take a latency hit on writes or else accept that an acknowledged write can still lead to data loss.

General guidance

The technology to perform replication of state across instances does exist, but using it adds more complexity to your system—and with more complexity comes more risk. We again see the contradiction that introducing technology to make our system more robust can actually make it more vulnerable through new failure modes. And such technology cannot

change the fundamental challenges of a distributed system—latency is a thing, and instances will go offline.

Because of these challenges, I strongly recommend adopting stateless processing wherever possible. You'll find it much easier to build and scale these systems, not to mention the fact that you'll find it easier to maintain good availability of the systems themselves.

> **STATELESS OR STATEFUL?**
>
> From the point of view of simplifying failure scenarios, stateless processing is strongly recommended. If you are considering a stateful processing solution, make sure you *really* do your homework, and make sure that you fully understand how your chosen solution will react to machines failing and what impact state replication will have on latency and resource use.

Testing Machine Failure

There are a few techniques you can try to verify whether your attempts to deal with machine failure have worked.

If you already have some load tests to run against your system (or even an individual service), you can run those tests while taking down individual machines. Assuming you can provision machines using an API, then that same API can be used to remove machines too. This would allow you to validate whether your service can continue to operate. You could script experiments like this yourself, but you could also consider using an appropriate chaos engineering toolchain to create machine outages.

I mentioned Stability Sim previously; this in-browser simulator can also be used to experiment with the impact of machine failure in different situations.

Summary

Computers fail. We do our best to reduce this failure, but it will still happen. As we've seen in this chapter, there are a number of things you can do to make sure that your system can keep operating even when a single machine goes offline.

Having a service discovery solution that can be dynamically updated when a machine fails can be incredibly useful. Load balancing, both client-side and server-side, is readily accessible because of software networking solutions. When a machine does fail, desired state management can effectively provision a replacement. But how we design our solutions is also key to making failover as easy as possible—for this reason, you may want to think twice about stateful processing.

In Chapter 10, we'll see how to handle the failure of entire data centers. Before that, though, we're going to see the role that message brokers can play in making our interprocess communication more robust.

---

1 A quirk of the Raft algorithm is that odd numbers of nodes provide the sweet spot for fault tolerance and efficiency. Running four nodes, for example, provides the same fault tolerance as running three nodes but uses an extra node.

2 Note that this example has been edited for brevity.

Chapter 9. Message Brokers

Message brokers are often used in a distributed system, as they can help improve the reliability of communication between processes. That said, they aren't a silver bullet for reliable communication. Understanding the basics of how brokers work can be very helpful in making sure you're using them effectively. In addition to reviewing those basics, we'll explore a few complementary patterns in this chapter that will further help improve the resiliency of your broker-based communication.

Message Broker Basics

A *message broker* is a software-based system that acts as an intermediary in interprocess communication. Rather than a client and server establishing direct communication—for example, via an HTTP connection—messages are sent to the broker, which in turn delivers them to the intended recipient.

Vendor terminology can vary, so I'll use more generic terms here. In general, as shown in Figure 9-1, a broker sits between *producers* (which create messages) and *consumers* (which receive messages). These messages are routed via message channels.

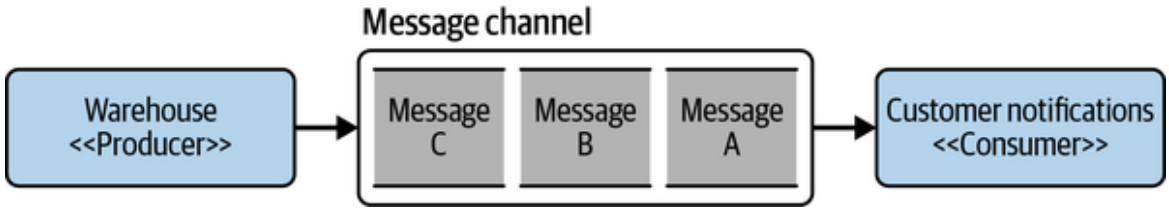


Figure 9-1. Message broker core concepts

In general in this chapter, we'll refer generically to the "message" as the thing that is created by a producer and then received by a consumer. A message, though, could contain many things. For example, a message could represent an event for event-driven interactions, but in the context of

request-response interactions, a message could be either a request or a response.

Queues Versus Topics

When it comes to message brokers, you will hear the terms *queue* and *topic*. Some brokers provide queues (e.g., RabbitMQ), some provide topics (e.g., Kafka), and some provide both (e.g., ActiveMQ).

Historically, the difference between queues and topics was more significant. Queues were designed for point-to-point communication, while topics were designed for broadcasting messages to multiple consumers. In reality, the distinction between these two has blurred significantly, with many queue-based brokers quite happily handling message broadcasts, for example.

This is why I've decided to use the term *message channel* to refer generically to a mechanism provided by a broker to get a message from a producer to a consumer; this could refer to either a queue or a topic, depending on which technology you are making use of. The term *message channel* comes from the book *Enterprise Integration Patterns* by Gregor Hohpe and Bobby Woolf,[1] and where appropriate I've used other terms from that book. If talking about a particular broker's implementations, I will use that broker's more specific terms.

Point-to-Point Versus Broadcast

A broker can be used to establish a message channel between a single producer and a consumer, as shown in Figure 9-1. However, most brokers also support a single message channel being consumed by multiple different consumers. In Figure 9-2, you can see three consumers, all of whom will be able to consume the same messages. So each consumer will be able to consume message A, message B, and so on.

In the case where multiple consumers consume from the same channel, the broker will normally keep track of which consumer has seen which message.

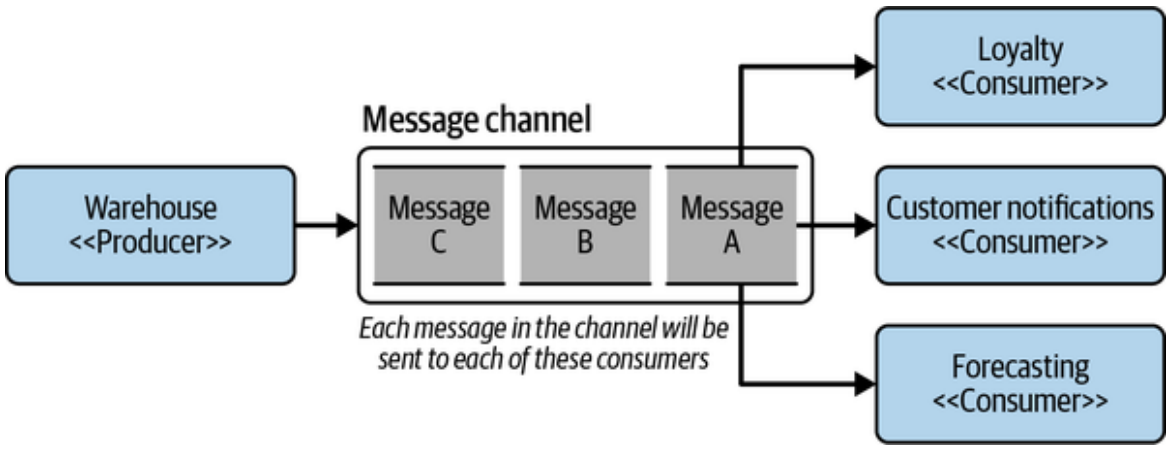


Figure 9-2. Multiple consumers can receive the same messages from a single producer

Consumer Groups

If a consumer is unavailable, then a message cannot be delivered (although the broker should keep hold of the message until the consumer is available again, which is something we'll explore more shortly). This is problematic, as it means a consumer going offline will result in messages not getting processed. To deal with this, some brokers support the concept of consumer groups. This means that multiple consumers can be grouped together, and the broker will ensure that a message is delivered to one of them, as shown in Figure 9-3.

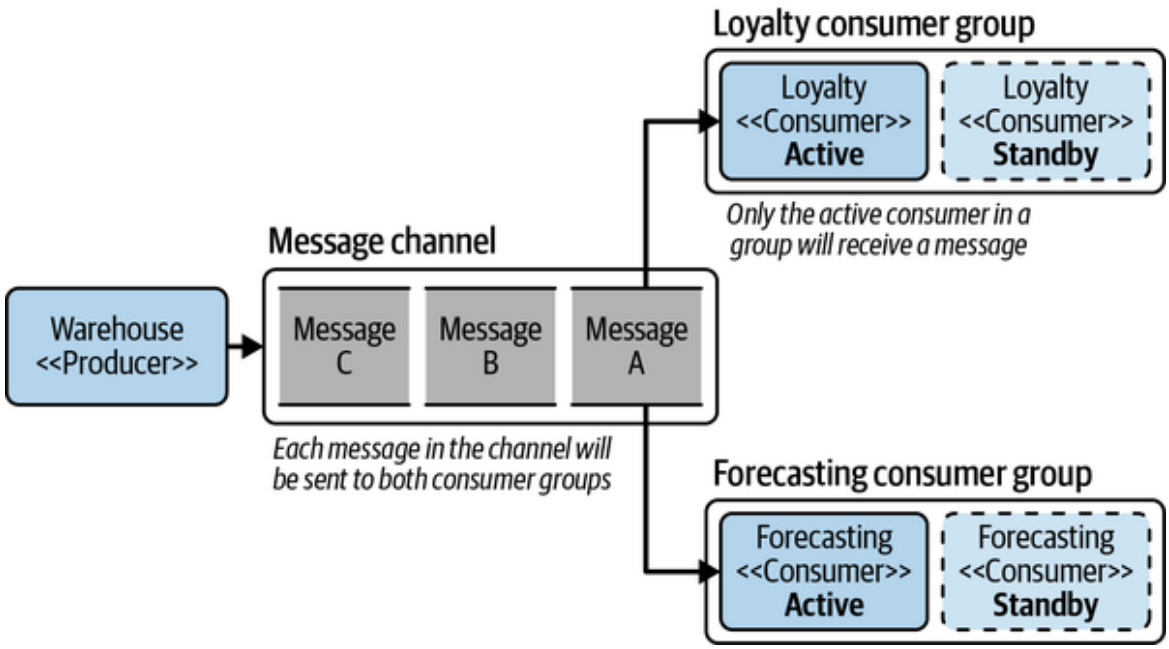


Figure 9-3. Consumers can be grouped, adding redundancy to hedge against the risk of a consumer instance failing

In such a situation, one member of each consumer group will be the active instance and will be able to consume messages. If the active instance goes offline, then one of the other nodes will be promoted to be the active node. For brokers that natively support this feature, this is normally handled by the broker and the associated client libraries.

If you are using a broker that doesn't support consumer groups, then you will be responsible yourself for ensuring that you can cope with consumer instances being unavailable. Kafka is an example of a broker that supports consumer groups out of the box. RabbitMQ, on the other hand, doesn't natively support this concept.

Why Use a Message Broker?

A message broker can provide a number of potential benefits, but we'll just focus on those aspects that are most interesting from a resiliency point of view.

Temporal Decoupling

In the context of a distributed system, *temporal coupling* refers to a situation in which both the client and the server must be available and be able to communicate with each other for an operation to complete.[2]

This is common in situations where direct point-to-point network connections are established between client and server—for example, when making use of an HTTP-based API. So in Figure 9-4, where the `Order` service is sending an HTTP request to the API of the `Warehouse` service, `Warehouse` needs to be available for the request to be received.

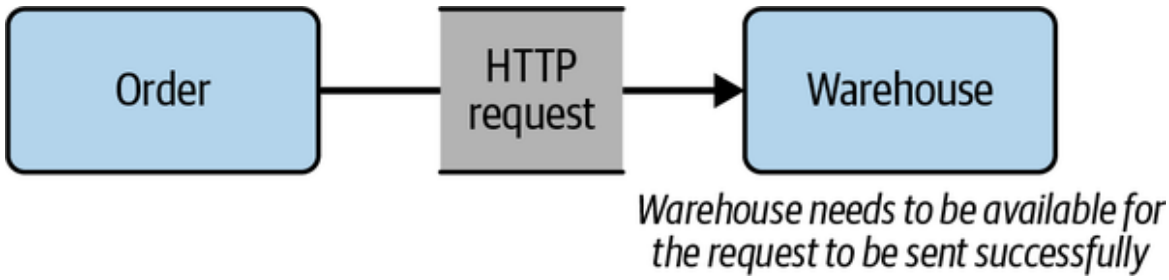


Figure 9-4. Temporal coupling occurs when the client and server need to be available at the same time for an operation to complete

With a broker, messages between a producer and consumer are relayed via the broker itself. If the consumer is currently not available to consume the message, the broker holds on to it until the consumer is ready. In Figure 9-5, we see the same scenario as before, but now the request is being sent as a message via a message channel.

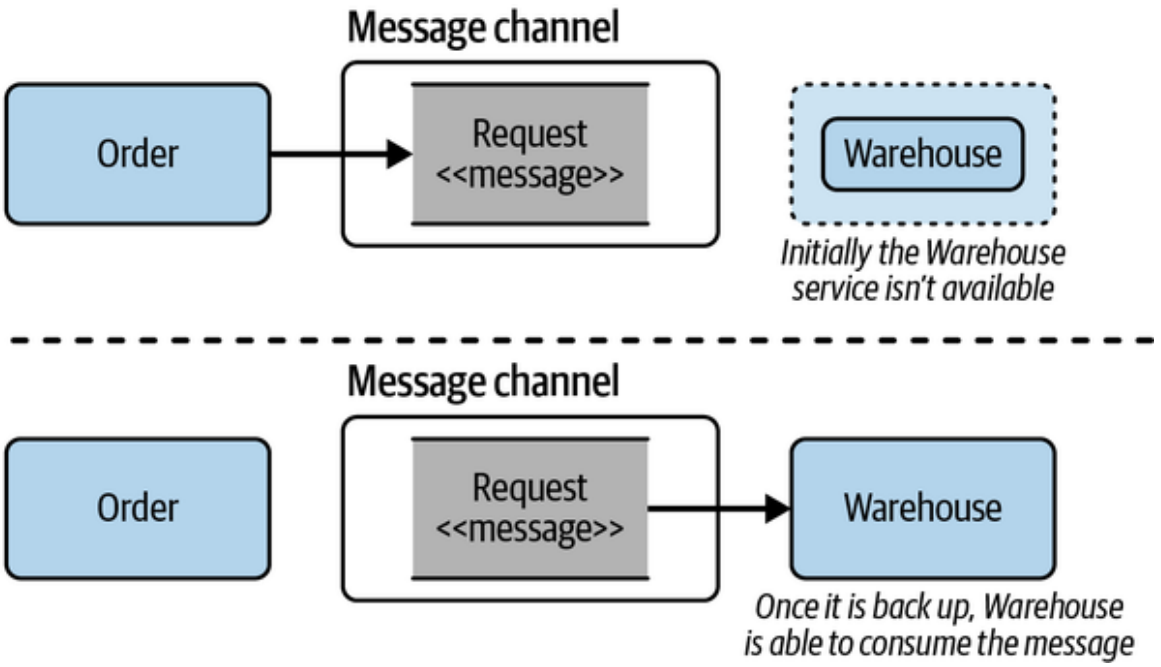


Figure 9-5. By using a message broker, temporal coupling can be avoided, as the consumer does not need to be available when a message is sent

This separates the producer from having to worry about the availability of the consumer at the point that the message is sent, but it does, of course, shift the concern to the broker itself, which we now need to provide some level of guarantee around how the message will get delivered.

Delivery Guarantees

Once the broker acknowledges to the producer that it has received the message, it will then provide some level of delivery guarantee. The attraction here is the general concept of "once the broker tells me it's got it, it will sort delivery for me."

However, the exact nature of these guarantees can vary, and as a result this may change how you design your system and make use of brokers. Some brokers provide very weak (or even no) guarantees around delivery. While there can be good use cases for such brokers, we'll focus most of our attention in this chapter on brokers that can provide stronger guarantees.

Scaling

As we discussed in Chapter 5, systems can sometimes be overwhelmed by sudden, unplanned spikes in traffic. Brokers can work effectively as buffers, helping smooth out spikes in load on the consumer side. We can use a broker as part of a rate-limiting solution. If consumers process work at only a fixed rate (such as when messages are being processed sequentially, for example), then consumers will be protected from being overwhelmed. This can put pressure on the broker, of course, and some brokers are better than others at handling large volumes of messages, something we'll touch on more shortly.

Do be aware that if you are processing messages at a constant rate but the rate at which new messages arrive is increasing (perhaps because of a spike in load), then overall latency is increasing. Although the time to process a single message may not change, messages may sit in a queue for longer periods of time before being picked up. This touches on a concept known as Little's Law, which we'll look at in more detail in Chapter 12.

TIMEOUTS, RETRIES, AND MESSAGE BROKERS

Given that we put the broker in charge of message delivery, you might be forgiven for thinking that this means that you don't have to worry about timeouts and retries. That is partly true. The broker will make use of timeouts and retries within the broker for message replication and also when delivering a message to a consumer.

However, if a producer wants to send a message to a broker, retries and timeouts will also be needed when establishing communication between the producer and the broker itself. In practice, the broker client library will normally have its own retry and timeout policy, so you don't need to implement one yourself. You may, though, have access to tweak the configuration of this policy.

How Brokers Work (in a Nutshell)

So a producer puts a message in a message channel, which is managed by the broker. The broker ensures that the message is kept safe (how this is done varies from broker to broker) until the consumer is ready to receive it. It all seems so simple! However, the detail of how a broker does this is more complex than it might appear at first glance.

The broker has two key challenges. Firstly, how can it maintain high degrees of availability? If the broker goes down, all interprocess communication over the broker will cease. That's a pretty major problem. Secondly, how can messages be held in a durable fashion to ensure they don't get lost?

While brokers vary in the way they solve these issues, fundamentally, both availability and durability are normally achieved by having the broker run across multiple machines as part of a cluster. The aim is that the loss of a machine won't result in either the loss of a message or loss of availability of the broker itself.

Exactly how each broker cluster is run and managed can differ greatly. Managing a RabbitMQ cluster is quite different from managing a Kafka cluster, for example. However, in each case, the cluster itself will have requirements around how it is run and operated that will need to be carefully adhered to if you want the promised guarantees. This adds to the complexity around operating a broker, and this is one of the key reasons that I tend to recommend fully managed broker solutions, where experts can do some of the hard stuff for you.

RABBITMQ AND KAFKA—CLUSTERS, REPLICAS, AND PARTITIONS

Again, terminology can differ across vendors when describing the collection of machines used for replicating data and maintaining high availability.

For RabbitMQ, replication and high availability are provided by having multiple instances in a single cluster. In Kafka, message durability and high availability are provided by replicas operating as part of a partition.

Fundamentally, RabbitMQ and Kafka are implemented quite differently, resulting in a different set of trade-offs for end users. In this chapter, I am focusing primarily on how these differences play out in terms of how you use the broker, rather than offering a deep dive into how brokers are implemented.

To keep things simple, when describing a cluster with respect to replication of messages, this can be considered the same as replication happening within the instances in a Kafka partition.

Maintaining Durability

To ensure that a message isn't lost when a broker instance fails, the broker will replicate a message around nodes in the cluster. When a producer then writes to the broker, a decision has to be made about what needs to happen before the broker write can be acknowledged.

One option would be to acknowledge the write once the first broker node has safely written the message to its local store, with replication happening later, as shown in Figure 9-6. This also means that the write to the broker will happen faster, as the producer isn't waiting for replication. However, there is now the possibility that the message could get lost even after the initial write was acknowledged, as the node that the message was written to could fail before replication has taken place.

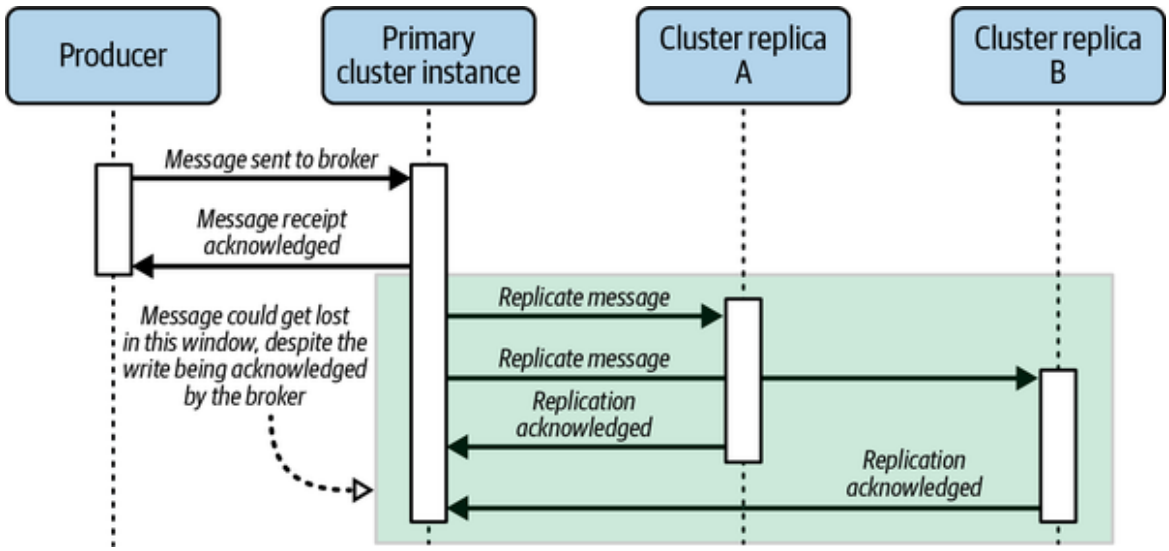


Figure 9-6. Acknowledging a write before replication can be faster for a producer but can increase the chance of message loss

Alternatively, the broker can wait until replication has taken place within the cluster before acknowledging the write. This could then result in a write to the broker taking longer from the point of view of the producer, as we see in Figure 9-7.

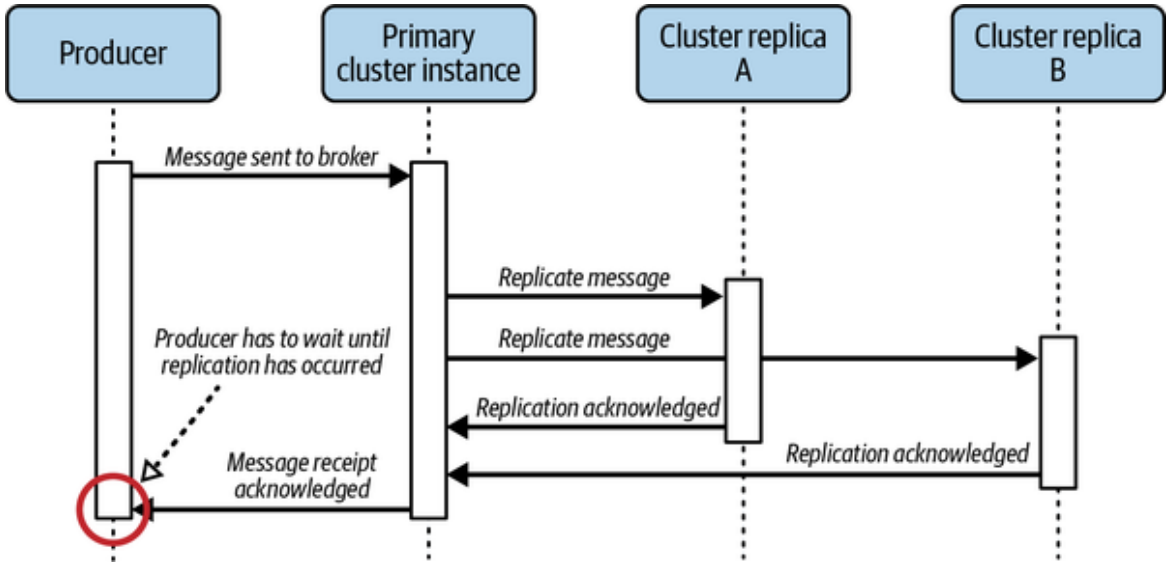


Figure 9-7. Waiting for replication before acknowledging a broker write can slow the producer down

For some brokers, the choice here is taken out of your hands, as the broker makes the decision for you. With RabbitMQ's quorum queues, for example, a write is acknowledged to the producer only after replication has taken

place. With Kafka, this is something that the producer is in charge of, allowing producers to make different trade-offs in different situations.

In either case, it's also worth understanding what happens if a broker replica doesn't respond in a timely manner. If a producer is blocking, waiting for replication to the cluster to complete, and one of the required replicas either is offline or doesn't respond quickly enough, then the broker may be able to alert the producer that there is an issue with the write. However, if the producer does not wait for replication, and replication within the broker cluster subsequently fails, the producer may be unaware that there is a problem. Exactly how such situations will be handled will vary from broker to broker. In Kafka, for example, if a replica was temporarily unavailable, once it starts responding again (or a new replica is added), Kafka can start replicating the required information.

It's important to pay attention to the broker's documentation relating to exactly how many instances are required, and how they need to be set up, to provide the durability guarantees you want. You will typically be given guidance on the minimum number of replicas, for example, and on the required network topology.

Maintaining Availability

In the event that a broker instance fails, high availability is achieved by ensuring that another instance in the broker cluster can take on responsibility for handling the receipt of messages from producers and the delivery of messages for consumers.

The replication of messages to improve durability can be done in such a way that it is mostly (or entirely) invisible to both consumer and producer, as everything happens inside the broker itself. However, things get a bit trickier with high availability setups. Fundamentally, a producer needs to know which machine to send its message to, and a consumer needs to know what broker machine to connect to in order to receive its message.

The most common solution for this is for the broker cluster to have some sort of leader election mechanism in place. This allows a node in a cluster

to be specified as a primary, where writes should be sent, with replication then happening from that node to other standby nodes.

If the primary node fails for some reason, a leader election mechanism then kicks in, whereby one of the standby nodes will be elected as a new leader. Again, how this is done depends on the broker implementation, but typically this election process is carried out using tried and tested algorithms designed for this purpose.

Both RabbitMQ and Kafka make use of the Raft protocol internally for this purpose.[3] A full description of how Raft works is outside the scope of this book; however, if you want to learn more, I can thoroughly recommend The Secret Lives of Data, an excellent (and interactive) website created by Ben Johnson.

As with providing message durability, pay close attention to how to set up and manage your cluster to provide the availability guarantees you want. At the very least, this typically requires having a minimum number of machines in a cluster.

Durability and High Availability Not Needed?

It's important to note that there can be a lot of use cases for which you might want a broker not to provide durable message channels. Generally speaking, if you reduce your durability requirements, then you can achieve much faster throughput. If the loss of a message can easily be tolerated, this approach can be sensible. Consider a signal being sent from a sensor, which is constantly sending a stream of information—if you lose one or two messages, that might be fine.

Although rarer, in some cases a broker doesn't need high availability either. An example might be development or test scenarios, where a broker might be spun up for nonproduction traffic.

Fleeting Messages

Once a message has been delivered to the last of the waiting consumers, most brokers will discard the message. This is because holding on to messages takes up computing resources, and as a result these resources can become more constrained, which can adversely impact the broker's behavior. For example, if messages are held in memory for faster access, memory could be exhausted as the number of messages increases. This would result in memory being swapped to disk, which can cause performance to degrade—and probably the worst time for a broker to start getting slower is when it's expected to handle more and more messages. So if keeping hold of lots of messages is going to slow down the broker, why not get rid of each message after it's been delivered?

Kafka is a rare example of where a message can be kept for an arbitrary length of time, even if it has already been delivered. So why might this be useful? From a broader perspective, there are a number of benefits. For example, newly deployed services would be able to consume messages that were sent earlier. Kafka also allows you to change how a previously processed message is handled. Imagine if you were using a broker to share data as part of a data processing pipeline, in which data is transformed prior to being published to an appropriate datastore for analytical purposes. If you wanted to change the logic involved in the transformation, you could just reingest the messages rather than having to get everything sent again (it's no surprise that Kafka was originally developed in part for use in the extract, transform, load [ETL] process).

From a resiliency standpoint, having a broker that can hold on to already processed messages could be handy. It could allow you to sidestep the need for a consumer-managed inbox, a pattern we'll look at later in this chapter. It can also give you the ability to recover from data loss. Imagine that a service has been happily ingesting messages all day, but then its database suffers a corruption and its data is lost. If the service could reingest the messages for the day, it might be able to recover.

Of course, holding on to messages creates a potential headache. These messages take up computing resources, and as we've discussed a lot throughout the book, we want to try to avoid consuming too many

computing resources. But not all resources are equal—some are (relatively) cheap and abundant. In the case of Kafka, messages are typically backed to disk. In fact, Kafka can be used with a variety of backing stores, including blob store services like Amazon Simple Storage Service (S3); this gives you another dimension to consider when balancing message durability against factors like cost and ease of operation.

Types of Delivery Guarantees

From a resiliency standpoint, one of the main selling points of a message broker is that it provides some level of guarantee that a message will be delivered to a consumer. In practice, the exact nature of a delivery guarantee can vary from broker to broker.

The first thing to understand is that a broker cannot guarantee to deliver a message if the consumer is never available. So with that being said, we're going to look in-depth at three types of delivery guarantees—at least once, at most once, and exactly once.

At Least Once

Most message brokers provide *at least once* delivery guarantees, meaning that there is a small chance of a message being seen more than once. This is far and away the most common delivery guarantee, and it's worth understanding why.

Message brokers push messages to a consumer when a consumer signals that it is ready to receive a message. This is normally handled via some sort of coding abstraction delivered as a client SDK over the broker-specific protocol. One of the challenges in delivering messages to a consumer is making sure the consumer got the message in the first place. If a message is pushed to the consumer, and the consumer acknowledges it received the message, then the broker can now remove the item from its channel—the message has been delivered.

If the consumer doesn't acknowledge receipt of the message, then the broker probably has to assume that the message was not picked up, and therefore the message remains in the channel. This creates a problem if the consumer processes the message but for whatever reason doesn't acknowledge receipt. This can lead to rare cases of the same message being delivered twice. At this point, I hope you're having flashbacks to our discussions regarding idempotency back in Chapter 4.

Now in many cases, the message broker consumer SDK may be able to filter out messages it knows to be duplicated without your involvement, but this is normally possible only if the same consumer instance sees the same message twice. In the scenario in which a separate consumer from the same consumer group consumes the message, the client SDK is unlikely to be able to help.

If you are making use of a message broker that provides at least once delivery guarantees, it is sensible to be prepared for receiving a message more than once. For operations that perform writes, making them idempotent as discussed earlier means you don't need to worry about these duplicated deliveries.

At Most Once

With *at most once* delivery, what is being offered is a guarantee that a consumer will never see a message more than once, though some messages might get dropped. Behind the scenes, the broker isn't waiting for an acknowledgment from a consumer that a message has been received. At first glance, this doesn't seem terribly useful. Who wants a message broker where messages sometimes get lost?

At most once delivery provides two key benefits. Firstly, you don't have to worry about a message being delivered twice. Secondly, as the broker doesn't have to wait for acknowledgment of message receipt and the like, delivery will be much faster. This is great for low-latency situations in which speed is critical but losing a message or two might be acceptable.

It might initially seem like the concept of at most once delivery doesn't have a place in the context of a system where you are trying to improve resiliency. However, remember that one of the enemies of system stability can be resource saturation. If you are able to make operations faster, then that can reduce contention on system resources, as the overall amount of work your system is handling at any given moment is also reduced.

Exactly Once

To say that *exactly once* delivery guarantees from a broker vendor are a complex topic would be something of an understatement. On the face of it, an exactly once guarantee is incredibly desirable. A broker giving a cast-iron guarantee that each message will get delivered and that you don't ever have to deal with duplicate messages is an attractive option. As we dealt with in Chapter 4, retrofitting idempotency into existing client-server protocols can be difficult, so I understand why having a turnkey solution to solve that challenge would be attractive.

The devil here is very much in the details. Firstly, many of the brokers that claim to provide exactly once delivery differ in how they describe what "exactly once delivery" is. For example, Google Pub/Sub guarantees exactly once delivery only if, as Google's documentation says, the consumer "acknowledge[s] messages within the acknowledgment deadline." If the consumer acknowledges the message late, it could be delivered twice. So a big chunk of the burden on delivery here is on the consumer side—meaning this is not a magic "set and forget" option.

In Kafka, the concept of exactly once delivery is further complicated by a Kafka client's ability to "rewind" its messages. Each consumer has an offset, which points to where that consumer is up to in the topic. A consumer is able to change its offset, enabling it to reread old messages. This can be useful in a lot of circumstances, but making such a change would invalidate exactly once delivery. There are a number of other caveats around how Kafka achieves exactly once delivery, and I'd strongly recommend you read the associated documentation to understand the nuances.

The Google Pub/Sub and Kafka flavors of exactly once delivery share a common trend. Both rely on the producer and the consumer working together (or more correctly, behaving within a strict set of parameters) to deliver on their respective exactly once promises.

One final consideration is that, because of how exactly once delivery is implemented, you should expect a performance hit. For example, producers will have to wait until the broker confirms that the message was received and that all required internal replication of the message has taken place. So there could be an undesirable knock-on effect in terms of resource contention as a result.

As someone who has spent a lot of time looking at design documents and has even spoken to the vendors themselves in some cases, I am certain that a lot of smart engineering work is going on inside these systems. However, the solutions they come up with don't deliver an entirely turnkey experience, in my opinion. Arguably, brokers providing exactly once delivery are a leaky abstraction—there are enough caveats around exactly once guarantees that a developer making use of such a broker still has to be aware of the nuances; otherwise, the unexpected might occur. For this reason, I would strongly recommend explicitly building idempotency into your application where it is required, rather than relying on a vendor to solve the problem for you. Aside from requiring a deeper understanding of a specific broker's implementation, this gives you more flexibility around changing your broker in the future, as the required semantics around idempotency are built into your code rather than residing in a vendor's product.

> **IDEMPOTENCY**
>
> Build idempotency into your application rather than relying on a vendor product to do this for you.

Guaranteed Delivery Versus Guaranteed Processing

Message brokers will provide guarantees around delivery, as we have discussed. However, it is important to know that guaranteeing that a message will be delivered is not the same thing as guaranteeing that the consumer processes the message. The broker is responsible for delivery; the consumer is responsible for processing.

If a consumer picks up a message off the channel but then immediately crashes, then the message was delivered but not processed.

So in situations where you need to know whether things have actually happened, you should provide an acknowledgment that a message was processed. Having the consumer deliver a confirmation of processing (likely via the transactional outbox pattern, which we'll look at shortly) would be one way of doing this.

Useful Patterns

This book is all about making sure that as much stuff as possible continues to work even when things go wrong. We've already looked at what the brokers themselves can do to deal with various failure modes. But due consideration also needs to be given to how our own code interacts with brokers.

With that in mind, we're going to look at four patterns to add to your toolbox that can be helpful in improving the resiliency of interprocess communication when using a message broker:

Transactional outbox

> Ensures that messages can be sent more reliably as part of producer-side write operations

Consumer-managed inbox

> Provides stronger message-processing guarantees by separating message receipt from message processing

Producer heartbeat message

Ensures that a message channel is open

Dead letter channel

Provides a quarantine for messages that cannot be processed

Pattern: Transactional Outbox

A store for messages that are yet to be sent over a broker, managed within producer-local transactional boundaries.

When a producer sends a message as a result of some local write operations, problems can occur if the message fails to send after the write has completed.

Imagine a scenario in which an `Order` service wants to send an event to let other services know that the state of an order has been updated. If the order is changed in the `Order` service's database, but the `Order` service crashes before the event can be sent, then the system may be missing a vital piece of information.

As outlined in Figure 9-8, when a transactional outbox is being used, outbound messages are stored locally in the same transactional boundary as the local write. After the local transaction completes, the pending messages can be sent as part of a separate operation.

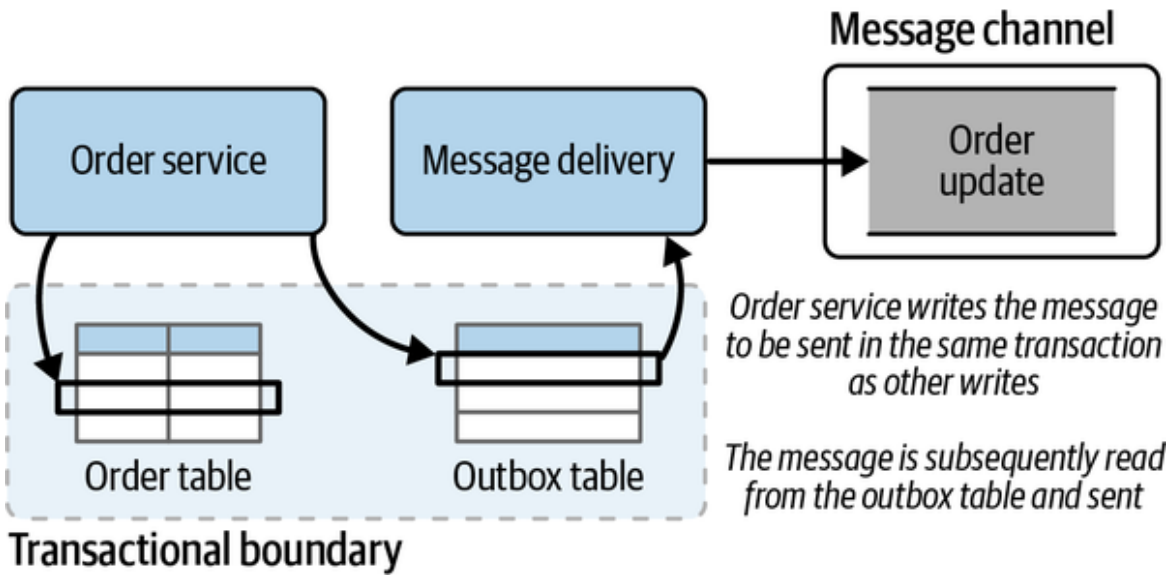


Figure 9-8. The transactional outbox pattern

How it works

Figure 9-9 breaks down the example in Figure 9-8 to give you a clearer idea of the sequence of operations.

During the database update, if an error occurs, the transaction will be rolled back. No updates will be made to either the order or outbox table. That means we get to take advantage of the atomic nature of a database transaction—either the writes are made and a new outbound message is stored, or none of that happens.

If the `Order` service crashes after the write, as long as the database transaction has been written, the message is safely stored in the outbox table.

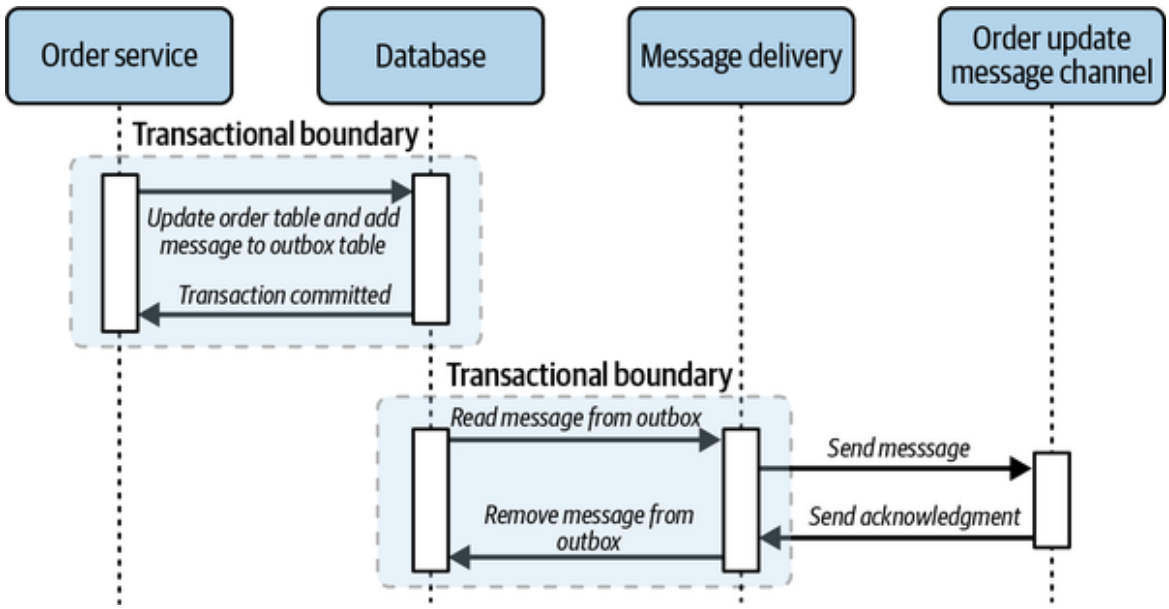


Figure 9-9. The sequence of operations when using the transactional outbox pattern

Failure in message sending

The eagle-eyed among you may have spotted a potential problem. If a message is sent over the broker but then the sending process crashes before the message is removed from the outbox table, the message would remain in the outbox table and might get sent again later. While this particular set of circumstances might be rare, it's important to note that this possibility exists.

If, after giving the broker a message, you then decide to send the same message again to the broker, it will in all likelihood appear as a *different* message. This means that any safeguards in place in the broker client libraries to avoid duplicate sending of messages would likely not apply. It also means that even if you are using a broker that supports at most once delivery guarantees, the use of a transactional outbox reintroduces the possibility that a message might get sent twice.

The net result is that the use of the transactional outbox pattern means that you are signing up for at least once delivery guarantees, with the caveats outlined earlier in this chapter. If you wanted to make sure that the message isn't *processed* more than once, you'd need to make processing idempotent,

either by fingerprinting the message on the consumer side or by having the producer send an idempotency key.

Retaining sent messages

In the simplest implementation, once a message is sent, it would be removed from the outbox. If you needed the ability for a producer to resend messages, however, it might be beneficial to keep the message instead. This could be done by adding a flag to entries in the outbox table to denote that this message has been sent, or else by moving the message to some other table.

With high volumes of messages, retaining messages would cause some headaches, of course. Additionally, any mechanisms to cause the messages to get resent would also have to be implemented.

And of course, if you're using a broker that natively supports retaining already delivered messages, such as Kafka, then you won't have to worry about keeping old messages in your outbox.

> **AUDITING MESSAGES**
>
> Note that if you just want to audit which messages were sent, keeping the messages in the database is likely overkill. Logging would likely be more appropriate.

When to use it

In situations where it is important that a message is sent, especially in relation to a local database write, the transactional outbox pattern is a simple solution to provide stronger guarantees that the message will actually get sent. A common example would be an instance in which events are being sent in relation to state changes in a service boundary, such as in the order example noted previously.

Pattern: Consumer-Managed Inbox

A consumer-managed store of messages that have been delivered but have not yet been processed.

When a consumer receives a message from a broker, the consumer could fail during the processing of the message. This can leave you in a state where the message has been removed from the broker (and therefore has been delivered) but has not yet been processed by the consumer.

With a consumer-managed inbox, as outlined in Figure 9-10, a consumer keeps its own record of messages that have been received but not yet processed. This ensures that if a consumer crashes while processing a message, the message won't get lost, and processing of the message can be reattempted.

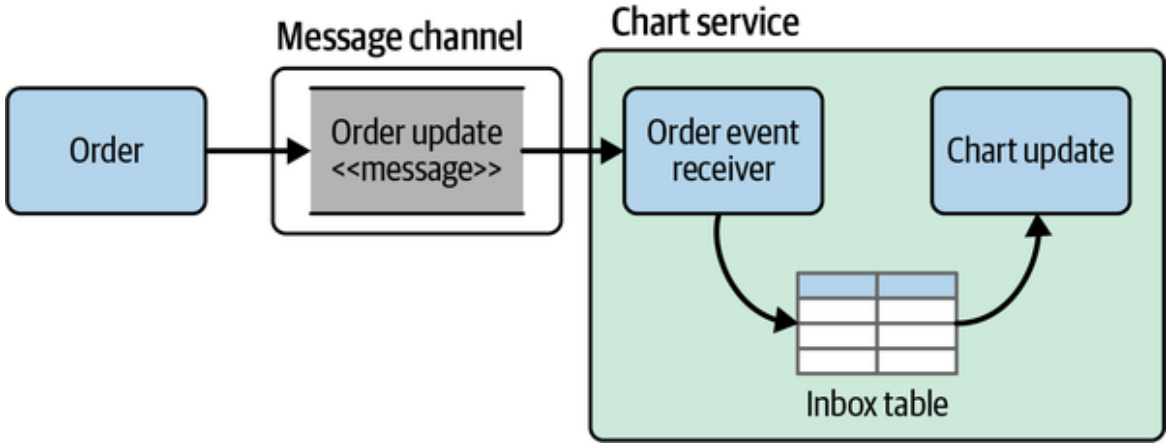


Figure 9-10. The consumer-managed inbox pattern

In effect, this pattern more formally separates message *delivery* from message *processing* from a consumer point of view.

INBOX VERSUS CONSUMER-MANAGED INBOX

This pattern is often referred to more simply as the *inbox* pattern. I've always felt that this was a bit too generic of a term. The important aspect for me is that the consumer has control over the inbox. When describing this pattern, I find that some people get confused, as at first glance the brokers themselves are providing a place where messages are being stored—so why do we need a second place to store them? Thus I feel it's important to make clear that the inbox has to be under the control of the consumer.

How it works

The most common implementation of the consumer-managed inbox is to store messages in a database table as they are received.

Picking up our example from Figure 9-10, we have our `Order` service, which is firing an event in reaction to an order being placed. The `Chart` service wants to react to this order placed event so it can update its listings around the most popular products.

In Figure 9-11, you can see the full sequence diagram outlining how this pattern is implemented. As events arrive, they are written to the inbox table by the order event receiver process, and once the write has been committed to the database, the receipt of the message is acknowledged to the broker, so the broker knows that delivery has taken place.

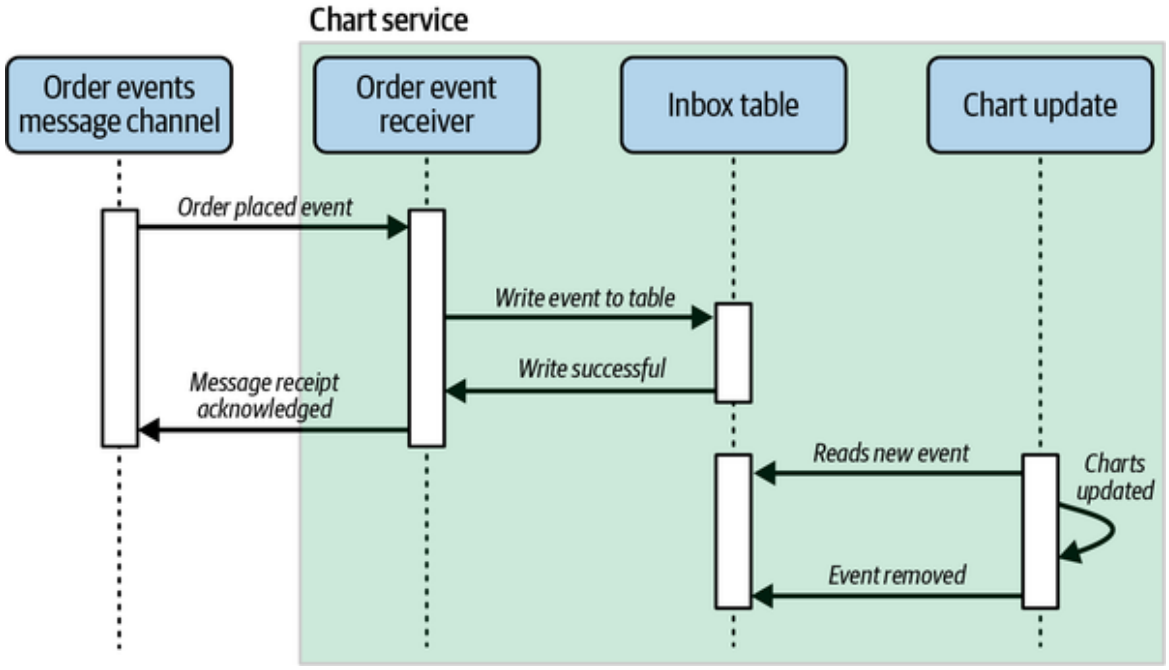


Figure 9-11. A sequence diagram representing the use of a consumer-managed inbox

Next, the chart update process starts a database transaction and reads the unprocessed event from the inbox table. Once the event has been processed, it can be removed from the inbox table, and the database transaction can be committed.

If an issue occurs during processing of the event, the database transaction being used to update the inbox table can be rolled back, and the event will remain in the table, allowing it to be processed again later.

I'd consider both the order event receiver and chart update processes to be hidden implementation details of how the `Chart` service works—they'd be invisible to external consumers of the `Chart` service itself. These processes could just as easily be different threads running in a single "chart" process.

REPLAY VERSUS REMOVAL

If received messages are removed from the inbox after being processed, then the size of the inbox can be kept to a minimum. In addition, the inbox itself is quite simple—things in the inbox are things that have to be processed.

On the other hand, if the consumer-managed inbox retains messages after processing, this allows the consumer to potentially replay previously received messages. Let's imagine that when the `Chart` service received an order placed event, there was a bug, and a set of items being sold was assigned to the wrong category, thus breaking the best-selling charts. If you still had copies of the received events in the inbox, you could fix the `Chart` service, redeploy it, and reingest the messages.

If the messages were not retained by the `Chart` service, then to fix this problem you may have to get the `Order` service to resend the events, which may or may not be easy to do.

From the point of view of this pattern, the main trade-off here is around managing inbox size versus the ability to replay messages. Note, however, that the consumer will need to have some capability to replay messages appropriately—if the goal is to behave differently when reprocessing an old message, then you need to ensure that any safeguards you have around things like idempotent processing are taken into account. In the case of reprocessing an old event to fix a bug like the one in this example, you wouldn't want these messages to subsequently be ignored because they had been seen before!

When to use it

In situations where you need to provide stronger guarantees around message processing, this pattern allows you to reattempt message processing if it initially fails, without you requiring that the message be sent again.

In a situation where the message can easily be redelivered, a consumer-managed inbox may not be required. As an example, Kafka gives consumers the ability to reingest previously consumed messages, as we discussed earlier.

KAFKA OFFSETS

Kafka consumers can reingest messages by changing their *offset*. For each consumer,[4] Kafka stores a pointer to the message that consumer last processed. This pointer is the offset—it's literally an integer pointing to the message in a topic. A consumer can ask the broker to change the offset, which can then cause previously delivered messages to be sent again (assuming these messages are still held ini the topic).

Pattern: Producer Heartbeat Message

A regular message sent by a producer to let the consumers know that the message channel is working and that the producer is capable of sending messages.

When a consumer is expecting messages from a producer but no message arrives, what is the consumer to think? One possibility is that no message has been sent, in which case all is well. The second possibility is that a problem has occurred with the producer or the broker itself, which means that messages are not being propagated successfully. Consider a typo appearing in the name of a channel when a new version of the producer is deployed, or the producer being offline for a prolonged period of time. In such cases, the consumer would be in blissful ignorance, unaware that anything had gone wrong.

A producer heartbeat message is a special type of message delivered at a regular interval over a message channel to let the consumers know that the producer is still capable of sending messages.

How it works

This is typically a simple pattern to implement. All you need is some way to send a message on a regular interval. On a Java application, for example, I would have a dedicated thread in the producer responsible for this.

On the consumer side, there would have to be expectations set around how frequently this message can be expected. For example, a consumer might expect to see a heartbeat message every 30 seconds. The message itself can be very simple. Its arrival is typically enough for a consumer to assume that all is well. But you could also use the heartbeat message to send other useful information.

For example, if the heartbeat message contains the time the message was sent, then the client could get some idea of how long it's taking for messages to propagate by comparing that time with the time the message was received.[5] In a system where you may need to measure the impact of eventual consistency, this information can be useful. You could send the propagation time for each heartbeat message to your observability tool of choice to see how this varies over time, for example.

When to use it

A heartbeat message is very useful in situations where there will be lulls in messages being sent from producer to consumer as part of normal operations.

If messages were regularly being sent over the channel, a consumer could infer that the lack of messages was likely a cause for concern. But in a situation where there are regular prolonged gaps between messages during normal system operation, the consumer will find it difficult to know whether there is an issue.

BROKER INTERNAL HEARTBEAT MESSAGES

Many message brokers will internally make use of heartbeat messages inside client SDKs. For example, in Kafka, clients can send a consumer heartbeat to let the broker know that the consumer is still available to receive messages. RabbitMQ makes use of heartbeats on both the producer and consumer side to keep connections open.

Just because you know that your consumer is connected to the broker doesn't mean that the producer is properly sending the messages that it should be.

These types of heartbeat messages are more about the broker knowing which producers and consumers are attached and represent something different from what we are trying to achieve with a producer heartbeat message. For example, a configuration bug with a producer that results in messages going to the wrong channel wouldn't look like an error to the broker. Or consider a transactional outbox where messages are being stored to be sent, but the process responsible for actually sending them has crashed and hasn't recovered.

Pattern: Dead Letter Channel

A place where messages that cannot be delivered or safely processed can be quarantined.

If a message can't initially be delivered, continual attempts to redeliver it can cause problems. But just throwing it away denies you the ability to interrogate the message to understand what has gone wrong.

As outlined in Figure 9-12, when a message cannot be delivered to a consumer, it is instead diverted to a *dead letter channel*, so that it will be quarantined from the system but can still be inspected.

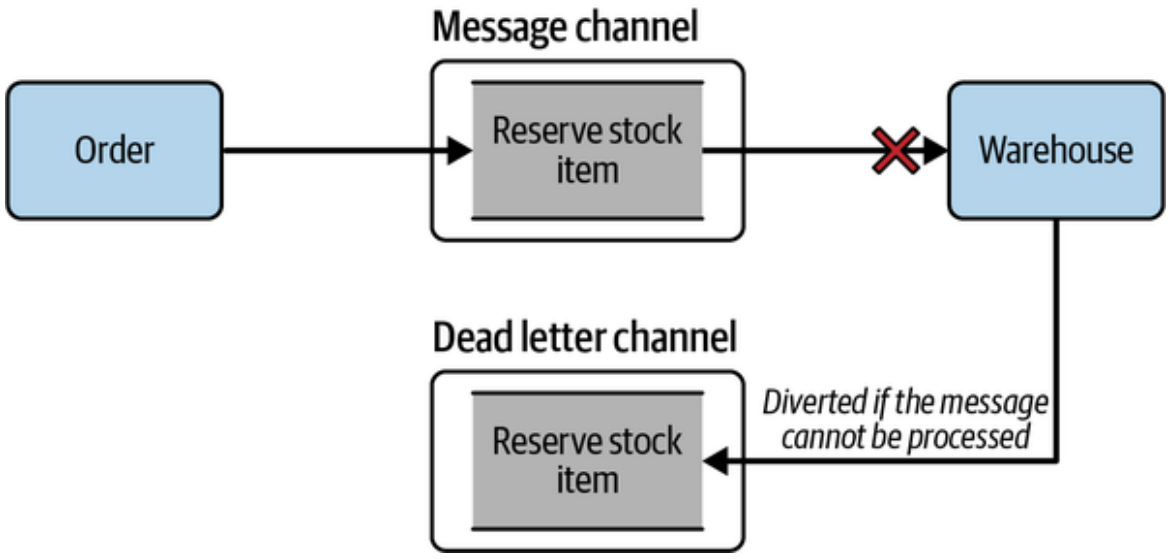


Figure 9-12. A dead letter channel

Once in the dead letter channel, the message could be investigated and triaged by either an operator or an automated process.

So under what circumstances would a message be unable to be delivered? One example might be a message sent in a format that the consumer cannot parse—perhaps because of a backward-incompatible change in the message format. A more prosaic example would be a misconfiguration resulting in the wrong messages being sent over an inappropriate channel. Or worse yet, a message that causes a consumer to crash when it attempts to process the message.

From a resiliency point of view, messages that remain in the main message channels and cannot be delivered are a headache. They could cause further failed processing (and the associated errors), fill up channels (causing delays in processing), and ultimately tie up resources that could be better used elsewhere.

By placing the problematic message in a dead letter channel, you keep the main channels free but allow the message to be properly assessed to understand what caused the issue.

How it works

The circumstances that would cause message delivery to fail can be determined only in the consumer itself. If a message format is invalid, or if expected fields are missing, it's the consumer that can work that out. Therefore, the consumer is typically responsible for diverting messages to the dead letter channel.

It's important to note that messages should not be sent to the dead letter channel for standard error conditions that are part of a normal protocol. For example, looking again at Figure 9-12, if the `Warehouse` service rejected the reservation request because that item was no longer available and then the message was processed, it just resulted in an error, and you could imagine a matching message being sent back to the `Order` service.

When to use it

If a message has repeatedly been rejected by a consumer, leaving it in the message channel could cause further problems for other consumers. Rather than just deleting the message altogether, diverting the problematic message to an alternative location keeps the message channel flowing and allows the message to be viewed by an operator to help diagnose what has happened.

A viable alternative is to just throw away problematic messages rather than store them. This can deny you the chance to view or retry the messages, but if you are dealing with very large message volumes (especially in situations where message loss can be tolerated), then this approach might be appropriate. You could still log some useful information to allow a human operator to investigate what happened later, of course.

There is a related pattern, an extension of the dead letter channel pattern, that is known as the *message hospital*. This pattern introduces the concept of triaging messages, with a view that they could be put back into the normal message channels. It's a pattern I've used only rarely, and in any case, you'd start with the dead letter channel pattern when implementing it.

Self-Hosted Versus Fully Managed Brokers

Message brokers can be complicated things to set up and manage. I've often seen complaints from people regarding message loss being caused by a buggy broker implementation, only to discover that the broker wasn't configured properly or was running in an inappropriate environment. RabbitMQ, for example, has fairly specific requirements around latency between nodes in a cluster; if you were operating in an environment in which intermachine latency varied greatly, you would have to take that into account.

It's not just the initial setup; there will be continuing operations work required. Handling upgrades, backups—this is all work that has to be done.

For this reason, I always try to use fully managed versions of the brokers where possible. Azure and AWS both provide a variety of broker options with various capabilities. Some of these broker services are unique to the platform (such as AWS Simple Queue Service) and have their own trade-offs. However, you can also make use of fully managed brokers like RabbitMQ or Kafka that you can otherwise run yourself.

In general, I like having other people do work for me, especially if they'll do it better than I could.[6] Even if I could become an expert in installing and running a message broker (and don't get me wrong—I'd give it a red hot go), to be honest, I have much better things to do with my time, and I suspect you do too.

Summary

When using a message broker, you get to offload some of the work around safely delivering messages to a separate system. The attraction is clear: smart people build and (one assumes) operate the broker, and the broker can handle things for you better than you can handle things yourself.

It's important, however, not to treat brokers as totally magical and infallible things. Hopefully, by now you understand just enough about how brokers work to recognize their limitations, and as importantly, you understand how to use them effectively.

In this chapter, we've looked a lot at what happens when a single machine fails. But what happens when an entire data center fails? We'll explore that in Chapter 10, where we look at disaster recovery.

---

1 Gregor Hohpe and Bobby Woolf, *Enterprise Integration Patterns* (Addison-Wesley, 2003).

2 Temporal coupling has a different meaning at a code level.

3 Historically, Kafka offloaded this responsibility to a separate application called ZooKeeper, but that requirement has been phased out.

4 Or more correctly, each consumer group, which could contain one or more consumers.

5 Although this may not be super precise—clock skew is a thing.

6 I'm pretty lazy; however, I drew the line at having AI write my book for me! Aside from the quotes, all the words in this book are my own, even the mistakes. Especially the mistakes.

Chapter 10. Disaster Recovery

We've already discussed how to handle a variety of failures that impact individual service instances. But what happens when a catastrophic failure causes widespread outage of large parts of your system? Fire, flood, electrical supply issues, and errors in the underlying cloud infrastructure are all outside of your control, and at one fell swoop, they can render your attempts to make your system more robust totally redundant.

Handling these situations falls under the auspices of *disaster recovery* (DR), a set of procedures that should kick in when something really bad happens. As with much of what we've looked at in this book, trade-offs abound. As you reduce your period of acceptable downtime or data loss, cost and complexity increase, so picking the right approach for your system can be a delicate balancing act. In this chapter, I'll give you some concrete targets to work with to help drive your own DR approach.

Cloud Region, Data Center, or Site?

The exact terminology you use to discuss your infrastructure will likely vary based on implementation details, and this in turn can make discussions around generally applicable patterns more difficult. When running on the public cloud, for example, you'll probably be familiar with the concept of a cloud region, which is where your cloud resources live. When running on a private cloud or in a more traditional on-premises setup, you'll instead think in terms of data centers.

As we discuss the various options around handling DR, the patterns that we'll see later in this chapter can be used for either public or private cloud scenarios. Therefore, to keep things simple, I'll use the term *site* as a general descriptor of where your computing infrastructure lives.

There are some specific issues to consider around mixing cloud vendors as part of a DR strategy, and we'll look more at those in Chapter 11.

ARE AVAILABILITY ZONES "SITES"?

For the sake of our discussion surrounding DR, we should consider a public cloud region as synonymous with a physical data center, and we could just call them both "sites." Those of you with knowledge of the public cloud may immediately stop and ask, "What about availability zones?" Each public cloud region (at least in the case of the main public cloud vendors such as AWS, Azure, and Google Cloud) is actually broken into two or more availability zones (AZs). When provisioning cloud resources, you ensure that they are spread across AZs to hedge against the risk of any single AZ becoming unavailable. In practice, each AZ is deployed as its own data center (and perhaps as more than one).

So shouldn't we treat AZs as "sites" too, and apply our DR patterns for dealing with outages of a specific AZ within a cloud region? The reality is that, because of how cloud regions are put together, the interconnections between the AZs are designed to deliver very low latency communication. This means running cross-AZ services is typically much less of an issue—having a single Kubernetes cluster or Kafka cluster running across multiple AZs is straightforward. Latency between cloud regions is much higher; the regions are much farther apart, and the quality of networking between regions is much more variable. Fundamentally, that difference in latency results in you having to look at different solutions to handle a full region outage.

So in a nutshell, from a DR perspective, as long as you make sure that within each cloud region you distribute your workloads (and data) across each AZ within that region, then you can consider one region to be a site for the purposes of our discussions here.

How Good Does Your DR Need to Be?

A spectrum of DR options are available to you, and they can range greatly in terms of cost, complexity, and recovery. To help you decide which DR approach is most appropriate, we should first try to understand how to determine the appropriate requirements for your context.

A simple way to define technical requirements around DR is to make use of two concrete measurable targets, *recovery point objective* and *recovery time objective*. In Figure 10-1, you can see how these objectives relate to an incident.[1]

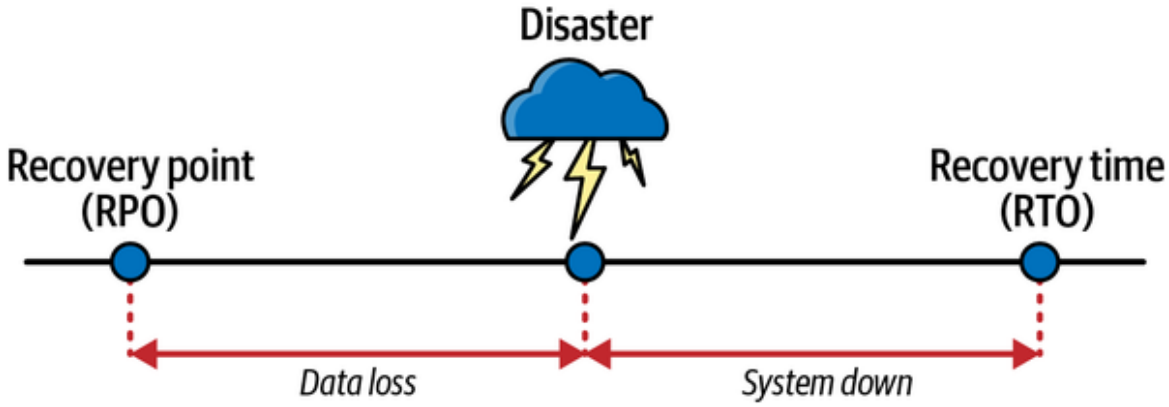


Figure 10-1. The key objectives to define as part of selecting an appropriate DR strategy

Recovery Point Objective

The recovery point objective (RPO) defines the maximum amount of time that can have passed since the data was last backed up or synchronized. Put another way, RPO defines the upper duration of potential data loss. If the data for your system backs up once an hour, then it's possible to lose up to an hour of data if your system has an incident. Therefore, your RPO would be an hour.

Recovery Time Objective

The recovery time objective (RTO) defines how long it's acceptable for your system to be unavailable in the wake of an incident. If all your computers became unavailable, perhaps because of a data center–wide

outage, how long would it take you to have your services running in an alternative data center? Your RTO includes all the time needed to recover data from the last recovery point, make computing resources available for your workloads, and redirect traffic to your new site—so there are more variables to consider in terms of bringing your system back online than just time.

You Can’t Have Everything

If asked what RPO and RTO you’d want in a vacuum, you’d probably give a very simple answer: you’d want an RPO of zero (i.e., a guarantee of no data loss), and an RTO of zero too. This is an understandable wish, but as you reduce your RPO and RTO, cost and complexity increase dramatically—and in technical terms, it’s not possible to guarantee a zero RPO or RTO in any case.

An additional complication is that different types of workloads might require different RPOs and RTOs. Consider a bank, for example. Having a low RPO to ensure that payment information isn’t lost would be key, and having a low RTO to ensure that people can still take money out of their account is vital. On the other hand, the part of the system that enables new customers to sign up for an account could have a higher RTO and RPO, as the impact to the business is less severe.

So when deciding on the appropriate RPO and RTO, there needs to be a discussion with stakeholders regarding end-user needs and regulatory and legislative requirements, which are balanced against the cost and complexity of implementing the required solution. For example, your product owner might initially say they need an RPO of 10 minutes, but if you could tell them it would take months of work to implement (during which other work has to be deprioritized), they might reduce their target, perhaps accepting an RPO that is easier for you to deliver. Your target RPO and RTO should also exceed anything specified in any service level agreements you have.

DR Objectives and SLOs

At first glance, RPO and RTO would appear to make for good service level objectives (SLOs). After all, they are both important measures that directly impact user experience. However, there are some challenges around managing RPO and RTO as SLOs.

When determining if we are meeting an SLO, we need to collect lots of service level indicators (SLI) to confirm whether or not we are on target. So if we had an SLO that defined 99th percentile latency over a one-month period of less than 500 milliseconds, we would analyze the latency-related SLIs for that monthlong period to make sure we were meeting the target. With RPO and RTO, though, what would our SLI be?

The reality is that we don't experience disasters regularly enough to gather useful SLIs. We can of course define related SLOs around things related to our DR approach. For example, it would be possible to define an SLO around speed of data replication, as this is something we are doing in production and can programmatically measure. But at best SLOs are only tangentially measuring RPO and RTO.

So RPOs and RTOs don't really fit neatly into the SLO concept. Of course, that doesn't mean that you can't still test your system to make sure it is delivering on your DR objectives—something we'll discuss more later in the chapter.

Disaster Recovery Patterns

With the discussion of RPO and RTO in mind, let's take a look at some common patterns used to implement DR in the wake of a major incident. In Figure 10-2, we can see an overview of each of the four patterns we will explore, arranged (broadly speaking) in order of increasing complexity and improving RTO and RPO.

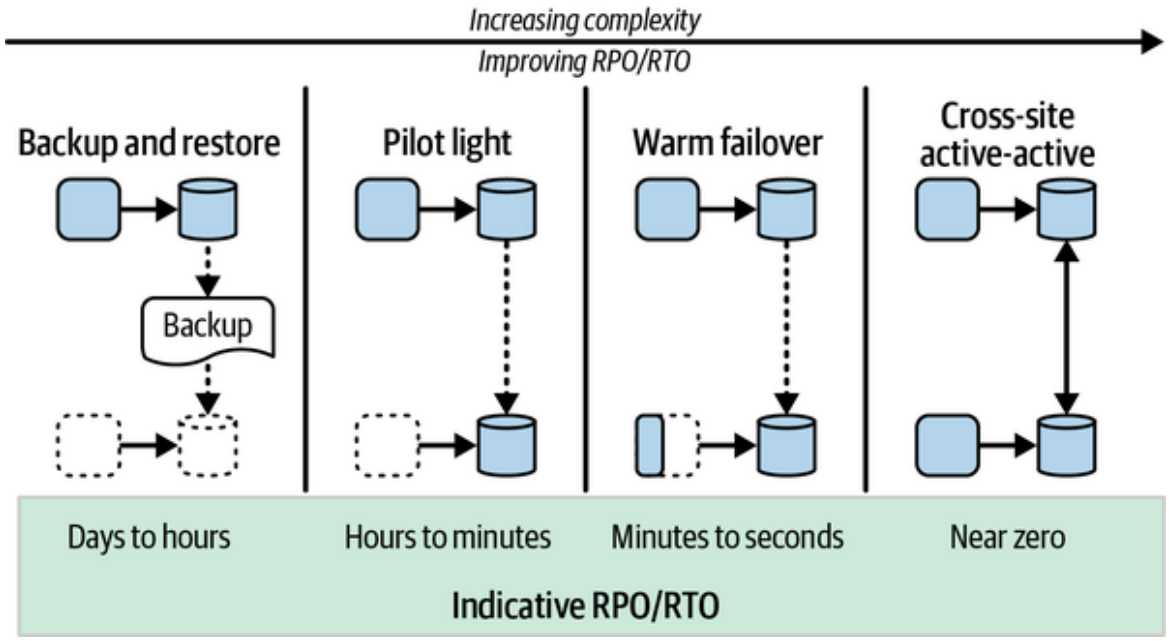


Figure 10-2. The key objectives to define as part of selecting an appropriate DR strategy

If starting from scratch, many of these patterns can build on one another, allowing you to incrementally increase the complexity (and cost) of your DR solution over time. For example, a backup and restore approach requires rerouting, a pilot light setup will add the need for cross-site data replication and infrastructure automation, and a warm failover adds on the cost of having some infrastructure ready to go. So if you aren't sure where to start, working your way through the patterns from left to right in Figure 10-2 makes a lot of sense.

If you have different RPO and RTOs for different parts of your system, you may also end up using a mix of these patterns. Note that they all require some form of rerouting of traffic, and most require some form of cross-site data replication. Those options are discussed later in this chapter.

Pattern: Backup and Restore

Recovery by restoring data from a backup and loading it into a replacement system.

As outlined in Figure 10-3, with this pattern, data is periodically backed up to a separate site. In the event that the primary site fails, infrastructure needs

to be provided at a new site, and the backed-up data restored.

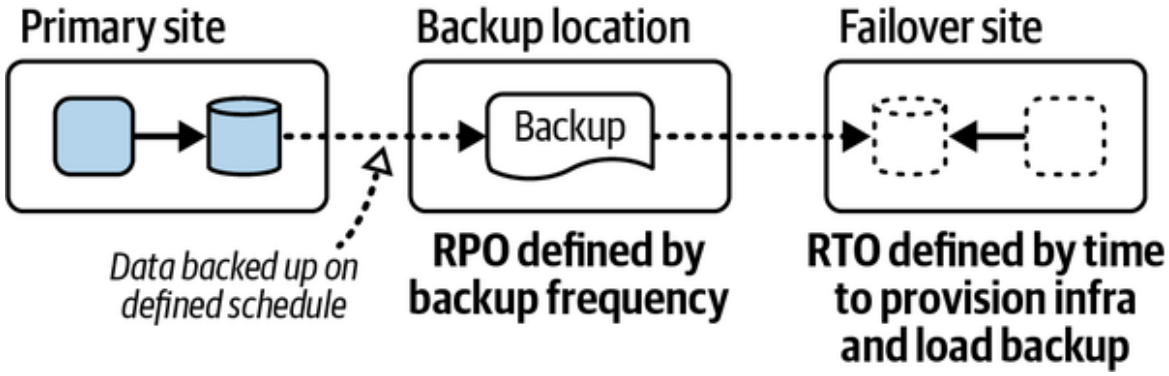


Figure 10-3. The backup and restore DR pattern

How it works

This pattern is very straightforward to implement, consisting of a few steps:

1. Back up data regularly.

All data that you want to make available to the failover site needs to be regularly backed up, with the backup data stored somewhere other than the primary site.

2. Provision infrastructure on the failover site.

This would ideally be automated, as manual infrastructure provisioning would likely lead to longer RTO and an increased chance of errors.

3. Restore data from backup to the failover site.

Data is loaded from the backup location into the newly provisioned infrastructure.

4. Switch traffic from the primary to the failover site.

Once infrastructure is available and the data is restored, traffic can be switched to the secondary.

Ideally, your failover site runs the same core technology stack as the primary site, meaning that any investment in infrastructure automation on

your primary can also be used on the failover (for example, being able to reuse Terraform scripts). Keeping the code to automate setup of the primary and the failover the same is very useful, as it means that you'll have a higher degree of confidence that spinning up the infrastructure on the failover site will work. If your failover site uses a different technology stack—meaning it's not possible to share automated infrastructure provisioning code between the primary and secondary sites—then it becomes even more vital that your DR failover process is regularly tested. A caveat to spinning up infrastructure on demand here comes in terms of guaranteeing that infrastructure is actually available when you need it, something we'll explore later in this chapter.

When to use it

This pattern is generally applicable in situations where your primary site is unavailable and you can tolerate a long RTO, and especially in those situations where you want to keep costs low. It's important to note that regular tests of a backup and restore process are necessary to ensure that it actually works. One approach I've found useful is to perform a backup and restore to a secondary site as part of other regular activity (for example, a regular load test exercise).

The RPO of this pattern will depend on the frequency of backup. The RTO will be the longest of the patterns we'll look at, and its time will depend primarily on how long it takes to restore data from the backup and provision replacement resources. With a fully automated process and smaller data volumes, this could be in the minutes to hours range. Manual recovery processes will likely extend these timelines greatly, as will the size of backups involved—terabytes of data take longer to send than gigabytes! Another major factor will be availability of the required computing resources. If you end up having to purchase replacement equipment, this pattern could easily result in a timeline of weeks or longer.

THE IMPORTANCE OF BACKUPS

It's vital to note that even if you decide not to adopt the backup and restore pattern and instead go for one of the more sophisticated patterns outlined elsewhere in this chapter, you should still maintain off-site backups as a fallback. These backups should be routinely tested. If using a public cloud vendor for your primary system, I'd strongly suggest storing backups either with a separate vendor or else in your own private infrastructure as a hedge against situations such as root cloud accounts being compromised.

As an example of what can go wrong, back in 2014 the root Amazon account of Code Spaces, a secure source-code-hosting provider, was compromised. The attacker who gained access to the account deleted all cloud resources, including all customer data, the source code for Code Spaces itself, and even the backups. If the backups for the system had been stored with a separate vendor, the company would have had a chance to recover. Instead, the company went out of business.

Pattern: Pilot Light Failover

Data is constantly synced to a failover site, with infrastructure provisioned after failover.

With a pilot light setup, data is continually replicated from the primary to the failover site, as shown in Figure 10-4. Infrastructure is not provisioned until a failover is required, ensuring that costs can be kept down when working with public cloud infrastructure.

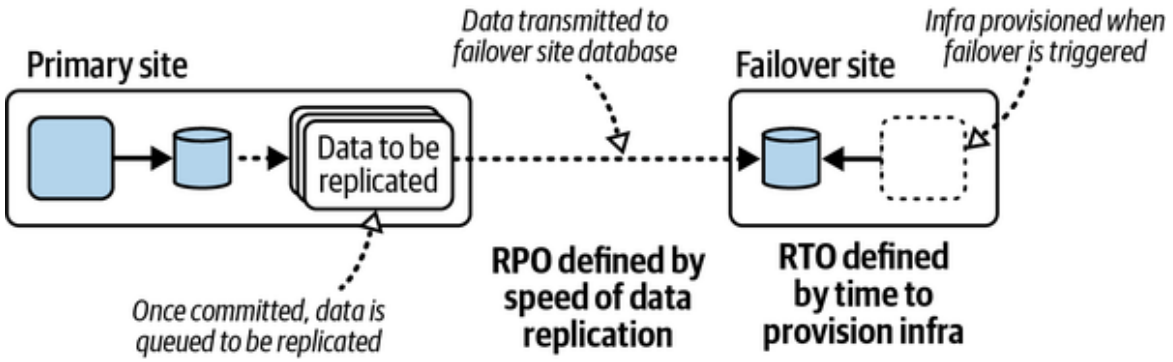


Figure 10-4. The pilot light failover DR pattern

How it works

This pattern builds on a lot of the work done with a backup and restore, the main change being that the data is continually synchronized to the failover site. As such, the steps involved look fairly similar to the backup and restore pattern:

1. Replicate data continually.

As data is committed to the primary, it is continually replicated to the failover site. This can be done in a variety of ways, such as by making use of native database technology or application-level replication.

2. Provision infrastructure on the failover site.

The assumption is that databases and datastores are already present on the failover site, so only remaining resources like compute would need to be provisioned.

3. Switch traffic from the primary to the failover site.

Once infrastructure is available and the data is restored, traffic can be switched to the failover.

From an implementation point of view, the new source of complexity comes from the replication of data from the primary to the failover. For systems designed with this process in mind, it might be somewhat trivial to implement. However, things become more complex when adapting existing systems if you're making use of a data product that doesn't support this

replication strategy natively. This is worthy of a more in-depth discussion later in this chapter, especially as the next two patterns also rely on this capability.

When to use it

When compared to a backup and restore, the pilot light failover allows for a much shorter RPO because of more frequent data replication, while also reducing RTO by eliminating the time taken for data restoration.

The cost-saving of a pilot light failover really makes sense only in the context of the public cloud. Costs are reduced primarily in terms of not paying for infrastructure unless it's used. But if your failover site is your own private infrastructure that you have to pay for regardless of whether it is used, then some form of a warm failover makes a lot more sense. At that point you're already paying for the hardware, so you might as well use it.

Pattern: Warm Failover

Data is synced to a failover site, with at least some infrastructure already provisioned to handle traffic.

With a warm failover, as shown in Figure 10-5, data is continually replicated from the primary to the failover site, with at least some infrastructure already provisioned in advance of the failover itself.

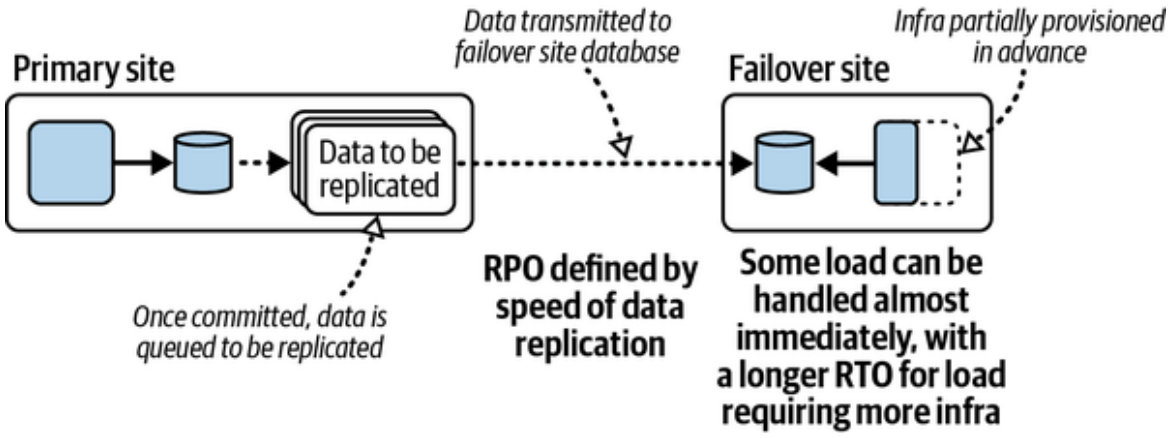


Figure 10-5. The warm failover DR pattern

Having some infrastructure provisioned even before the failover occurs results in a reduction in RTO for some functionality. How much infrastructure is provisioned on the failover can vary, but the idea is that some capacity already exists to allow for a much faster failover.

A common approach would be to prioritize the subset infrastructure that is provided on the secondary site for the more critical capabilities of the system, allowing for a shorter RTO for the most important functionality.

COLD, WARM, AND HOT FAILOVERS

A brief note is warranted here regarding the term *warm*. You'll find various solutions and vendor products promising *cold*, *warm*, or *hot* failover, with the only real distinguishing factor between these terms often being that a hot failover would give you a lower RTO in comparison to a warm failover, and a warm failover would give you a lower RTO than a cold one. Given that what constitutes hot or warm from vendor to vendor and from tech stack to tech stack varies so greatly, I just decided to pick *warm* and stick with it.

Naming things is hard.

How it works

From an implementation point of view, the warm failover is just a logical extension of the pilot light pattern, with two key changes regarding infrastructure provisioning on the secondary site and the need for some more sophisticated failover routing:

1. Provision infrastructure for critical functionality on the secondary site in advance.

> This needs to include enough infrastructure to run critical functionality, in addition to supporting any data replication from noncritical sources.

2. Replicate data continually.

As data is committed to the primary, it is continually replicated to the failover site.

3. *Divert failover critical traffic to the secondary.*

   Once a failure of the primary is identified, the traffic for critical functionality can be diverted, as the infrastructure should already be available.

4. *Provision remaining infrastructure on the failover site.*

   Spin up additional infrastructure to serve the remaining noncritical functionality.

5. *Switch traffic from the primary to the failover site.*

   Traffic for the remaining functionality can now be diverted over to the secondary.

It's important to note that you might decide *not* to bother failing over noncritical functionality to the secondary site. In a situation where the cost of providing DR for the noncritical functionality is too great, or where you think the primary might be offline for only a short period of time, you might want to wait before switching over the remaining workloads. This implies your system has been built in such a way as to support graceful degradation of functionality, a pattern we'll explore more in Chapter 11.

It's possible to be even more nuanced about what traffic you allow to immediately fail over. For example, you could prioritize some user workloads over others (likely by making use of a tenant-aware routing mechanism).

When to use it

The fact that infrastructure is available in advance of the primary site being unavailable means a reduction in RTO compared to a pilot light setup. This is balanced against the increased cost of having redundant infrastructure running in the secondary site.

In the current AI boom, infrastructure availability cannot be guaranteed. So another potential benefit of the warm failover pattern is that having a subset of your infrastructure being provisioned already in the failover site means you know you can use it when needed. If you can make opportunistic use of this redundant hardware during normal (nonfailover) operations, some of the cost downsides can be offset.

Given the challenges around infrastructure availability, we'll look at it in greater depth shortly, as well as looking at examples of how "redundant" hardware could be put to good use.

What this pattern does clearly opens up the ability to offer a much improved RTO for specific parts of your system and/or for important tenants. If you are building a SaaS-based platform, for example, this gives you the option to offer some customers an improved RTO as a premium option, or, more generally, to have a shorter RTO for critical functionality. If you want to push your RTO to zero, though, the next pattern is probably for you.

Pattern: Multisite Active-Active

The system is distributed across two or more sites, with writes being able to be sent to any site.

With a multisite active-active setup, data is replicated constantly between two or more sites, as you can see in Figure 10-6. Traffic for any customer can be served from any site at any time.

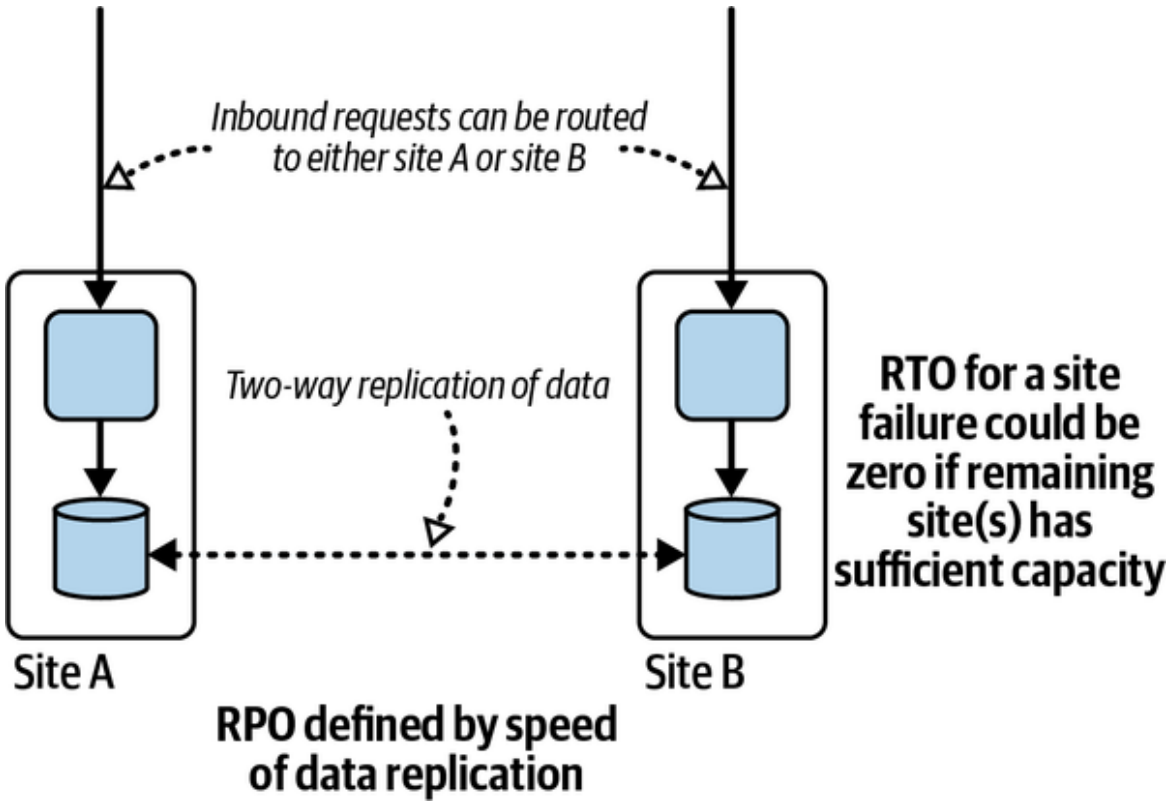


Figure 10-6. The multisite active-active DR pattern

This pattern sidesteps the need for a failover, as it is assumed that traffic for any user can switch between sites at any time. As such, it delivers the lowest RTO and RPO of any pattern we've looked at so far, though at much higher cost and complexity.

How it works

At a high level, this pattern seems like a simple approach to handling DR. There is no need for any operations to kick in when a site fails—everything should just keep on running, right? Well, sort of.

Firstly, our need for data replication immediately gets more complex (and more limited). Lots of data replication technologies work in an active-passive setup; it's something databases have done for decades. But far fewer technical solutions support two-way replication of data *and* allow for data to be written to either node at the same time.

Secondly, for this pattern to work as a viable DR approach, it has to ensure that a single site is capable of running all your traffic. This means either that

each of your active sites is constantly running excess capacity (which has a cost), or that you can dynamically provision new capacity when the failover occurs, which implies there would have to be some ramp-up time and perhaps a period of reduced capacity for the overall system as new infrastructure is provided.

In practice, traffic for a given user session will likely be pinned to a specific site. Data takes time to be replicated between the sites. If calls within a single user session could switch between the sites, then you would have to accept that the user may see inconsistent data, or else you would have to delay user activity as it switched between sites until you were sure that any relevant data had been copied across. If you keep all activity for a single session within a single site, you sidestep this particular challenge. If one of the active sites becomes unavailable, then a new session for the user would have to be established on the other active site—this may or may not be visible to the end user, depending on how it is implemented.

When to use it

Fundamentally, the cost and complexity of a multisite active-active setup for an entire system ends up being prohibitive for many organizations. It's something I've rarely encountered in the wild, and when I have, it's been considered a core driver for the system architecture. Notable examples of large-scale companies practicing multisite active-active include Netflix, which has talked openly about the lengths it goes to achieve this. It's worth noting that, purely from a traffic point of view, the vast amount of Netflix's use case is for reads—specifically, people streaming video media. The "writes" from a user point of view will be in terms of tracking user behavior to drive Netflix's recommendation algorithm (and also to remember where you were up to on your latest show!). As a result, Netflix can likely tolerate quite large RPOs for a lot of its data, whereas a low RTO would be vital.

It's possible, of course, that you adopt this pattern for specific critical situations in which the cost (and implementation challenges) of multisite active-active could be justified.

You can get more of a sense of how rare multisite active-active is when we look at the impact of the AWS US-EAST-1 region outage of 2025 in Chapter 11. Multiple big-name companies were taken offline as a result of that outage, and they had all decided that a cross-site active-active approach wasn't for them.

Infrastructure Availability

At the time of writing, GenAI is being used by more and more industries. Whatever your views on the usefulness of this technology (or its broader societal impact), the nature of GenAI workloads for both training and inference purposes has driven ever-increasing demand for infrastructure, in turn leading to reduced capacity for many users. Even if you aren't running what you consider to be specialized GenAI workloads (for example, those requiring specific GPUs), you may well see the impact of these shortages more broadly.

Aside from the ever-growing cost of GPUs, memory costs have spiked significantly in 2025 and 2026, impacting general server availability. I've already spoken to clients who are struggling to run "normal" (non-AI) workloads on some of the large public cloud providers, as they can't secure enough capacity. Being able to shift regions in some cases is helping, but guarantees of future availability seem fairly weak. Even buying your own hardware doesn't seem to be a quick fix, with lead times for servers seemingly extending further and further with each passing week.

This has to be taken into account when considering the patterns detailed previously. If your chosen approach assumes you can dynamically provision infrastructure when a failure occurs, you will have to look at what guarantees you have around the capacity you want. In a scenario where a major cloud vendor's region has failed, you may find yourself fighting over scraps with everyone else.

There are three ways to handle this scenario: reserve capacity; prioritize critical services; or lastly, consider running a fully redundant set of infrastructure at all times.

Not every vendor offers the ability to reserve capacity, and even then it is limited to certain cloud services. On AWS, for example, you can pay a fee to guarantee the availability of EC2 instances (managed virtual machines) when you need them. This has a cost impact, as you pay a fee for the reservation whether or not you use the instances (the silver lining being that if you did end up using them, the cost would be cheaper than normal in-demand pricing). This doesn't help you, though, when using cloud resources for which reservations aren't possible—you can't purchase guaranteed availability of Lambda invocations, for example.

Next, consider prioritizing critical services. If there is a cost to guaranteeing availability (or if only a limited amount of infrastructure is available), then it's vital that this goes to workloads that have a higher criticality. This is where having separate RTOs and RPOs on a service-by-service basis comes into play.

Finally, if you could justify the complexity and expense of having a fully redundant set of infrastructure ready to run your system, then you'd have the confidence that your computing resources will be available, with the caveat that this option can be extremely costly. But that cost can be mitigated if you can make alternative use of the redundant resources.

Back in Chapter 4, I mentioned BankCo, where we ran calculations across a large pool of machines. A large portion of these computers were desktop PCs sitting idle in an office situated outside London. These PCs formed part of the bank's DR strategy: in the event that traders had to evacuate the bank's main location, they could relocate to the secondary site and continue working. It was a requirement that these standby PCs were constantly running to ensure that we knew that they worked, and so that updates could be pushed out to them. This was obviously back in the day when most of the critical work relied on thick-client applications, so they couldn't be accessed via the web.

We were able to make use of these DR workstations to grow our pool of available workers. This meant that when the DR site wasn't in operation (which was most of the time), we were able to carry out our calculations

much faster, as we could parallelize the work over more machines. If the DR site's machines were unavailable to us, we still had a set of our own servers on which we could run the calculations to suit our normal targets.

So if you are thinking about securing ring-fenced infrastructure as part of a DR approach, see if you can think of useful ways to put it to good use when it's not needed for production workloads. You could use the spare infrastructure for a load test environment, for example; the load test could easily be aborted when DR is needed.

Multisite Data Replication

All the DR patterns we have explored, aside from backup and restore, require some form of continual data replication between sites. A host of implementation options are available here, each with its own trade-offs. Before we consider those, though, we should look at why one of the fundamental challenges of distributed systems can make replication challenging.

Impact of Latency

When it comes to replicating data across sites, the main technical challenge is caused, fundamentally, by latency. As you'll remember from the discussion of the three rules of distributed systems way back in Chapter 1, you cannot beam information instantaneously between two points. How long it takes to send data between those two points can vary due to a variety of factors, including the quality of the networking equipment used, the volume of data being sent, and the protocols used to transmit the data. Another massive factor is simply distance.

When replicating between sites, this distance becomes a massive factor—as does the networking options available to you. Many of the faster networking technologies don't work over longer distances, and those that do can end up being prohibitively expensive. Why does this cause an issue?

When replicating data across sites as part of a DR strategy, we most likely want to make sure that the data has been successfully written. If, when writing data, we wait until we get confirmation that the data has been written, our writes will take longer. If, on the other hand, we want to make our writes faster, we could return after a write and assume that the data will be replicated later on—but this runs the risk of us thinking our data is safe, only to find we lost data because it wasn't replicated before we had an outage.

Some of the technical solutions we'll look at shortly enable you to decide how this replication is done, so you can choose on a case-by-case basis which approach best fits your scenario. However, some of these solutions don't give you that ability, meaning the decision is made for you. This needs to be taken into account when considering the right solution.

Active-Passive

For both the pilot light and warm failover patterns, we need a data replication solution that supports an active-passive setup. With active-passive, data is written to an active node and replicated to a passive node. The passive node keeps our data safe in case of the active node failing. The passive node would then need to become an active node to allow for writes to continue.

Once the primary site becomes available again, the data needs to be replicated back to the previously active node in the primary site before that node can be made active again.

Exactly how this is handled varies from technology to technology. Where appropriate, I'll call out implementation-specific details of interest as we look at some of the implementation options.

Active-Active

For the cross-site active-active pattern, we need a technical solution that allows data to be written to nodes in either site and have the written data replicated to the other site. There is no notion of one of the nodes being

more in charge than any other; however, node partitions still have to be taken into account. If site A goes down, and data is being written only to site B, then when site A is available again, data has to be replicated over. This also has to be handled even in a situation where both site A and site B are working fine but the connection between them fails—there has to be a "healing" process during which data that hasn't previously been replicated can be synced up.

A major challenge to be resolved with any active-active data replication is dealing with conflicts. As data can be written to either site, what happens if the same piece of data gets changed at two sites at around the same time? Ultimately, there has to be an approach for handling these conflicts; as an example, Cassandra adopts a simple model in which the last write wins.

Many of the active-active solutions we'll look at support multiple sites, although as you add more sites, latency can become more of a concern in situations where you are waiting for all sites to confirm replication of data after a write.

Fundamentally, as I've already touched on, you should expect setting up active-active data replication to be more complex and more costly. You should also expect there to be far fewer options to choose from than there are with active-passive.

Implementation Options

With our general discussion around data replication out of the way, let's look at some of the options for multisite data replication.

Relational databases

Standard relational databases have long supported active-passive replication of data. This is typically driven by replication occurring after transactions are committed, at which point the committed data is replicated from the *active* database to one or more *passive* replicas. This is a convenient way to scale reads—read traffic can be diverted to *read replicas* that are structurally the same as the primary node, without allowing any writes.

From a consistency point of view, you shouldn't expect to have much control over how the data is replicated, and all the solutions I have used in this space end up being eventually consistent.[2] That is, it's only after the successful completion of an ACID database transaction against the active node that data will then be replicated to the passive node(s). Eventually, all replicas will get the newly modified data. Your RTO will be driven by the time taken to perform this replication. It's worth noting that relational databases typically give you the ability to see what data has been sent to your replicas, allowing you to dynamically measure the time taken for your replicas to get up-to-date data. This in turn would help you measure your current RPO and make sure that it matches your target.

To use a relational database with active-passive replication as a DR approach, you'd need the ability to promote the passive replica to being a primary node. In Postgres, for example, this is as simple as running a `pg_promote()` command on the appropriate node. This could be done manually or automatically.

An interesting niche in the world of relational databases is so-called distributed relational databases, such as CockroachDB or Google's Spanner.[3] These databases aim to give you the ease of use (and familiarity) of a relational database, albeit with the added benefit that they can work in an active-active setup. Spanner is a managed service that is available only on Google Cloud itself, and its impressive claims regarding its scalability have to be considered not just in terms of Spanner's architecture but also in terms of the data centers that it runs on. Google ensures very fast interconnects across sites, has tight control over the availability of machines that Spanner runs on, and even makes use of atomic clocks running in satellites in geosynchronous orbit to help synchronize machines.[4] These things are difficult to do on your own infrastructure. This needs to be considered when deciding to make use of CockroachDB, which was inspired by Spanner's design. If you run CockroachDB on infrastructure you manage, you may struggle to deliver the same quality of service as Google does with Spanner.

While I can't speak to the quality of implementation of either CockroachDB or Spanner, anecdotally it's extremely rare that I meet people who are actively using either. I've certainly heard reports of people struggling to get CockroachDB to perform appropriately, which at the time I put down to problems with the underlying infrastructure being used. Another issue I've heard is that both databases suffer a little from an "uncanny valley"-type problem—they are familiar enough that at first glance they seem just like a relational database, but there are a few key differences that can catch people unawares.

Vector databases

Vector databases have become a vital component of AI-driven workloads. During the training process, "unstructured" data gets converted into vectors that allow you to later make inferences. An example inference could be comparing a picture of a plant you just took and trying to work out what it is. The vectors created during the training process can have hundreds or even thousands of entries, and large training sets can have billions of vectors. When performing inference, we want the process of scanning similar vectors to be fast.

In some cases, preexisting databases have added capabilities to allow them to support use as a vector database. Examples include Postgres (using the `pgvector` extension) and Neo4j, a graph database that added support for vectors in recent years. Both of these databases already have extensive support for various failover models. The dedicated vector database sector is quite new, although products such as Pinecone offer a few different high availability options.

It's worth noting that vector data has a very different lifecycle from that of normal transactional application data. Much of our data—for example, the details of an order in an ecommerce system—will have a mix of reads and writes over its lifetime. With vectors for inference, however, the vector gets created during the (often long) training process and is only read during inference. This means that for inference workloads, your DR approaches

become much simpler. Creating multiple redundant copies of read-only data is a simple affair.

Cassandra

Cassandra is a type of column-oriented database, meaning that data is stored in columns rather than rows. The upshot of that is that you normally store your data in binary or textual objects in one column, with a single key used to look up that data. This immediately makes it harder to query than a relational database. However, the flip side is that this type of database allows for scaling of writes and reads in ways not possible for a normal relational database.

Cassandra is itself a natively active-active database. A Cassandra database is run as a ring, with at least three nodes normally making up the ring. Data can be written to any node. For example, when writing data, you can decide that the write should return after it is stored in the local node (which will be fast but won't guarantee that the data has been safely replicated)—or perhaps you might want the write to block until the data has been replicated to a minimum number of replicas. The consistency controls for writes are mirrored for reads, allowing you to trade off fast response times against making sure that you have the latest version of the data.

The eagle-eyed among you might think this sounds a lot like the consistency controls for Kafka that we discussed in Chapter 9, and you'd be right—there is a lot of shared mindset here.

Cassandra supports running a ring within a single site (this could be within a single data center or AZ, or across an AZ but still within a region) but also supports extending a ring across sites. This gives you a broader range of consistency controls—you could now, for example, block a write until data is replicated within the local site but not yet the other sites, or until a quorum of other sites has the data, and so on.

I mentioned that Netflix is one of the few organizations out there that aims to build true multisite active-active systems, and its use of Cassandra is a major part of this. It's worth noting that Cassandra is essentially a clean-

room implementation of the ideas behind Google's Bigtable, a column-based database that is available on Google Cloud.

Public cloud global tables

Both Azure and AWS offer fully managed database services that can provide for multisite replication and active-active use cases. Azure Cosmos DB and Amazon DynamoDB are familiar-looking databases that provide nonrelational datastores, where some tables can be replicated across multiple cloud regions (sites). As with Cassandra, when reading or writing from these global tables, you have some choices around consistency controls, although to my mind they don't seem as clear-cut as Cassandra's well-documented options.

Message brokers

Some message brokers can be used in multisite setups. Both Kafka (via a tool like MirrorMaker 2) and RabbitMQ (via a plug-in like Shovel) can have their data replicated across sites. It's worth noting that this isn't like a single broker cluster running across multiple sites (as in the case of Cassandra); instead, each site would have its own local cluster, with the appropriate tool used to replicate data between clusters.

CONFLICT-FREE REPLICATING DATA TYPES

Worthy of mention, though not a data replication technology per se, is a specific type of data structure known as a *conflict-free replicating data type* (CRDT). A CRDT is a data structure that can be replicated and have the replicas updated independently from one another, and the replicas can still converge to a known state. As a super simple example, consider a one-way switch, which starts in the "off" state. You could create lots of copies and switch any one of them to the "on" state, and when merged back together they can converge back to a simple "on" state.

In a multisite setup, the implications here are interesting. A copy of the data structure could be changed by any node, and once replicated around could be converged to a known state. CRDTs don't require any client-server communication to work; as such, they work really well in highly decentralized environments. The burden of implementation here shifts from server software such as databases or brokers to libraries that implement the CRDT types themselves. That said, most of the work done in the area of CRDTs relates to collaborative applications, such as in the automerge project. So the use of CRDTs is unlikely to help fulfill your general data replication needs.

Further Reading

I have barely scratched the surface in terms of the different databases and general data replication options available to you. If you want to explore patterns around cross-site data replication in more detail, I recommend Bartosz Konieczny's *Data Engineering Design Patterns*[5] and the second edition of *Designing Data-Intensive Applications* by Martin Kleppmann and Chris Riccomini,[6] which has a chapter dedicated to replication.

Testing DR

Having well-defined RPO and RTO for your system is all well and good, but how do you know if you can actually deliver against these objectives? Well, you need some way to test your system.

When testing your DR approach, and before trying to simulate a full-blown site outage, it's worth breaking out the individual pieces that you want to validate and finding ways to test them in isolation. For example, with a pilot light setup, you can already validate that data is being replicated across sites by checking production. You could then do a standalone test of your routing switch over functionality, and even go as far as to spin up the extra computer resources you'd need in the failover site. Spinning up extra resources in the production failover site should have zero impact on production, but you'll get a lot of valuable information from the exercise.

Once you've done as much testing as possible of the individual pieces of your DR approach, the ultimate test would be to do a simulated production incident. Now, you could do a full simulation—stand up a fake primary and failover site, simulate traffic, and then shut down one site. The issue, though, is that setting this up takes a lot of work (and expense). Ideally you'd get to a point where you can trigger a fake site outage in the real production environment. This isn't for the faint of heart, of course, but I would argue that if you've never been brave enough to actually do this in production at some point, then you can't have much confidence in your DR process. To limit user impact, you could schedule these exercises to be done outside of major working hours.

> **WARNING**
>
> Please do all the easy and safe testing first, *before* you just go and turn off a production data center.

With a system where you have a multisite active-active setup, your DR approach is constantly being tested to an extent. You'll know whether the

data replication is working well before a disaster actually occurs. You still may want to do a failover test, however, if for no other reason than to ensure that once a single active site becomes unavailable, you have sufficient capacity in the remaining site to handle load, and to ensure that your underlying data replication strategy handles a site being unavailable.

Summary

When considering which DR approach is appropriate for you, you need to start with clearly defined targets. Define a recovery point objective (RPO) to specify how much data loss is acceptable, and a recovery time objective (RTO) to specify how much downtime is acceptable. In general, as you decrease RPO and RTO, you should expect the cost and complexity of your DR strategy to increase, potentially significantly.

Many of the DR patterns we looked at build on each other. So starting with a backup and restore approach is sensible if right now you have no DR in place. Over time you can look at moving to a pilot light or warm failover strategy once you can justify the cost and increased complexity. A multisite active-active approach is not always justifiable.

To implement many of the DR strategies in this chapter, you'll need a way to replicate data, reroute traffic, and provision infrastructure. And whatever approach you pick, you'll need good backups.

Finally, your DR approach doesn't have to be one-size-fits-all. For critical functionality, you might pick a DR approach with lower RPO and RTO, as it's easier to justify the cost and complexity. For less critical functionality, you can pick a simpler approach or even decide to skip DR altogether.

One of the causes of site outages can be nasty cascading failures—small problems that suddenly snowball, taking out entire data centers. We'll look at this phenomenon in more detail in Chapter 11, when we explore the concept of progressive collapse.

[1] This diagram draws heavy inspiration from the very good overview of DR objectives in the AWS Well-Architected Framework.

[2] I've come to dislike the term *eventual consistency* and now prefer *eventual convergence* after reading Pat Helland's blog post. I've stuck with *eventual consistency* here, though, as that is still the most widely used name for this concept.

[3] I say "so-called" here because lots of "normal" relational databases are themselves distributed systems, once you consider read replicas or cross-schema views.

[4] I still find that pretty cool.

[5] Bartosz Konieczny, *Data Engineering Design Patterns* (O'Reilly, 2025).

[6] Martin Kleppmann and Chris Riccomini, *Designing Data-Intensive Applications*, 2nd edition (O'Reilly, 2026).

Chapter 11. Progressive Collapse

As I've touched on a few times in this book, as you increase the complexity of a system to improve one aspect of its resiliency, that new complexity can introduce new sources of failure all on its own. This is partly because as you introduce new components into a system, they bring their own challenges. But also, as a system becomes more complex and interconnected, it can become difficult to assess the impact of a seemingly small thing going wrong.

How can an overly tightened nut cause a tower block to collapse? How could a simple race condition cause thousands of systems to fail? In this chapter, we'll look at the phenomenon of progressive collapse and examine what you can do to protect yourself from it.

What Is Progressive Collapse?

When the failure of a single component results in the failure of other connected elements, this is known as a *progressive collapse*. In the words of an NIST report on reducing the potential for progressive collapse in buildings:

> *The underlying characteristic of progressive collapse is that the final state of failure is disproportionately greater than the failure that initiated the collapse.*[1]

In other words, the end result of the progressive collapse is much worse than would have been expected based on what initially went wrong.

In this chapter, we will mostly look at this concept in the context of our distributed systems. Before we do that, however, it's useful to look at how this plays out in the context of physical structures. For example, when a

building collapses or a bridge falls down, a lot of analysis is done in the wake of the disaster. Examples like these give us the chance to learn many lessons that can then be applied to our own distributed systems.

Real-World Examples

On March 26, 2024, the Francis Scott Key Bridge in Baltimore collapsed when a container ship, the *Dali*, crashed into one of the piers that supported the bridge. The pier failed, drastically increasing the stresses on the other supports, which in turn failed. Six people died. The container ship in question had lost power because of an electrical fault, causing it to drift downstream.

Another example of a progressive collapse occurred on May 16, 1968, when a gas explosion caused the partial collapse of the Ronan Point residential tower block in Canning Town, London. The explosion on one of the tower's upper floors caused the outer load-bearing wall to be blown out. The floors above, now unsupported, crashed down into the floors below, causing most of the southeast corner of the 22-story tower to collapse. Four people died as a result. The saving grace was that the explosion happened in the early hours of the morning, and it was mostly living rooms that were impacted.

The initial triggers for these incidents are quite different from each other. In the case of the Baltimore bridge, a loose wire caused the container ship to lose power, resulting in it crashing into the supporting pier.[2] In the case of the Ronan Point tower, it was determined that the cause of the initial gas leak was a cracked nut, which was likely damaged by being overtightened during installation. In both cases, however, a single component failing—a supporting pier, a cracked nut—caused connected elements to also fail, resulting in a catastrophic event.

To be clear, catastrophic failures are never caused by just one thing. There is no such thing as a single point of failure: many factors end up contributing to these adverse outcomes. You could argue that if the nut

wasn't cracked during the installation of the gas system in the Ronan Point tower, the explosion wouldn't have happened, and the building wouldn't have collapsed. But if the building could have withstood a single load-bearing wall failing, the explosion would have caused much more limited damage. The reality is that things will always go wrong—it's our job to make sure that when they do, as many things go right as possible.

Remember our discussion in Chapter 1 of how resilience engineering emerged from the previous, more limited world of safety management? In safety management, the focus would have been more narrowly on making sure the nut didn't get cracked. In resilience engineering, we do our best to make sure the nut doesn't get cracked, but we also consider how to build a system that ensures that a single cracked nut doesn't result in a much larger failure.

In Distributed Systems

Let's bring the concept of progressive collapse back to our distributed systems. Consider the example of the load balancer failure in Chapter 5, reproduced here in Figure 11-1. When one of the `Pet` service instances fail, the resulting load is spread across the remaining instances. The remaining instances cannot handle the load, so they too fail, resulting in a domino effect.

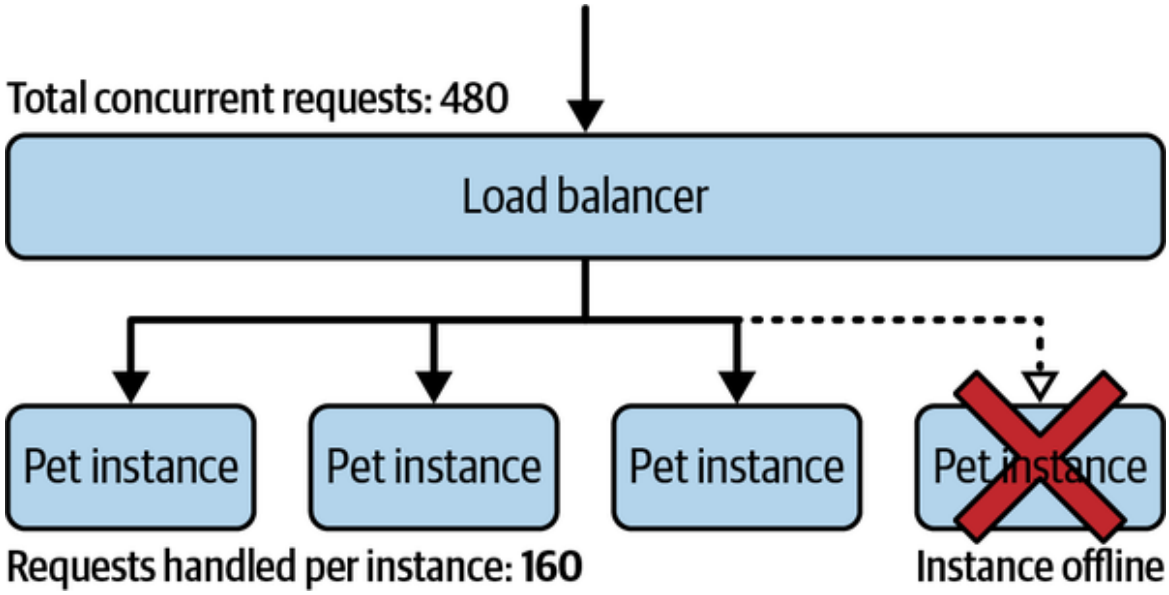


Figure 11-1. An instance failure means incoming work is distributed across the remaining three instances

However, we should take a wider view and consider the impact of the `Pet` service failing on the wider system. Imagine if the `Pet Sitter` and `Trainer` services are both consumers of the `Pet` service. What happens to them if the `Pet` service has an issue?

This example is obviously somewhat contrived, but it also has the benefit of being fairly easy to understand. Let's look now at a more complex example of a progressive collapse in a digital system.

Case Study: AWS US-EAST-1 Outage, October 2025

On the evening of October 19, 2025, the US-EAST-1 region of AWS suffered a major disruption that eventually led to a large-scale outage of AWS services and customer systems running in that region. This had a knock-on effect, impacting systems and devices globally.

Context

AWS's cloud services are broken down into a series of regions. Each region is geographically separated from other regions and is further broken down into two or more availability zones (AZ), which are to some degree isolated from each other (although much less so than the degree to which regions are isolated from each other). The idea is that if one AZ suffers a failure, you should still be able to run cloud resources out of the remaining zones in that region.

AZ failures have historically been uncommon. Less common yet has been the failure of an entire region.

The AWS region that suffered the 2025 disruption was US-EAST-1, which is situated in Northern Virginia. This was the first cloud region that AWS launched, and many companies that were early to the cloud started in this region (and they remained there). This region often saw new cloud services and features rolled out earlier than in other regions, which meant that many people picked this region to run their own cloud resources to ensure that they could take advantage of the latest and greatest features more quickly.

What Happened

DynamoDB, one of AWS's products, is responsible for maintaining DNS entries used by load balancers and other services inside its AWS region. As capacity changes, resources become available (or not), so keeping these DNS entries up-to-date is important to making sure that people get the resources they need. Unfortunately, the dynamic updating of DNS entries, which was at least partially intended to improve the fault tolerance of AWS, itself ended up being the source of this particular outage.

The DNS routes were dynamically maintained by two components. The first, the DNS Planner, determined what new DNS routes were required and then created plans for the changes that would be executed by the second component, the DNS Enactor. Multiple instances of the DNS Enactor could run in different availability zones: even if one Enactor was offline, one of the other copies could pick up the plans.

On October 19, 2025, one of the DNS Enactors applied a plan, but for a multitude of reasons, the plan took much longer to apply than normal. Because of the abnormal delay in applying the initial plan, enough time had passed that the DNS Planner was producing newer plans that also needed to be applied.

With the first DNS Enactor still working, a second DNS Enactor picked up the latest plan and quickly completed its work. Subsequently, the first DNS Enactor finally completed its work, thus overwriting the newer plan.[3]

The net result is that DNS entries ended up getting removed, with the system left in an inconsistent state that required manual intervention. These DNS records were vital to internal services in AWS, and without them those services started to fail.

Impact

The knock-on effect of this region outage caused significant impact to companies like Zoom, OpenAI, Snapchat, and even Amazon itself (both its Alexa and Ring devices were affected). Many people who relied on services delivered from this region were also impacted: smart door locks failed to lock (or unlock), wake-up alarms didn't go off, expensive smart beds stopped responding, and even smart cat litter boxes stopped working.[4]

The fact that it was the US-EAST-1 region that failed definitely compounded the scale of the failure. It's reasonable to assume that a failure of one of the other regions would not have had such a great impact.

In the following sections, we'll explore how progressive collapses are mitigated in the physical world. We'll also see what lessons we can learn for similar failures in our digital systems, including in the context of the AWS outage.

Mitigating Progressive Collapse

A lot of work has been done to eliminate progressive collapse in the context of physical structures, and this can be useful framing for examining this phenomenon for our distributed systems.

Borrowing and somewhat adopting the suggestions for managing progressive collapse in physical structures, we have three broad forms of mitigation we can consider:

Reduce hazards

> Reduce the causes of component failure.

Strengthen individual components

> Reduce the chance of a single component failing, perhaps by increasing redundancy.

Reduce interconnection

> Ensure that the failure cannot cascade across to other components, either by ensuring that load is able to be transferred elsewhere or by adopting bulkheads to seal off a failed component.

Let's look at each form of mitigation in turn, and where appropriate I'll call out specific useful patterns that we can explore in more detail later.

Reduce Hazards

Progressive collapse starts with some inciting incident. In the case of the Ronan Point tower collapse, that incident was a problematic installation of the gas supply. In the case of the Baltimore bridge collapse, it was a ship having a power failure and as a result traveling out of control and colliding with the bridge.

In the wake of the tower collapse, it was suggested to stop the use of gas in high-rise buildings. In the case of the bridge collapse, recommendations included making specific improvements in aspects of the electrical wiring components to reduce the recurrence of an electrical failure in the future,

and better assessing bridges to determine their risk of collapse and come up with appropriate remediation plans.[5]

With the AWS outage, how could the chance of a similar inciting incident be reduced? In the short term, AWS decided to disable the dynamic updating of DNS entries—not just in US-EAST-1, but globally. This gave it time to fix the underlying issues without running the risk of a repeat incident in the meantime. The incident report also detailed that, once the race condition was fixed and the DNS Enactors put back into service, improved testing would be put in place to catch issues like this one in the future.

Other examples of reducing hazards in a distributed system can include rate limiting to ensure that individual components aren't overwhelmed (which could protect our `Pet` service), or perhaps the use of automation rather than manual operational tasks to reduce the impact of mistakes—as we saw with the desired state management pattern in Chapter 8. Many security controls are, in effect, attempts to reduce hazards by stopping malicious parties from taking control of a system or stealing data.

Strengthen Individual Components

If the inciting incident cannot be stopped, what about ensuring that the component is strengthened to resist failure in the first place? With the Baltimore bridge, the supporting piers were intended to survive collision with ships of a certain size. Since construction of the bridge in the 1970s, ships have gotten bigger. In the wake of the incident, it has been posited that a future bridge design should incorporate piers that are able to survive impacts from larger ships.

With the Ronan Point tower, the structural integrity of the building was based on large concrete panels that made up the building's outer skin. After the incident, the connections between many of these panels were found to be missing or badly installed. Properly installed connections may have helped maintain the structure of the building and stopped the subsequent collapse.

With the `Pet` service, if the instance went offline because of a power supply failure, then redundant power supplies would help greatly. This measure would be less effective, though, if the instance failed because of a water leak, fire, or configuration error.

Adding more components to a system can help avoid some forms of progressive collapse. If the Baltimore bridge had had more piers supporting the span, it's possible that even when one of the piers failed, the remaining piers could have supported the weight of the span.

With our `Pet` service, our progressive collapse is due to the number of instances remaining in the service being unable to handle the load being sent. Having more instances could have enabled the `Pet` service to withstand the initial loss.

Increasing the redundancy in a system has a cost. Unless a failure occurs, the extra instances of the `Pet` service aren't needed. Consider the huge cost of building extra support piers for the Baltimore bridge. Aside from the work to build the piers, their addition would result in a more congested riverway under the bridge, which would in turn increase the chance of collision.

We can look at the AWS outage from a couple of angles. Firstly, from the point of view of AWS, how could it make the DNS update process "stronger" and less likely to suffer a failure? Ironically, having multiple DNS Enactors was an attempt to make the process more fault tolerant, but it introduced the face condition that ultimately brought the whole thing down. This is not the first time in this book that we've seen an attempt to make a system more robust end up introducing new sources of failure.

From the point of view of AWS customers, what could they do to "strengthen" their use of AWS? One of the major talking points in the aftermath of the incident was moving to a multiregion setup. At a high level, you could see this through the lens of scaling up a service to add redundancy: I have multiple instances of the `Pet` service, so if one fails, I can still provide the required functionality. So, the theory goes, why not distribute your services across multiple cloud regions, mitigating a single

region failure? This is a tricky topic, and one we'll explore in more detail later in this chapter when we look at distributing work across failure planes more generally.

Reduce Interconnection

The final approach to mitigating a progressive collapse that we'll look at is reducing interconnection between components in the system. If two components are isolated from each other, a failure in one component shouldn't impact the other.

In the case of the Baltimore bridge, it was designed as a single continual span, as outlined in Figure 11-2, rather than as separate isolated spans. This means that the bridge roadway was one continuous piece. When a pier failed, the whole weight of the span was transferred to other piers, which couldn't handle the load. However, if the roadway had been split into separate spans, losing a pier might have caused part of the roadway to fail, but other spans would hopefully have survived. The bridge might have no longer functioned as a bridge as a result, but the risk to life and the time taken to rebuild the bridge would have been reduced.

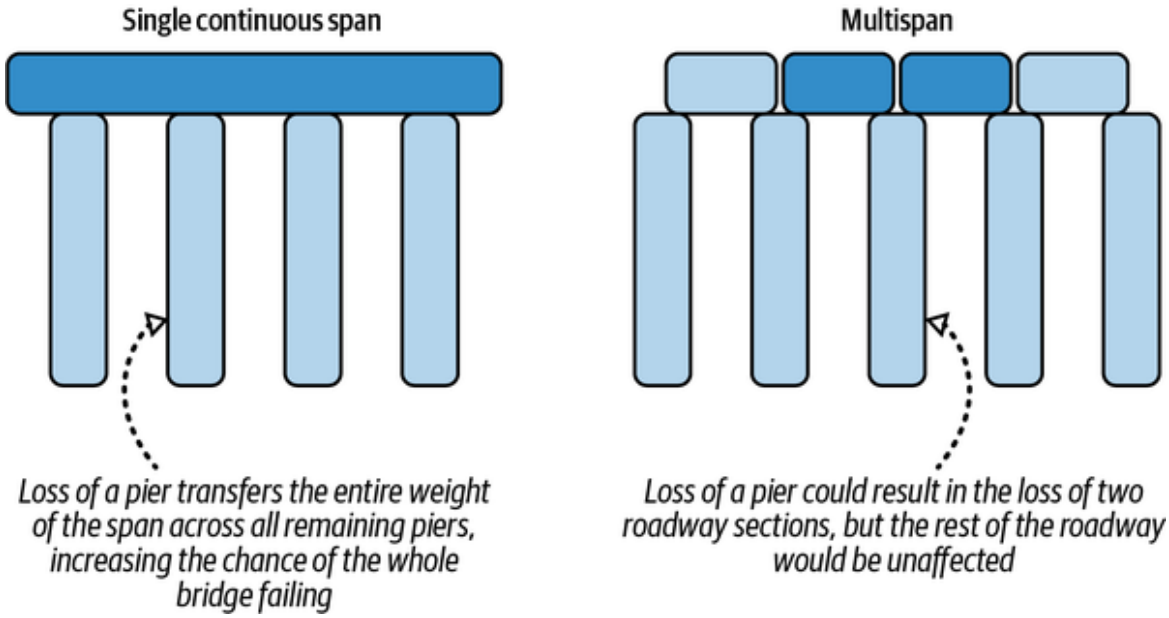


Figure 11-2. Two types of bridge designs—single-span versus multispan

With the Ronan Point tower, each level was supported by the level below it. When the explosion happened, and the outer walls of the 18th-floor apartment's living room were blown out, the floors above were no longer sufficiently supported, and so they collapsed. Most other buildings of a similar (or greater height) rely on a structural frame—if part of the frame fails, the load can be transmitted down to other parts of the frame, reducing the chance of a cascading failure. This was a major recommendation in the wake of the Ronan Point collapse that has subsequently been widely adopted.

In the case of the `Pet` service failure, it's less clear how to make the instances of the service itself more isolated. Fundamentally, they share work, and that same pool of work (the requests coming in) is the thing that links them. If the work could be partitioned, with the `Pet` service having different pools to handle different types of work, the damage of a progressive collapse could be contained.

Consider Figure 11-3, which shows multiple shards of the `Pet` instances, with different shards used for household pets, agricultural animals, and exotic animals. If the shard for the household pets partition suffered a progressive collapse and was taken offline as a result, agricultural and exotic animals would be unaffected.

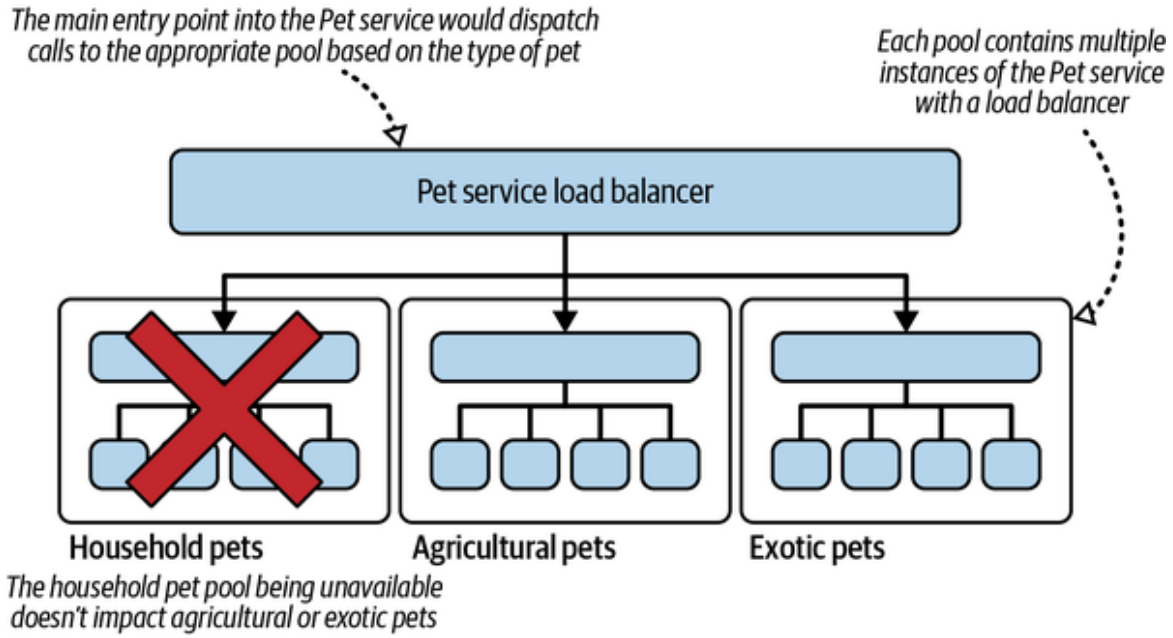


Figure 11-3. Splitting the `Pet` service into separate worker pools

You could also look to the clients of the `Pet` service in order to reduce interconnection. If the `Pet` service was responding in a potentially unsafe way—if it was taking a lot of time to respond, for example—clients could sever their connection with the service to avoid resource contention, something that a circuit breaker could help achieve. Obviously, if clients could no longer make use of the `Pet` service, they'd need some way to degrade their own functionality when the `Pet` service was unavailable while hopefully still providing essential functionality, which would significantly mitigate the impact of the failure.

Services dependent on the `Pet` service would have to think carefully about how they could reduce their interconnectedness to the `Pet` service. If we have a `Pet Sitter` and `Trainer` services, and both of them are dependent on the `Pet` service, they would have to know how to degrade their own functionality in the case of a `Pet` service outage.

An even more extreme form of avoiding interconnection is to have a complete standby system that reimplements the core, critical functionality of a system, but without any shared parts (code included). Another more extreme approach would be to adopt a cell-based architecture, with each

cell having its own instance of a `Pet` service pool. We'll look at cell-based architecture and critical standby systems later in this chapter.

In the case of the AWS outage, many customers started looking at ensuring that their devices could operate independently of the cloud when required. For example, the company behind Eight Sleep smart mattresses is looking to change its products so they can be operated with a local connection, meaning people won't be stuck with overheating mattresses in the wake of the next cloud outage.

For some companies, the discussion turned to shifting to a multicloud approach—the idea being that if AWS suffered a failure, those companies' products could instead make use of alternative cloud vendors. The reality is that a multicloud approach involves all the complexity we explored in Chapter 10, but the complexity *and* costs are magnified. Building skills and tooling to support a new cloud vendor is going to add a lot of expense, and fundamentally you are hedging against a different risk than that of a standard multiregion approach. When using multiple regions from the same vendor, you are hedging against the risk of a single region failing. If you go for a multicloud approach, you are hedging against the risk of an entire cloud vendor failing. There may be other good reasons for a multicloud approach, but in general I'm not convinced it's worthwhile for most disaster recovery (DR) situations.

Patterns for Avoiding Progressive Collapse

So far we've looked at examples of progressive collapse both in the physical world and in the context of a distributed system. As we saw, we can draw parallels between the two domains. With that in mind, let's look at a few practical patterns that can contribute to mitigating the risk of a progressive collapse in our distributed systems and may also deliver some other useful benefits: bulkhead, graceful degradation, distribute across failure planes, cell-based architecture, and critical standby system.

Pattern: Bulkhead

A mechanism to isolate the impact of a failure in order to help mitigate the risk of a progressive collapse.

A *bulkhead* is a mechanism in a system that stops a failure from propagating. As we touched on in Chapter 5, the term itself comes from shipbuilding, where a bulkhead can be used to seal off compartments. If a compartment flooded but the bulkhead doors were sealed, the ship would remain afloat. Our example of a bridge with multiple spans is also arguably an example of a bulkhead in action—the failure of a single span impacts only that span.

In our previous example, I suggested that clients of the `Pet` service could stop sending calls in situations where the `Pet` service was taking a long time to respond, thereby avoiding more problematic resource contention. This is another example of a bulkhead. When looking at a bulkhead through the lens of a progressive collapse, the bulkhead is going to be a very useful mechanism to help reduce interconnections and therefore help reduce the likelihood of a progressive collapse expanding in scope.

How It Works

A bulkhead can be applied at multiple points in a system. For example, with the clients of the `Pet` service, if a circuit breaker trips open to stop calls being sent to a faulty component, then this is a bulkhead—one part of the system is sealed off from another when a problem occurs to stop the problem from getting any worse. Another example could be a single service having multiple resource pools for external services.

More complex forms of bulkheads include cell-based architecture, in which collections of components are deployed together in individual cells that are isolated from one another. We'll look more at that pattern later in this chapter.

When to Use It

Bulkheads are vital in mitigating the risk of a progressive collapse. The reality is that you are likely using this pattern without realizing it—but being cognizant of the pattern should help you see it in more circumstances, and help you see where it can fit in. The tricky part about giving advice on when to use a bulkhead is more down to the fact that the specific bulkhead mechanisms can vary drastically in terms of cost and the resulting increase in complexity for your system. The differential between implementing a circuit breaker and a full-blown cell-based architecture is vast, for example.

A key thing to understand is that a bulkhead by itself is not stopping a failure from having *any* impact on a system. Implementing a bulkhead is about ensuring that a failure doesn't have a *catastrophic* impact on the wider system. In a ship, if a compartment is flooded but the bulkhead doors are sealed, then that compartment is no longer usable (and you'd have to hope there was no one in it at the time), but the ship should still stay afloat. Ideally, you'd want to ensure that a failure within a bulkhead has zero impact on end users, but if that isn't possible, then the next pattern is an important consideration.

Pattern: Graceful Degradation

Reducing the functionality of a system while still continuing to operate in reaction to part of the system being unavailable.

With *graceful degradation*, the system adapts the functionality offered to the user when part of it fails. This is largely about the system knowing programmatically how to change the user experience in order to reflect the reduced system functionality.

How It Works

As a higher-level concept, graceful degradation can be implemented in many different ways. Fundamentally, though, this approach requires two things:

1. Knowing that something is wrong

2. Modifying the user experience to handle this as well as possible

To come back to something we've used before, consider the example of AdvertCorp in Figure 11-4. The `Turnip` service starts failing to the point that a circuit breaker moves into the open state, meaning that requests can no longer be sent. This gives us the knowledge that something is wrong. Next, the AdvertCorp website is updated with a simple message indicating that this functionality is not currently available. This is an example of modifying the user experience.

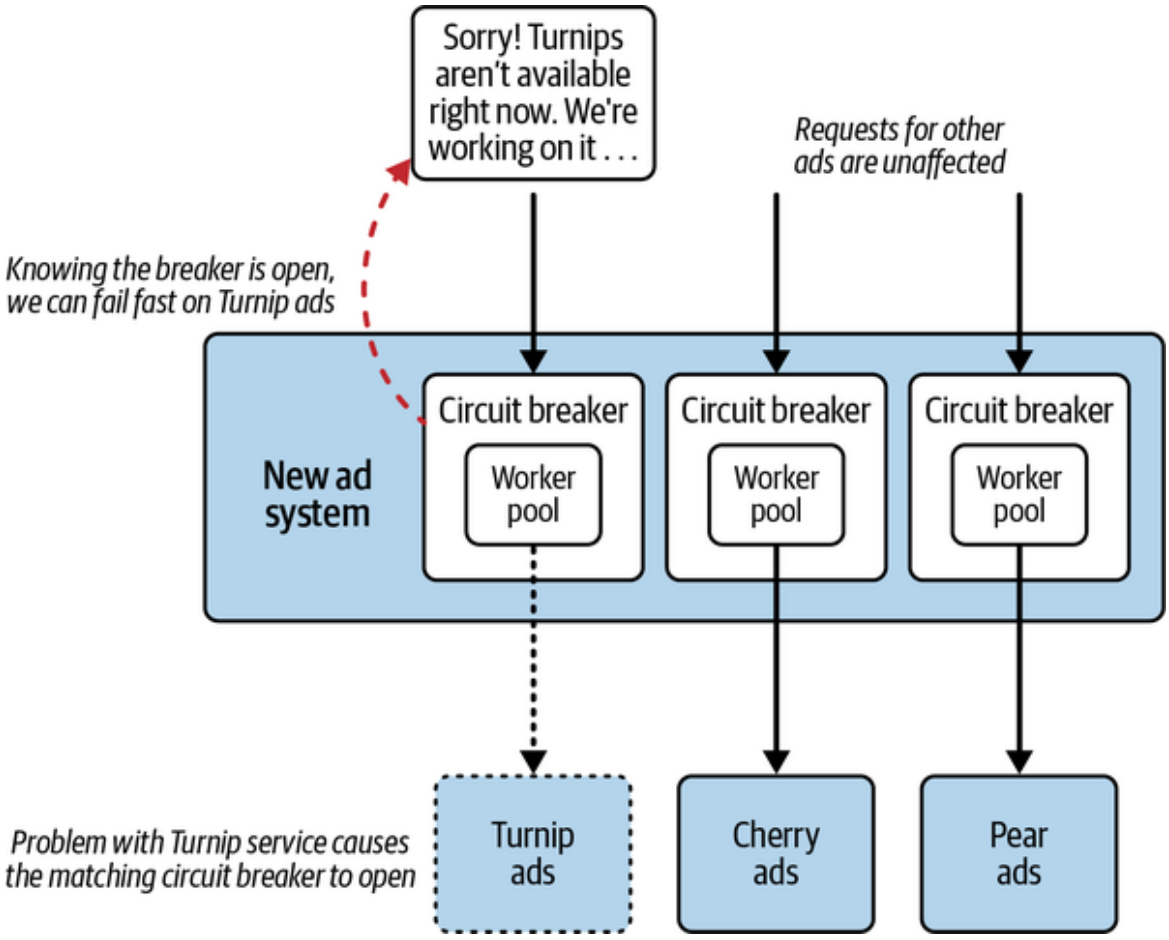


Figure 11-4. Adding circuit breakers to AdvertCorp

Here, the circuit breaker is letting us know that there is a problem, and we react to the circuit breaker state being open by updating the UI. Coming back to the example of having multiple pet pools earlier, we could again update the UI to acknowledge that the functionality for household pets is currently unavailable.

For graceful degradation to work, you have to build into the system the fundamental notion that things can fail and then implement the appropriate behavior for the system to exhibit. In effect, you end up implementing bulkheads around the potentially failing functionality.

When to Use It

For graceful degradation to work, there is an implicit assumption that you can implement bulkheads around parts of your system and can react based on the relative health of the functionality contained in said bulkheads. If you have a system that is currently not decomposed along these lines, then attempting graceful degradation is not going to be possible.

A classic monolithic web application with the entire system functionality baked into a single backend process may struggle to implement graceful degradation. All the functionality is contained in a single process. Failures that impact one piece of functionality are likely to impact the entire process. For example, if the CPU is exhausted, then the whole process is compromised. In fact, one of the reasons that monolithic applications are split apart can be to ensure that critical functionality is separated from noncritical functionality, allowing for concepts like graceful degradation on the backend to be implemented in the first place.

A variation on splitting a monolith around functional lines to make graceful degradation easier is to keep the monolith as a single codebase but allow it to be launched with configurable roles. Incident.io, for example, can run its monolith in different configurations to serve one of eight workload tasks. The monolith can act as a web worker to handle incoming HTTP requests, as a cron worker for periodic tasks, or as pub/sub workers that handle more asynchronous workflows, among other configurations. The implications of the different types of workers failing vary, and each of the workers can have different scaling requirements. This approach manages a neat balance of keeping the deployment topology simpler (as each worker is just a single monolithic process) without as much of the associated complexity that would be required for a finer-grained service approach, such as a microservice architecture.

In a context where you might have a wider business continuity strategy in place, graceful degradation can be useful to ensure that the critical functionality is still carried out. Likewise, graceful degradation makes sense when there's a need to preserve key functionality that is explicitly called out in customer SLAs.

Pattern: Distribute Across Failure Planes

Spread a workload across multiple planes to ensure that a failure in one plane doesn't take out all functionality.

When deploying multiple copies of a service, we can reduce the chance of a single inciting incident taking down all copies of the service by ensuring that these copies are distributed across different isolated failure planes. In Figure 11-5, we see a diagram of pods deployed onto a Kubernetes cluster.

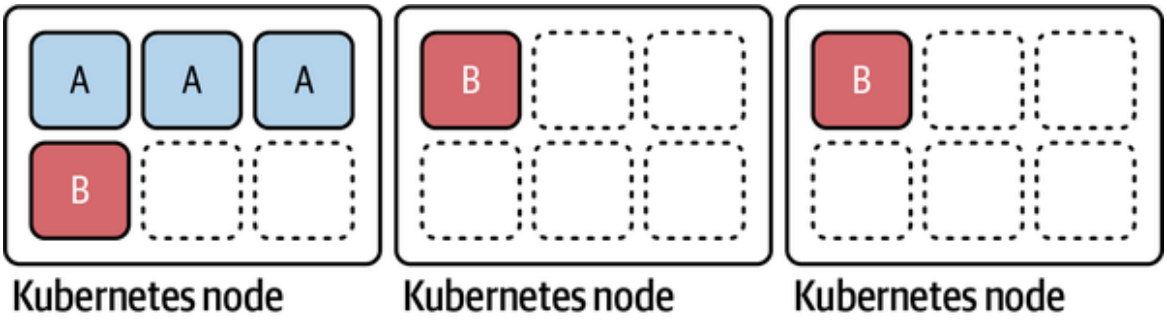


Figure 11-5. Kubernetes deployments across nodes

Each node in a cluster would normally run on a single machine and be considered its own failure plane. If the machine running the Kubernetes node goes offline, all processes running on that machine will also be taken down. Service A has all its copies on the same node, so it is vulnerable to a single node failing. On the other hand, service B has each copy deployed onto a different node, ensuring that the failure of a node won't take down all copies of the service.

Failure planes typically map to physical deployment. Aside from the Kubernetes example, other examples include running workloads across multiple data centers, using cloud AZs, or even running across multiple cloud vendors.

How It Works

There are three main challenges to be solved when running across multiple failure planes. Firstly, you need to create and manage each plane. Secondly, you need to ensure that work can be distributed across each plane. Finally, you need to address any need to synchronize data across the planes.

Creating and managing the failure planes

If we consider the example of Kubernetes or cloud AZs, the cost of managing the separate planes is close to nil from the point of view of the users. Kubernetes is built to distribute work across machines—in this context, each machine is its own failure plane. Kubernetes will make sure that multiple instances get distributed across multiple machines for you. Cloud AZs are likewise managed for you by the underlying cloud vendor. If you were running on your own infrastructure, though, and you decided to build and run a second data center to provide a new failure plane, the cost could be considerable.

Distributing work

To distribute work across the planes, standard load distribution mechanisms such as shared work queues or load balancers will typically suffice. These work distribution mechanisms thus deserve attention, of course, to ensure that they don't become single points of failure. I looked at how message brokers can provide high availability in Chapter 9, and at how this is done with load balancers in Chapter 8.

Synchronizing data

A major challenge is dealing with the need to synchronize data across failure planes. How complex this challenge may be can vary greatly, and it typically becomes more complicated with increasing latency between planes. In the case of nodes in a Kubernetes cluster, it might be enough for each service instance to be able to access a shared database. However, as the planes become more isolated, it might be more appropriate to have local

copies of the data within each plane, with some synchronization mechanism in place.

As the latency between failure planes increases, so too do the challenges of synchronizing the data between these planes. For example, running a single Cassandra ring[6] across multiple AZs in a cloud region would likely be fine (as high-speed connections are available between AZs). But the latency you'd see across cloud regions might be a step too far for your use case.

Increasing isolation

How well this pattern delivers on the promise of reducing progressive collapse comes down to how well the failure planes are isolated. For example, consider distributing Kubernetes workloads across multiple nodes—are those nodes in the same rack in the same data center? If so, you might be more vulnerable to a problem in the data center itself (such as an air conditioner leak) taking out your entire service.

Alternatively, consider AWS AZs. The AZs within a region can be up to 60 miles apart to ensure that situations impacting one locale (such as a fire or flood) are highly unlikely to impact the other zones in that region. However, US-EAST-1 still went down, didn't it? That was because the same code ran in both places—the race condition with the DNS Enactors wasn't isolated to a single AZ. We'll explore the problem of shared code a bit more when we look at the case study of Monzo later in this chapter.

When to Use It

In the vast majority of distributed systems, making use of this pattern is highly desirable. Decreasing the chance of a single incident taking down an entire service always sounds like a good idea. The real question is, how far do you push it? As the degree of isolation increases, as shown in Figure 11-6, so too do the cost and complexity of the solution, broadly speaking.

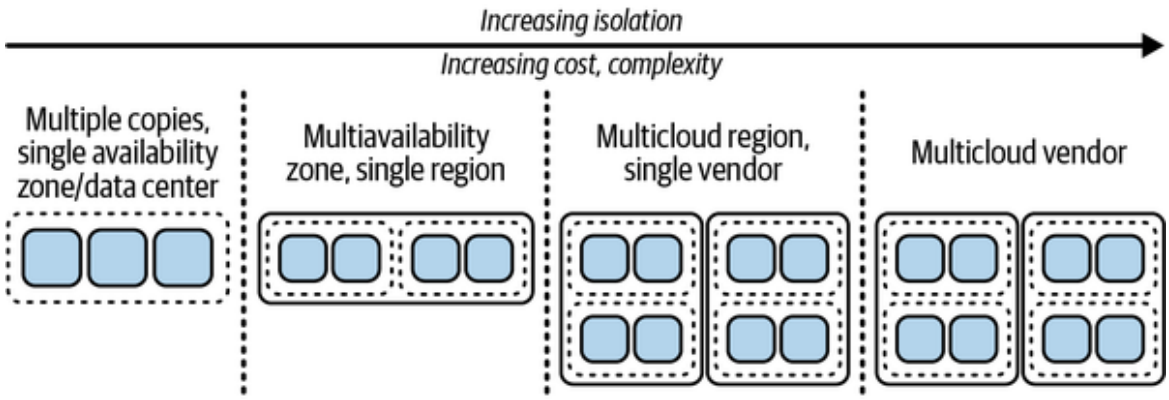


Figure 11-6. As isolation between failure planes increases, so too do cost and complexity

For example, increased geographical distribution means more latency, challenging both work distribution and data synchronization. Running across two cloud vendors would require dealing with two cloud vendors' toolchains; it would also likely increase operational costs in terms of data ingress and egress costs, as well as potentially reducing the opportunity for bulk infrastructure discounts from a single vendor.

Additionally, you typically have to implement the simpler form of distribution across failure planes before moving on to more complex forms. For example, you'd want to have a working solution deployed across multiple AZs in a single cloud region before trying to get a multiregion approach working. This gives you the chance to gradually ramp up the degree of isolation over time.

Pattern: Cell-Based Architecture

An approach in which a collection of collaborating services are deployed and communicate within an isolated failure plane.

Cell-based architecture is an extension of the concept of distributing work across failure planes, but it applies this concept to a collection of collaborating services that will be deployed together, as outlined in Figure 11-7. Each cell, in effect, becomes its own bulkhead.

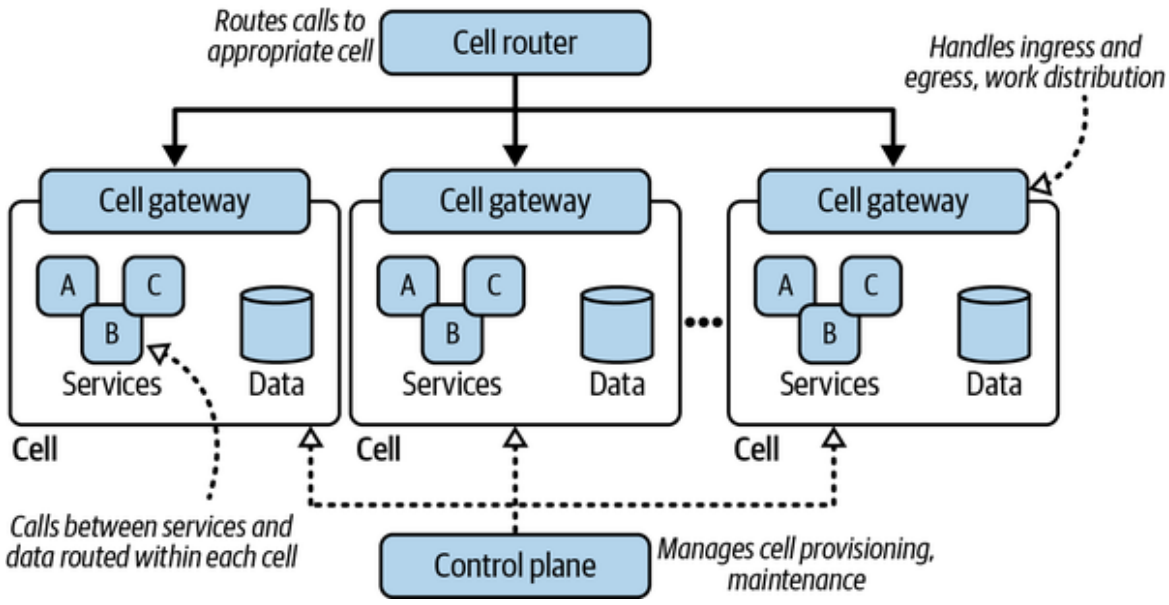


Figure 11-7. Cell-based architecture

One of the challenges around distribution across failure planes is the impact of latency in situations where services need to communicate across planes, or where data needs to be synchronized. Cell-based architecture deals with this by ensuring that communication between services inside a cell is routed inside the cell itself.

How It Works

When implementing a cell-based architecture, there are a few parts to consider, as outlined in Figure 11-7. In brief, they are as follows:

Cells

A collection of collaborating components, typically comprising a mix of services and data

Cell router

Directs inbound calls to the appropriate cell

Cell gateway

Allows call ingress and egress to a specific cell

Cell-local routing

: Ensures that calls between components in a cell stay within a cell

Control plane

: Manages cell provisioning and general maintenance

I'll take you through each of these in more detail before looking at how this approach can help in terms of resilience.

Cells

A cell is a logical grouping of collaborating components deployed within a bulkhead. In our example in Figure 11-7, each cell contains services A, B, and C, along with any data managed by those services. Within each cell, these components may well have multiple copies to deliver some cell-local redundancy in the case of a component failing.

Cell router

The cell router directs work to the appropriate cell. How challenging this is depends largely on whether a specific cell is required to handle a specific piece of work. If any cell could handle the work, then something simple like round-robin DNS might suffice.

If the work requires specific cell-local state (for example, the work might pertain to a specific user whose data lives in a cell), then the work item itself would need to contain enough information to allow the router to direct it to the appropriate cell. For example, if an HTTP request contained a tenant ID, then the router might be able to map this to the appropriate cell, likely using a consistent hashing algorithm.

In situations where the cells are geographically distributed, you may then need to route the call to an available cell in the required locale. This could be done by making use of geography-specific DNS servers. If it was simply a case of routing calls to a cell that was near the client for improved latency

reasons, then making use of the Border Gateway Protocol (BGP), similarly to the way CDNs route calls to local caches, would be appropriate.

Cell gateway

Once a work item arrives at the cell gateway, the gateway routes the work accordingly. If it was something like an HTTP request, then the gateway would route the call to the appropriate service. If the work item was a message, then the gateway might just be a cell-specific message channel.

The cell gateway would also be responsible for egress—that is how data leaves the cell.

Cell-local routing

Calls between components in a cell need to somehow be routed locally within the cell, rather than calling components in other cells. Without this, the cells are limited in their usefulness as a bulkhead. This routing will typically be done using some form of cell-local service discovery. For example, if service lookup is done using DNS, each cell could have its own DNS server.

Control plane

A cell control plane is used to manage cells. This would include provisioning or resizing cells, shutting cells down, or rolling out software updates to cells. This could be anything from a set of scripts to a full-blown orchestration system. At present, this pattern seems niche enough in its use that approaches to the cell control plane are extremely varied.

Cell resiliency

For a cell-based architecture to deliver resilience in terms of availability of service or durability of data, there are a few things to consider. The first port of call would be for each cell to provide a degree of resiliency itself. This would likely mean running multiple copies of services to maintain availability of a service within that cell, as well as providing any

appropriate data replication within the cell to maintain durability of the data.

It gets trickier when we consider what happens if an entire cell becomes unavailable. The obvious solution is to allow work to be sent to other cells, but that becomes difficult when considering cell-local data.

If a cell contains specific tenant data, for example, and that cell becomes unavailable, then all work related to that tenant could fail. A desirable alternative would be for that work to be routed to another cell, but then you'd need a mechanism for data to be distributed between cells.

Distributing data between the cells to an extent undermines our desire to keep the cells isolated from one another. You'd have to be careful to ensure that whatever replication technology you pick for data replication doesn't itself become a vector for a progressive collapse.

If the work to replicate data and route calls to other cells in the case of a cell failure is too great, then you of course have the option of just accepting the outage. The loss of a single cell would obviously be less than ideal, but if only part of your user base is impacted, then this might be acceptable.

Case studies

Nubank is an example of an organization that uses this pattern extensively to help it support its 127 million global customers. Laís Oliveira's talk "One-to-Many Products, One-to-Many Countries" from QCon London 2026 does a great job of exploring the benefits and challenges of this architecture from Nubank's point of view. Note that in her talk Oliveira refers to Nubank's cells as "shards," but sharding is used in lots of contexts—from Oliveira's description of Nubank's architecture, it seems to fit the definition of a cell pretty well. Both DoorDash and Slack also make use of variants of a cell-based architecture.

When to Use It

There is something almost paradoxical about a cell-based architecture. At first glance, it presents a very simple and easy mechanism for scaling a

system while also delivering a degree of fault tolerance. However, this simplicity hides a lot of complexity required to implement this pattern well.

The challenge of implementing an appropriate control plane for a cell-based architecture should not be overlooked. The uncommon use of this pattern and the variation in how the cells themselves are constructed mean there isn't really any existing dedicated software for managing cell-based architectures. Instead, you will likely have to build this yourself, perhaps leveraging preexisting tools and services for automation of the infrastructure and deployment of your software. This will not be a trivial activity and is something I would suggest you attempt only when you already have a high degree of maturity in automating your infrastructure.

It's worth noting that a cell-based architecture can work as an effective bulkhead mechanism to defend against certain types of failure propagation, but not all. For example, a bug in code that causes a catastrophic failure in one cell could well recur in another cell if the same version of the code is running there. This is not just a hypothetical concern—in Oliveira's aforementioned talk about Nubank's cellular architecture, this was covered as an issue that Nubank has had to deal with.

Cell-based architecture has a host of other potential benefits that may well justify the additional complexity. For example, limiting the scope of credentials to a single cell could have benefits from a security point of view, data sovereignty concerns could be alleviated by citing cells in different jurisdictions, and just adding more cells can be a quick way to scale up to handle more users. However, when viewed narrowly through the lens of system resilience, I consider the additional complexity required to implement this pattern well to mean that there are probably a host of things you should look at first when seeking to improve the resilience of your system.

Pattern: Critical Standby System

An alternative system providing only critical functionality that shares as little as possible with the primary system and is used in failover scenarios.

With a *critical standby system*, you create an alternative implementation of the primary system's most vital functionality that can be used when the primary fails, as shown in Figure 11-8. The idea is that the alternative implementation shares as little as possible with the primary system, so a problem with the primary is much less likely to extend to the standby.

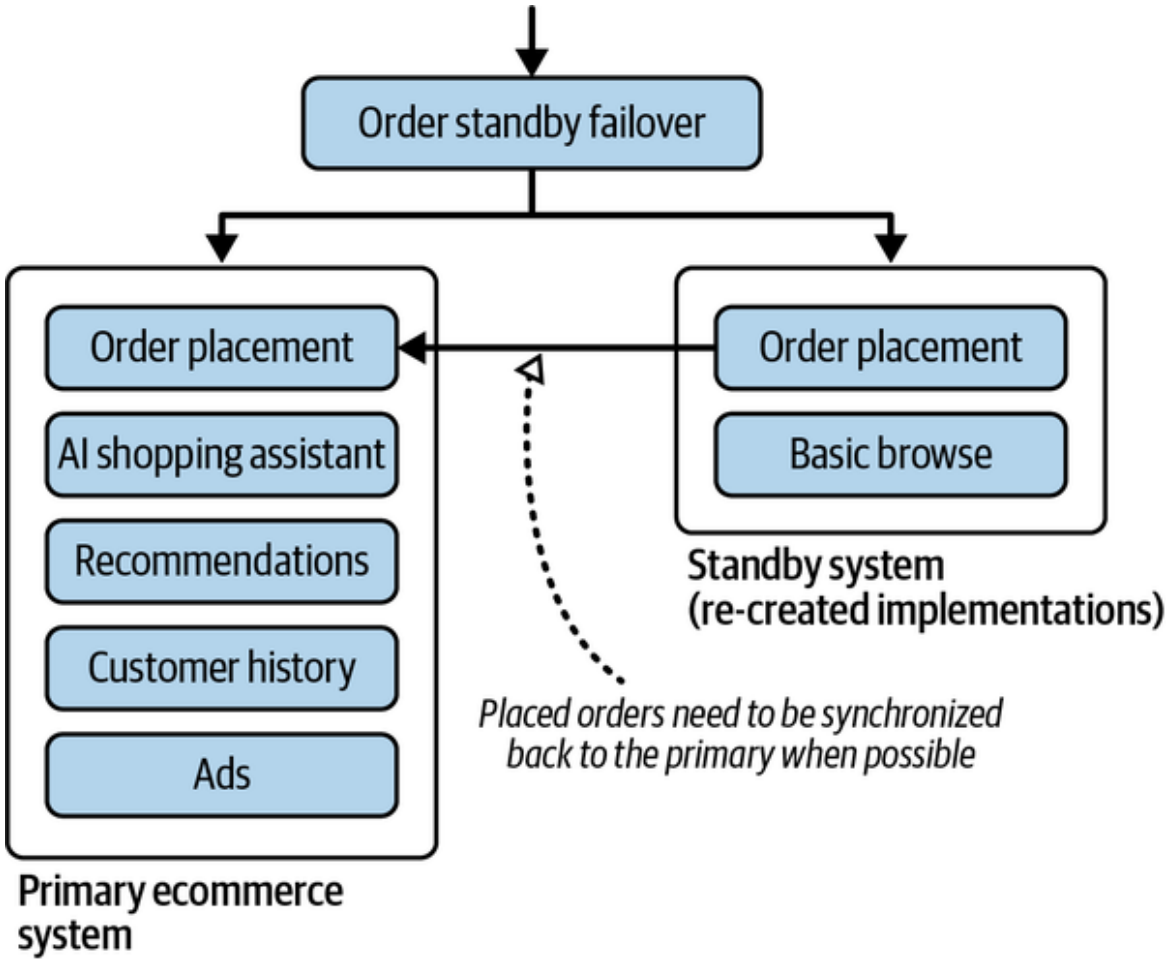


Figure 11-8. A standby system with only critical functionality can be brought online in emergencies

The less that is shared, the less the primary and the standby are connected, and therefore it is less likely that a progressive collapse will bring down the entire system. In this example of an ecommerce system, only functionality required for order placement is considered critical. So a more basic ability to browse products is available, as well as the ability for an order to be placed. However, features such as ads, a customer history, and an AI shopping assistant are not considered critical and thus are not reproduced in the standby.

The critical standby in a system takes an aggressive approach to reducing interconnectedness. The goal is to share as little as possible with the primary so as to ensure that a problem with the primary is less likely to also impact the standby. A bug in the code that takes down the primary won't take down the standby, as the standby uses different code. Noncritical features cannot cause any cascading issues, as noncritical features are not present in the standby.

How It Works

These standby systems will focus only on the critical functionality that needs to be maintained, as the cost of reimplementing all the functionality of the primary will be too high. Many years ago I was part of the team that helped rebuild the *Guardian* website. In the event of the main system being down, it was vital that articles could still be made available. So periodically a crawl of the entire public website was carried out, which created a read-only view of the articles. The *Guardian* could then switch over to the backup of the articles when required. As I mentioned in Chapter 5, the *Guardian* standby system allowed people to read the news but didn't provide more dynamic functionality. If the articles needed to be changed while the primary system was down, then the static HTML pages would have to be edited by hand. Not ideal, but the standby shared virtually no functionality with the primary system, so a bug that took down the primary would be highly unlikely to also take down the standby.

As you decrease the amount of commonality between the primary and the standby, you decrease the extent to which the two are connected—but the price also increases. For example, if you ran the standby on a different cloud provider, you would have the additional cost of building tooling and experience to work with both cloud vendors, but you'd hedge against the risk of a total outage of your primary vendor. If you don't share any code between the primary and the standby, then you decrease the chance of a bug or a security issue impacting both systems, but you have to factor in the cost to reimplement the critical functionality from scratch.

This leads to tricky choices about what functionality should be replicated in the standby. In the *Guardian* example, the standby allows for articles to be read, but editing articles becomes painful, and more dynamic content like reader comments wouldn't be possible.

When implementing a critical standby system, it's important to remember that the complexity increases if we expect bidirectional state synchronization. With the *Guardian*, the standby was just a static backup of the articles generated from the primary. If changes were made manually to the standby article content, they would also have to be manually applied on the primary if we wanted to keep the primary and the secondary in sync. Things can get a lot more complex if you need two-way synchronization of state, especially when it applies to data that we really want to be correct—people's bank account balances, for example.

Monzo, a consumer bank in the UK, has a critical standby system for its core banking operations. If its primary system is impacted, the stripped-down standby still allows for people to check their balances, spend on cards, and withdraw money. The standby system shares no code with the primary, and it has a very different architecture. The primary system runs on AWS on over 3,000 services. The standby platform, which implements only the bare minimum functionality, runs on just 18 services and makes use of Google Cloud.

Monzo's standby system is capable of switching over to handle all payment processing for the company in around 15 seconds, leaving a very small window for failed payment processing.

A lot of the complexity in Monzo's case is the need to perform two-way synchronization of data between the standby and the primary. With Monzo, information like account transactions are vital to record—payments made in the standby will need to flow back into the primary.

Ensuring that the standby system is working properly is often the tricky part of this approach, as hopefully the standby will only very rarely be used. It would be a nasty compounding issue to find that your primary system fails, and then issues with the standby stop it from functioning as well.

Monzo solved this challenge by opting in a small number of users to use the standby each day, albeit allowing them to opt back out to get the full experience.

When to Use It

It's important to note that a critical standby system should not be your sole technique to ensure that critical functionality can be provided. More basic techniques such as having redundant service instances and having multi-AZ cloud deployments would be things I'd expect you to be doing in your primary first, and I certainly hope you would explore some of the simpler DR options we looked at in Chapter 10 before reaching for this particular pattern.

One of the major challenges of implementing a critical standby is the cost. Monzo shared that the cost of running the standby is only 1% of the cost of the primary platform, but I have to assume this comparison applies just to the cloud costs and does not include the cost to initially rewrite the critical functionality and to maintain the system.

In situations where the critical site functionality can be reproduced in a standby quite cheaply, such as in the *Guardian* example, the case for the standby becomes easier to consider. When the cost of the standby starts increasing, it then comes down to a discussion about the impact of the outage. Monzo operates in a highly regulated sector, and its customers need access to their money. If I couldn't check the price of a TV for an hour or two, it wouldn't be the end of the world. But if my bank went down for an hour at the wrong time, it could be disastrous.

So, in short, you should do your best to make sure your primary doesn't fail in the first place. Then you should look at more conventional DR options. But building an alternative standby might make sense if the cost of implementation is low, or if the other DR options don't address all the risks you are worried about.

Summary

Things can snowball. Small problems can become massive all too quickly. In this chapter, we looked at examples of this happening in the physical world and identified three key approaches to mitigating a progressive collapse—namely, reducing hazards, strengthening components, and reducing interconnection.

Then we looked at a number of practical patterns that can help us implement these approaches in our distributed systems. Bulkheads create separation between our components, reducing interconnectedness. We can further reduce the interconnectedness between our systems by distributing them across failure planes, perhaps going so far as to implement a full cell-based architecture. When things do go wrong, we can make use of graceful degradation to ensure that parts of the system do work, and we may even create dedicated standby systems for our critical functionality.

A few times throughout this chapter, and previously in this book, we've looked at the trade-offs that need to be made when things go wrong. That is a topic we'll be exploring in more depth in Chapter 12.

---

1 Bruce R. Ellingwood et al. *NISTIR 7396: Best Practices for Reducing the Potential for Progressive Collapse in Buildings* (National Institute of Standards and Technology, 2007).

2 National Transportation Safety Board, *MIR-25: Safeguarding Bridges from Vessel Strikes: Need for Vulnerability Assessment and Risk Reduction Strategies* (NTSB, 2025).

3 This is a small simplification of the events that led to the outage.

4 No cats were harmed as part of this outage, at least to my knowledge.

5 National Transportation Safety Board, *MIR 25-40: Contact of Containership* Dali *with Francis Scott Key Bridge and Subsequent Bridge Collapse* (NTSB, 2025).

6 A *ring* is the term given to a collection of Cassandra nodes operating as a single distributed database.

Chapter 12. Using Formal Theorems to Understand Trade-Offs

Essentially, all models are wrong, but some are useful.

—George Box and Norman Draper

When things go wrong, we need to make sure that as many things as possible go right. Typically this means that we need to gracefully degrade our functionality in some way. We've looked at a number of examples of this approach throughout this book, including things like load shedding to get rid of excess load or using bulkheads to protect us from parts of the system that are malfunctioning. But we can make some fundamental trade-offs in the design of the system itself to ensure that when things go wrong, our system handles it in the way that is best for the given context.

In this chapter, we'll look at two formal theorems, CAP and PACELC, which describe the trade-offs that have to be considered when parts of a distributed system are unable to communicate. Put another way, these two models help us understand what options we have to gracefully degrade our application when something goes wrong. We are also going to look at a concept known as Little's Law, which will help you reason about how much load your system can handle and about how to keep load to manageable levels.

While all these concepts can appear at first to feel quite academic and removed from the software you build, I'll share some examples of these models in action to help you see that in reality they are just formal definitions of fairly straightforward ideas. I also say upfront that this is not intended to be an exhaustive deep dive into all the nuances of these concepts. Instead, I want to share these models and give you just enough information about them to decide if you want to learn more.

CAP is the better known of these models, while PACELC is in effect an extension of CAP. So it makes sense to start with CAP.

THEORIES AND THEOREMS

In this chapter, I cover three theorems—CAP, PACELC, and Little's Law. A *theorem* describes a theory that has a formal mathematical proof that confirms the concepts behind it are correct. All these ideas started as theories but were later formally proven, often by people other than those who came up with the original ideas.

A fuller discussion of the mathematical proofs of these theorems is way beyond both the scope of this book and my capabilities, but for those inclined to dig deeper, you can read Seth Gilbert and Nancy Lynch's proof for CAP[1] and Wojciech Golab's proof for PACELC.[2] Little's Law actually has multiple proofs, but I'm led to believe (by Wikipedia!) that Shaler Stidham's proof from 1972 is the most intuitive.[3]

CAP Theorem

CAP theorem states the trade-offs that have to be made when one part of our system can't speak to another—or, more technically, it states what happens when our system suffers a *partition*.[4] Think of a partition as a split in our distributed system: without a partition, all the machines on the distributed system are connected. When a partition occurs, one or more of those machines are now isolated from the rest.

A partition could be small- or large-scoped. For example, it could occur when a single machine is unavailable, when all instances of a service are offline, or perhaps even when an entire data center is unreachable. A partition can also be short- or long-lived. It could occur when a call to a service times out, for example—or it could persist for days, perhaps caused by a data center fire or a local flood.

CAP stands for three guarantees: *consistency*, *availability*, and *partition tolerance*. The theorem tells us that when a partition occurs, we get to pick two of these guarantees. It's important to note that although the terms *consistency* and *availability* are widely used in lots of contexts in computing, each has a very specific meaning in the context of CAP theorem. So to clear up any misunderstanding, here are the formal definitions of these three guarantees:

Consistency

> Every read receives the most recent write or an error.

Availability

> Every request receives a (non-error) response, without the guarantee that it contains the most recent write.

Partition tolerance

> The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

So with those guarantees in mind, let's look how these are traded off against each other.

Pick Two—Sort Of

As mentioned, CAP theorem states that when a partition occurs, you'll have to pick two of the three available guarantees. What this means is that you can have a system that is consistent, available, and partition tolerant, right up to the point where a partition occurs. So confusingly, it can look like you have all three, right up until your ability to tolerate partitions is tested.

The benefits of consistency and availability seem clear. So if CAP theorem states we can pick only two, why not just pick those two and give up on partition tolerance?

Partition tolerance means we can continue to operate despite messages being dropped or delayed. Remember our rules of distributed systems in Chapter 1? Rule 1 states that information cannot be sent instantaneously between two points. In other words, it takes time to transmit information. Rule 2 states that sometimes you may not be able to reach the thing you want to talk to. So to build a distributed system is to accept that messages may be delayed or even dropped.

To give up on partition tolerance would mean, fundamentally, that we are giving up on our system's ability to handle any partitions. But we are building a distributed system, in which we *must* have the ability to handle partitions. So you can never give up on partition tolerance.

In practice, that means we need to favor either consistency or availability when a partition occurs. It is worth highlighting, though, that you can make different trade-offs for different parts of your systems; we'll expand on that idea later in this chapter.

Let's look at a worked example so we can put these ideas into a real-world context and see how they might get applied.

Example: Online Record Shop

In Figure 12-1, we can see a simplified example of an ecommerce system.[5] The `Order` service is responsible for making sure an order gets placed. The `Inventory` service knows if the item is in stock, the `Catalog` lets us know how much the item costs, and `Payment` is where we take payment from.

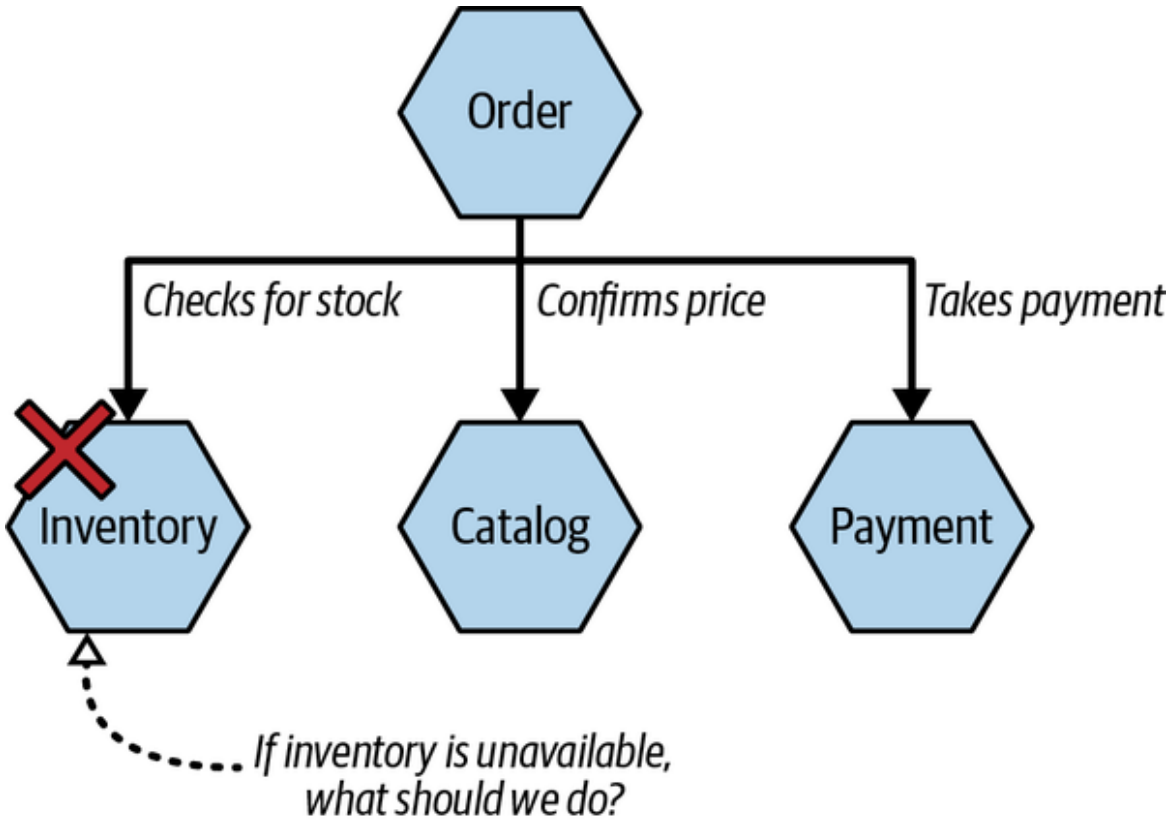


Figure 12-1. A simplified ecommerce architecture

Let's imagine that just as we are about to place an order, the `Inventory` service is unreachable. Putting CAP theorem to one side for the moment, this scenario gives us a simple question to ask: should we still sell the item?

Sell the item!

In spite of not knowing whether we have the item in stock, we could decide to go ahead with the order anyway. This means that we'd have to deal with the implications of the item not being available when we decide to ship it, but that might not be too bad.

Firstly, it's possible that, even though we aren't sure if we have the item, we do in fact have it in stock. So when we go to dispatch the item, we have it on hand, the order goes out, and the customer is unaware there was ever an issue. And we make a sale!

What happens, though, if it turns out that we don't have the item? Well, we have a number of ways to deal with this scenario. We could let the customer

know it's out of stock and offer them a refund, or we could order a replacement and/or let the customer know that the order will take longer. The customer might not be happy about this, but if it happens rarely enough, there may not be any lasting damage to our company's reputation.

Coming back to CAP theorem, if we choose to sell the item, we do so knowing that we have an inconsistent view of stock, and we have instead favored our system's *availability* in terms of order processing during this partition.

Don't sell the item

If we decide not to sell the item because we aren't sure if it's in stock, then we can't go ahead with order processing, and we don't make a sale. The customer is likely unhappy, as they wanted to buy something but can't. We are also unhappy, as we didn't make a sale! Worst of all, we may actually have had the item, after all, but the customer ended up buying the item from somewhere else.

By choosing not to sell the item, we have decided that having a consistent view of our stock is more important than making order placement available. So from a CAP point of view, we have decided to pick *consistency* over availability.

In the real world

If you spoke to most stakeholders in an ecommerce business about which of the two choices was more sensible—selling or not selling the item—you'd get a pretty clear response: the majority of them would be all for selling the item.

The reality is that if the item is in stock and the sale goes through, the customer is happy. If the item is not there, then we can easily compensate the customer. We really don't want the customer to go elsewhere!

Another real-world consideration factors into why in ecommerce systems we would let the sale go ahead even if we weren't sure about our stock levels. Much of what we've talked about is the electronic world—bits and

bytes stored in memory. We talk about consistency in an almost childlike fashion; we imagine that within the scope of the system we have created, we can stop the world and have it all make sense. And yet so much of what we build is just a reflection of the real world, and we don't get to control that, do we?

Let's revisit our inventory system. This maps to real-world physical items. We keep a count in our system of how many albums we have in our warehouse. At the start of the day we had one vinyl copy left of *Getting Killed* by the band Geese. So we can sell it, right? But what happens if while the order is being picked from the shelves, the album gets knocked onto the floor and is stepped on and broken? We have no copies of the album left, so we cannot fulfill the order.[6]

Our system has to have support to handle situations like this. Even if we decide to favor consistency in our digital system, we must have support for handling an inconsistent view of our stock anyway! So if we have to do that either way, we could just make our lives easier and favor availability, which is likely to translate into more sales.

So it's the real-world context that points us in the right direction. Now let's change the context and look at another example and see what choices we might make.

ECOMMERCE IN ROMANIA

When delivering a workshop for a team in Romania, I shared the ecommerce example while we were discussing CAP. This led to a lot of confusion among the workshop attendees, which I initially assumed meant I had done a bad job of explaining the scenario. To paraphrase one of the attendees, "But that's not how this works!"

On further discussion, I realized very quickly that in Romania it was much more common for a customer to pay for their order only when the goods were delivered. The couriers took payment, often in cash. Although this trend is slowly changing in Romania, paying on delivery is still very popular, making up 60–65% of all online ecommerce. Romania is far from an outlier here, with pay-on- or pay-after-delivery models existing across the EU; in Germany, a significant portion of online sales still are paid either on delivery or afterward, via invoice.

Once again, context is vital. A different set of customer preferences and behaviors leads to a different system design, and likely a different set of trade-offs to consider in the event of a failure.

Example: Concert Ticketing

In Figure 12-2, we see an example that is virtually identical to the scenario outlined in Figure 12-1. The difference is that here we are selling tickets rather than records.

In this situation, would we make the same decision and still sell the ticket even though we weren't sure we had one available? Or would the context of this example push us in a different direction?

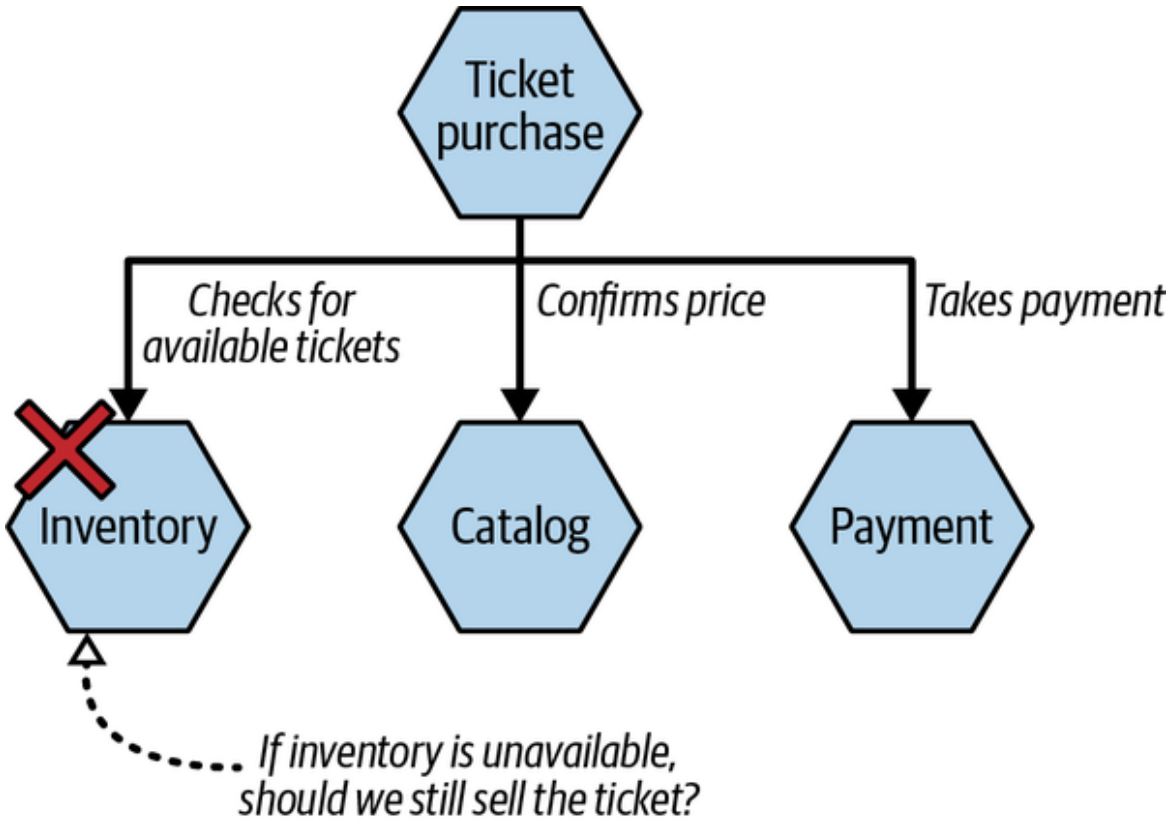


Figure 12-2. A simplified concert ticket system

Sell the ticket!

Letting our capitalist desires run wild once again, we sell the ticket without knowing if we have one available. So we're favoring availability. Is that a good idea?

Well, what's the worst thing that could happen? Picture the scene: A fan of the band Geese[7] has secured a ticket to see their new favorite band. They're excited! *Very* excited. They go on social media to say how excited they are; they book travel and a hotel near the venue. And then we have to send them an email saying, "Er, sorry, you don't actually have a ticket!"

I'm sure the fan will understand, right? As if.

Don't sell the ticket

Alternatively, sanity might prevail. Should we sell the ticket and run the risk of rabid Geese fans flaming us online and threatening legal action? The demographics of the band's fans seem to suggest that many of them are old

enough to be fully qualified and experienced lawyers who could easily make our lives miserable.[8]

And besides, if we don't sell the ticket on this occasion, are we going to miss out on a sale? If we decide to pick consistency and therefore decide not to sell the ticket during a partition, what are the fans going to do? They can't get a ticket from elsewhere! We have the tickets, and no one else does. So the fans will come back to us later.

In the real world

In the real world, ticket purchasing systems are going to favor strong consistency. This causes some challenges around managing systems at scale, which we will explore in more detail when we revisit this example later in the chapter.

It's Not All or Nothing

Our system as a whole doesn't need to make the same trade-off between consistency or availability. In our online record shop example, we could decide that we don't want to go ahead with a sale if the `Payment` service is unavailable, since we'd have no way of knowing if the customer has a valid means of payment. So in that context we would be picking consistency. But we might be happy to sell the item when the inventory is unavailable for the reasons already discussed, in which case we'd be picking availability.

But even individual services don't need to have a fixed view of favoring either consistency or availability.

Let's imagine that we have a `Points` service, where we store records of how many loyalty points our customers have built up. We could decide that we don't care if the balance we show for a customer is stale, but that when it comes to updating a balance, we need it to be consistent to ensure that customers don't use more points than they have available. Is this `Points` service favoring consistency, availability, or both?

At this stage, I think it's worth stepping back and remembering that the point of us examining CAP theorem is to help guide us toward ways in which we can gracefully degrade our system when something goes wrong. At that level, it seems more obvious that we want to degrade our functionality in different ways in different contexts.

You'll often see posts about people "beating" CAP theorem. They haven't. What they have done is create a system in which some capabilities favor consistency, and other parts favor availability. The mathematical proof behind CAP theorem holds.

CAP THEOREM VERSUS HARVEST AND YIELD

In the Fox and Brewer paper cited earlier in which CAP theorem was first detailed, the authors present an alternative model for examining how a distributed system can gracefully degrade—*harvest and yield*. *Harvest* defines the completeness of the data returned from a response. If you return all the required data, your harvest is 100%. *Yield* is the probability that the request will be successfully completed.

Yield can be reduced in the case that the system cannot guarantee that all the data can be provided. So in our record shop example, yield would be reduced if we cannot check inventory. Harvest is reduced if we allow the order placement to go ahead, accepting we don't have all the data.

Whereas the trade-offs around CAP are quite binary, the harvest/yield concept gives us a more incremental model to work with. When yield is reduced, it's not clear that it goes to zero—it might degrade from 100% to 95% for a period and then go back up again. In larger-scale systems, it's common to see some fluctuation around both harvest and yield, and this is why some teams working on systems like these will use these concepts to actively track how the systems are behaving.

Although the harvest and yield model was presented in the same paper as CAP, it's not clear that the concepts are even compatible with the concepts of CAP itself. Where CAP is absolute, harvest and yield is more fuzzy, which I suspect explains why it is less well known. This framing from Marc Brooker, though, points the way for how harvest and yield can be used with CAP (or, by extension, with PACELC):

> *In any case, harvest and yield isn't really a CAP replacement. The CAP theorem is one boundary of the space of possible designs, a fence that can't be crossed. Fox and Brewer's ideas are more about the shape of the landscape inside the fence. That's useful knowledge, but it's really in a different category from CAP.*

PACELC

CAP is a widely used model, but it is not without its flaws. PACELC, an extension of CAP, was proposed by Daniel Abadi to address what he saw as a limitation of the CAP model.[9] Whereas partitions are relatively rare in a modern distributed system, the trade-off between latency and consistency is something we always have to deal with, and this is missing from CAP entirely.

PACELC is somewhat clumsily named (it's not as neat an acronym as CAP is, to my mind), but as with CAP, the name is made up of a number of properties. The theorem states that if a partition (P) occurs, then we have to decide between availability (A) and consistency (C), or else (E) we have to decide how to trade off between latency (L) and consistency (C).

This decision tree of trade-offs is detailed in Figure 12-3, which also shows some of the examples we've looked at and how they might map to this example. The left hand side of the diagram details the same trade-offs as they apply to CAP.

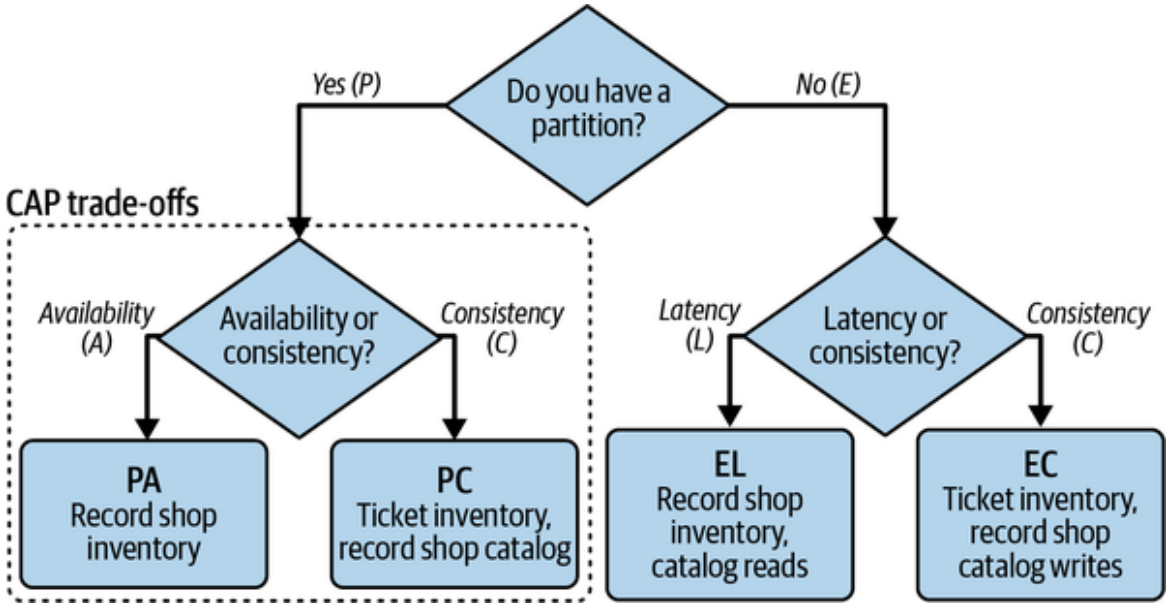


Figure 12-3. The four states for the PACELC model

Given that there is one set of trade-offs when in a partition and another set when *not* in a partition, we end up with four potential configurations for our

trade-offs:

PA/EL

Prioritize availability in a partition, and prioritize latency when not in a partition—for example, online record shop inventory.

PA/EC

Prioritize availability in a partition, and prioritize consistency when not in a partition—for example, online record shop catalog writes.

PC/EL

Prioritize consistency in a partition, and prioritize latency when not in a partition. This is a very rare set of trade-offs to make, with Yahoo's PNUTS database being the only system I could find that uses this configuration.

PC/EC

Prioritize consistency in a partition, and prioritize consistency when not in a partition—for example, record shop's catalog, ticketing system's inventory.

It's worth noting that, just like CAP, PACELC doesn't have to be all or nothing. You can make a different set of trade-offs for different parts of your system, and one piece of functionality might vary which trade-off it picks based on context; in Figure 12-3, you can see that variation with the `Catalog` service, which we'll look at shortly.

Latency Versus Consistency

So with PACELC, when we have a partition, CAP theorem is at play. If there is no partition, we have a trade-off between consistency and latency to consider. But why is *that* the trade-off? At this point, it might be helpful to recall our exploration of message brokers in Chapter 9 when we were examining how messages are written. As I explained there, if your broker

consists of many machines, then when you write a message to a broker, does the write block until all the nodes in the cluster are updated? If so, the write will be delayed until all the nodes report back. On the other hand, if you return the write once a single node has the message, then you have to accept the potential for other nodes in the cluster to see an inconsistent set of messages.

This is fundamentally a trade-off between consistency and latency. When we return after the first node has the message, we are deciding that we want a faster response and accepting that consistency might suffer.

So with this basic understanding of PACELC, let's pick up our examples of the online record shop and ticketing companies from before and expand our discussion to consider how PACELC might highlight some new trade-offs to consider.

Record Shop and Ticketing Examples Revisited

Coming back to two of the key services involved in our record shop example, let's look at `Inventory` and `Catalog`. We've already made the case that we can be somewhat relaxed about our precise stock levels. As a result, when the call is made to check `Inventory`, we could allow the read to return more quickly, even if that read isn't guaranteed to be up-to-date. This could be achieved by allowing a *dirty read*, in database terminology—a write transaction to update that the stock level could be in progress but has not yet been committed, but we allow the read transaction to return the value anyway. This is an example of improving the latency of the `Inventory` service at the cost of consistency.

With `Catalog`, we'd like to give the right prices if we can, but if we can't, it's not the end of the world. Thus during a partition we might allow for stale catalog information, but in the absence of a partition, we would want to aim for consistency. So if the `Catalog` is backed by a simple DB, we would disallow dirty reads—if that record is being updated when our read comes in, the read would be blocked. Consistency is preserved at the cost of latency.

With respect to the ticketing company, we know that we need a strong consistency for the ticket inventory itself. This means that in a nonpartition state, we will also sacrifice latency.

I quite like PACELC as an "expansion" of CAP. As Abadi states in his original post on the topic, our systems are most commonly in the "no partitions" space, shown at right in Figure 12-3. Whereas CAP provides no information about the trade-off space there, PACELC does. I think PACELC also highlights why strongly consistent systems can be harder to build—latency is often going to suffer, and as we'll explore shortly, this will also reduce your ability to handle larger loads.

Latency Versus Consistency and Resiliency

Initially, it might seem like the trade-off between consistency and latency doesn't have much impact from a resiliency standpoint; after all, if we have no partitions, everything is working well, right? However, remember that on several occasions I've discussed the importance of keeping latency low where possible. This is because if operations complete more quickly, we can free up computing resources—remember that our third rule of distributed systems states that resource pools are not infinite. So making sure we reduce the chance of us exhausting our noninfinite resource pools is a good idea.

If your system can tolerate a degree of inconsistency even when there are no partitions, operations will be faster, your system can handle more load, and you are less likely to be resource constrained.

With that in mind, another model can help us reason about system load at latency—Little's Law, which we'll look at next.

Little's Law

Little's Law is another mathematical theorem that comes from the field of queuing theory. Before the mention of a third mathematical theorem in a single chapter has you running for the hills, let me assure you that it's

actually a surprisingly straightforward concept, and one that can be very useful in a number of areas. The underlying mathematics are simple enough that even I can understand it.

Little's Law is described in its algebraic form like so:

$$L = \lambda W$$

This is otherwise translated as:

$$\text{Items in the system } (L) = \text{Arrival rate } (\lambda) \times \text{Processing time } (W)$$

So the number of items in the system will be driven by how many items are arriving and how long it takes those items to be processed. On the face of it, this seems so simple as to lack value. If you process items quickly (W is low), you'd get rid of items in the system quickly, so the number of items in the system would reduce. Likewise, if your processing time increases but the rate at which items arrive (λ) stays steady, you'd expect the number of items in the system to increase.

A (Contrived) Real-World Example

Before I show how we can use this theorem in the context of a distributed system, we can see how Little's Law works in the context of a sandwich shop serving customers who are lining up to order a sandwich, as you can see in Figure 12-4.[10]

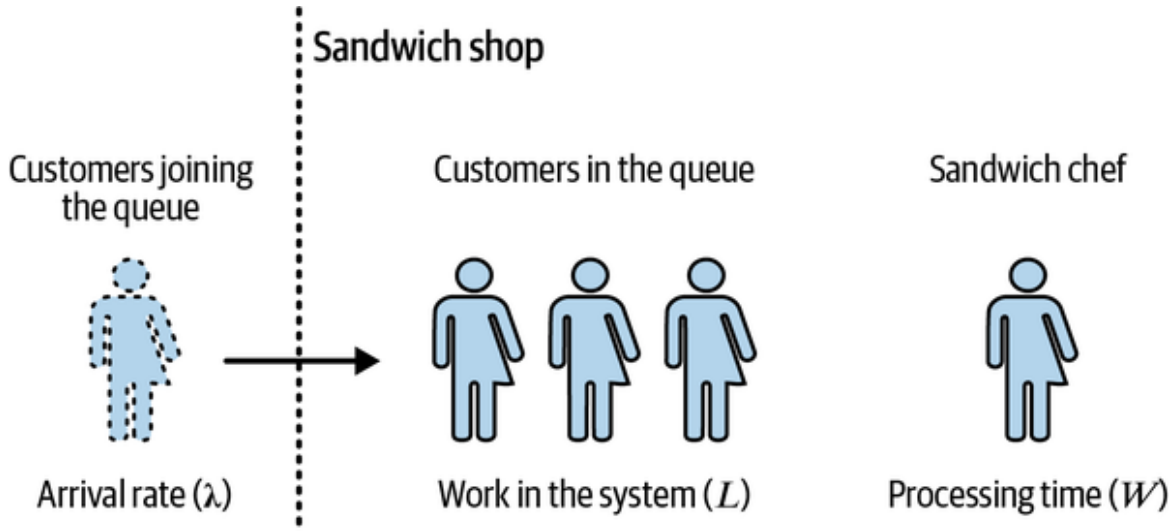


Figure 12-4. Little's Law applied to the lunchtime rush at a sandwich shop

At lunchtime, more customers arrive (λ increases). If the sandwich chef at the counter is processing the orders at the same rate (*W* is static), then the queue gets longer (*L* increases), as we can see in Figure 12-5.

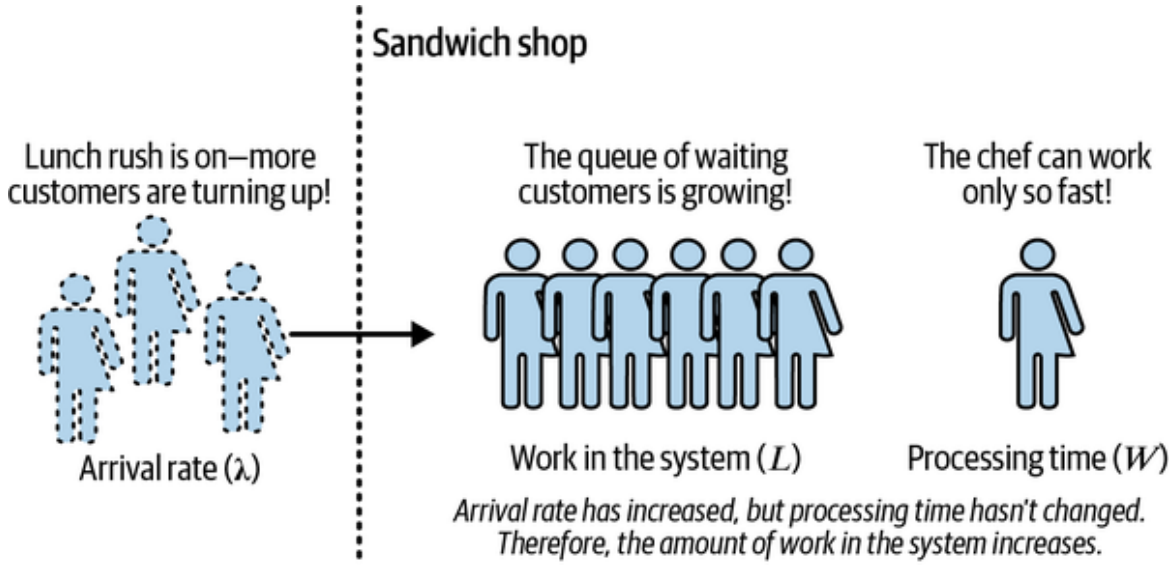


Figure 12-5. Adding a second shop assistant reduces the length of the queue

If you were the shop owner, you'd either need to find a way to reduce the amount of time it takes for a sandwich to be made or else accept that the queue length increases. If you could have a second sandwich chef come and help make sandwiches, then the rate at which requests are being handled would increase, reducing processing time (*W*). As you can see in Figure 12-6, this ends up reducing the length of the queue.

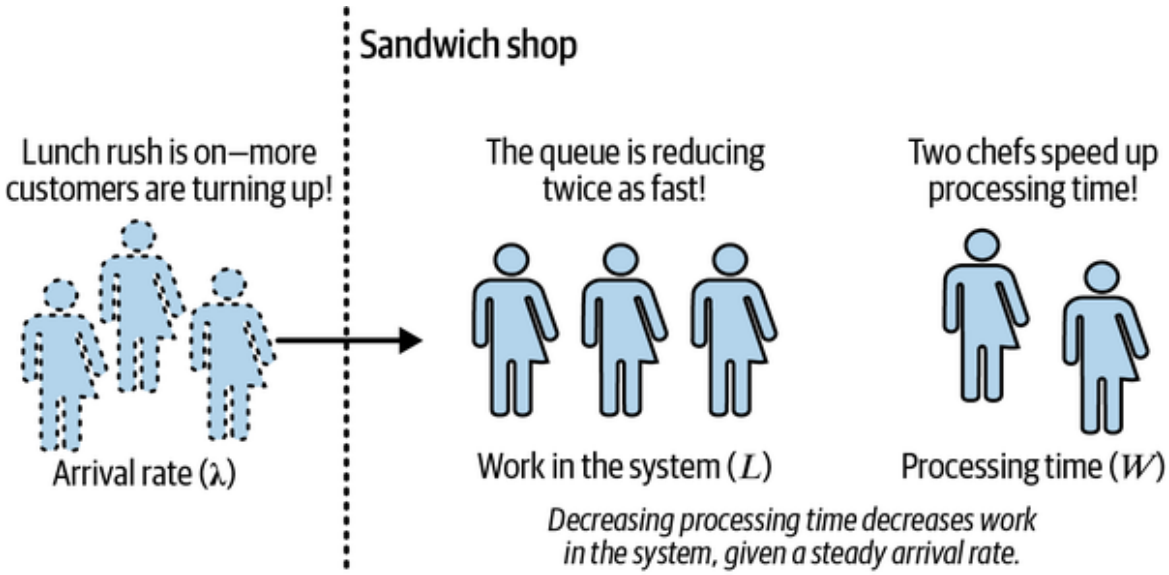


Figure 12-6. Adding a second shop assistant reduces the length of the queue

Little's Law and Ticketing Systems

Coming back to the example of the ticketing system, we know that in a zero partition state, we still need to pick consistency. Based on PACELC, that means latency will be higher, so according to Little's Law, our work processing rate (W) will likely increase too. Assuming a constant arrival rate of new work, that means we have more work in the system at any given point in time.

This could be a problem for our concert ticketing platform. What happens when tickets for a huge new tour going live causes a flood of traffic (a massive increase to λ), which in turn uses up all our computing resources, because each call takes longer to handle? Well, using Little's Law, what else could we do to keep the amount of work in the system down if we assume that our time taken to process work is fixed? We could change the arrival rate of work.

This is why ticketing systems commonly use some form of queuing process in front of the main ticket purchase flow, as shown in Figure 12-7. They need to keep a cap on how many active ticket purchase flows can be moving through the system at once to ensure that resources don't become exhausted. The customer-facing queue is a different subsystem around

which different trade-offs can be made, and as a result it can scale in very different ways.

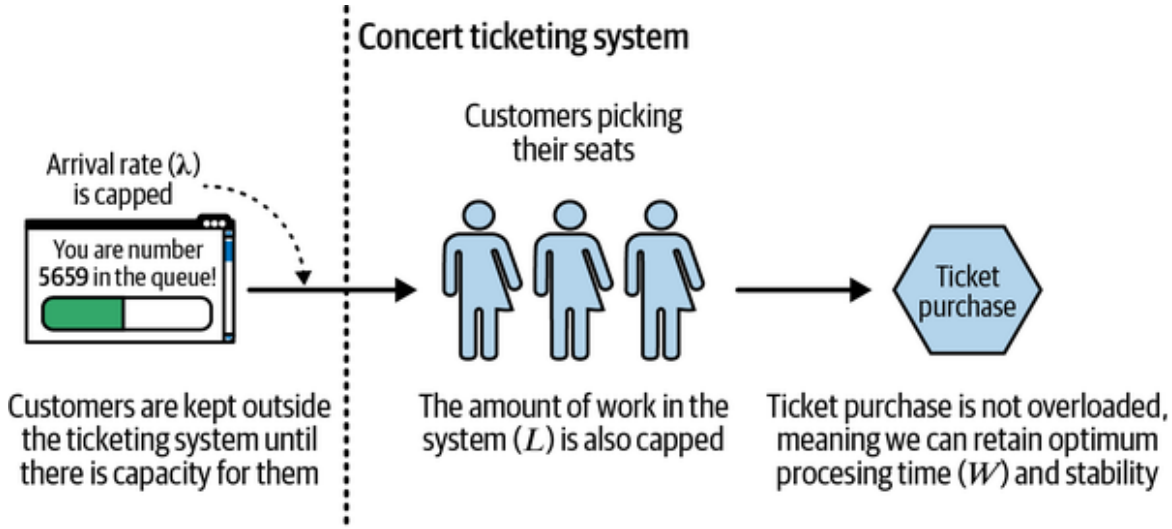


Figure 12-7. A user-facing queue can reduce the load for slower, strongly consistent operations

Queues like this are basically examples of backpressure (as discussed in Chapter 6) that happens to be visible to the end users. Part of the system (the ticket purchase flow) can handle only a portion of the overall demand. This limit on the amount of load that can be handled means incoming work is pushed back and held in the queue. Rather than leaving a page hanging or issuing an error page, the visual feedback helps customers know what is going on, and even in the queue page, you give them an estimate of how long they have to wait, assuming you know the customer's position in the queue and have an average time taken for a customer to leave the queue. For those of you who would like to dive more deeply into the design decisions made for ticketing systems, Anderson Parra and Vitor Pellegrino delivered a talk at QCon Plus 2022 that explored the architecture of SeatGeek.

Limitations of Little's Law

An important thing to note is that Little's Law states that the various parameters are independent of one another. But is that true? Let's come back to the sandwich shop example. As the queue starts getting longer, the experience of individuals waiting for their sandwiches would get worse.

Although the time taken to make a sandwich once a customer reaches the counter hasn't changed, that same customer will have waited longer to get to the counter in the first place. It's reasonable to expect that some customers might get fed up with waiting and leave. It's also reasonable to expect that some people might not get in the queue in the first place if it's very long—and this would reduce the arrival rate of work.

Similar things happen with our distributed systems. From Little's Law, we might expect that processing time shouldn't be impacted by the length of the queue, but we know that can happen. If the queue causes a resource constraint that also impacts the speed at which work can be done, then processing time can decrease too. I shared an example of this in Chapter 9, where I described how the throughput of some message brokers can degrade when lots of messages are in the queue.

So Little's Law is useful, but it's not foolproof. The more you can make the different parts of the equation independent from one another, the more likely it's going to hold up and be a useful tool to help you balance the scale at latency of your system.

Summary

When things go wrong, we still need as many things to go right as possible. With CAP theorem, we have a formal framework for understanding how we can degrade the functionality of our system when a partition occurs. But even when we don't have a partition, PACELC shows that we still have to trade off between consistency and latency.

Systems that favor consistency will typically have higher latencies (W), and Little's Law shows us that those systems will in turn have more work in the system (L). This helps us understand why highly consistent systems are hard to operate under large loads, and why mechanisms such as user-facing queues that reduce incoming load are so important.

Knowing which trade-offs are most appropriate in our distributed system requires an understanding of technical models like CAP and PACELC,

combined with a good understanding of the context in which our system will be used, including who the system is for. This requires both the technical and the social aspects of our system to come together. With that in mind, in Chapter 13 we turn our attention from the technical aspects of our system to the social aspects, as we explore the nature of sociotechnical systems.

---

1 Seth Gilbert and Nancy Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services", *ACM SIGACT News* 33, no. 2 (June 2002): 51–59.

2 Wojciech Golab, "Proving PACELC", *ACM SIGACT News* 49, no. 1 (2018): 73–81.

3 Shaler Stidham Jr., "Technical Note—A Last Word on $L = \lambda W$", *Operations Research* 22, no. 2 (1974): 417–21.

4 Armando Fox and Eric A. Brewer, "Harvest, Yield, and Scalable Tolerant Systems", in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HOTOS '99)* (IEEE Computer Society, 1999), 174.

5 To be clear, I know this is a *very* simplified view of how this actually works, so if you do work in ecommerce, try not to get cross about the details I've glossed over.

6 Losing inventory like this is commonly known as *slippage*; the term also covers other forms of accidental damage, stock misplacement, and even theft.

7 Guilty as charged. Yes, I'm a hipster.

8 Again, guilty on the age side of it—but I am very much not a qualified lawyer.

9 Daniel Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP Is Only Part of the Story", *IEEE Computer* 45, no. 2 (2012): 37–42.

10 Figures 12-4, 12-5, 12-6, and 12-7 use images from Don Clark Atlanta under CC BY-SA 4.0.

Part III. People, Process, and Culture

When considering the resiliency of a distributed system, we have to move beyond the technical and explore the behaviors, culture, and processes of the people building and maintaining the system itself.

Chapter 13. The Sociotechnical Systems

> *Software design is an exercise in human relationships.*
>
> —Kent Beck, *Tidy First?* (O'Reilly)

In 1979, a reactor at the Three Mile Island nuclear power plant near Harrisburg, Pennsylvania, experienced a partial meltdown. One of the factors contributing to the incident was that alarms coming from the monitoring systems were confusing, resulting in the human operators not taking action that could have prevented the issue from escalating.

In 1986, a failure in the solid rocket boosters of the *Challenger* space shuttle caused the craft to fail shortly after launch, killing all seven crew members. It was later determined that the warning signs had been there for all to see: previous launches of the shuttle had shown damage to the O-ring system, something that wasn't supposed to happen. Despite it being clear that no amount of damage to this component was acceptable, launches continued. Nothing bad had happened when this damage was seen before, so surely the launches could continue?

Humans were critical to these incidents playing out the way they did. In the case of Three Mile Island, the alarm systems left the operators confused and unable to carry out the right mitigating behavior. In the case of *Challenger*, what occurred became known as "the normalization of deviance."[1] Despite the system behaving in a way that deviated from the norm, because nothing bad happened initially, NASA engineers and managers were conditioned to ignore the preceding deviant incidents. The red light flashed and nothing exploded, so the light was ignored—until it became too late, leading to catastrophic results.

So, for a system to achieve the levels of resilience you want, you have to understand how to address human factors—not just to make working with

these systems easier and more pleasurable, but also to work out how to reduce the likelihood and impact of human error.

In fact, you shouldn't really think of humans as being separate from the system at all. When you think about our modern distributed system, the human operators and users are just as much a part of it as the computers are. In fact, seeing humans and technology as two parts of the same system is critical: they are both part of an interwoven *sociotechnical* system.

Understanding Sociotechnical Systems

Software created by and for a single individual is vanishingly rare. Even when an individual is lauded as being "the creator" of a piece of important software, the reality is often very different. Take the example of the Linux kernel, initially created by Linus Torvalds in 1991. While Torvalds wrote and released the first version, in less than a year contributions were being made from all over the world, and thousands of developers are now involved.

People build software both with and for other people. They operate within a set of processes (formal or informal) using software and hardware, and the work they do is directly impacted by the culture they work in. You can take any complex system and make the same argument; it's not just the software industry. This applies equally to manufacturing cars, extracting minerals, or constructing a building.

Fundamentally, our software systems have a lot in common with other complex systems that humanity has been constructing for decades, if not centuries.[2]

When looking at our distributed systems as sociotechnical systems, we have to accept that interplay between the technical aspects and the societal aspects exists. These aspects influence one another. So if you are trying to understand why a system failed, or you are looking to ensure that your system is made more resilient to avoid a future incident, it becomes incumbent on you to look at both the social and technical elements.

The Hexagonal Model for Sociotechnical Systems

An approach I've found useful when trying to grasp the realities of the sociotechnical system breaks things down into a hexagonal model, as shown in Figure 13-1.[3] While this model is designed to help us understand any generic sociotechnical system,[4] this book relates specifically to building and running a distributed system. Therefore, I have interpreted the model through that lens.

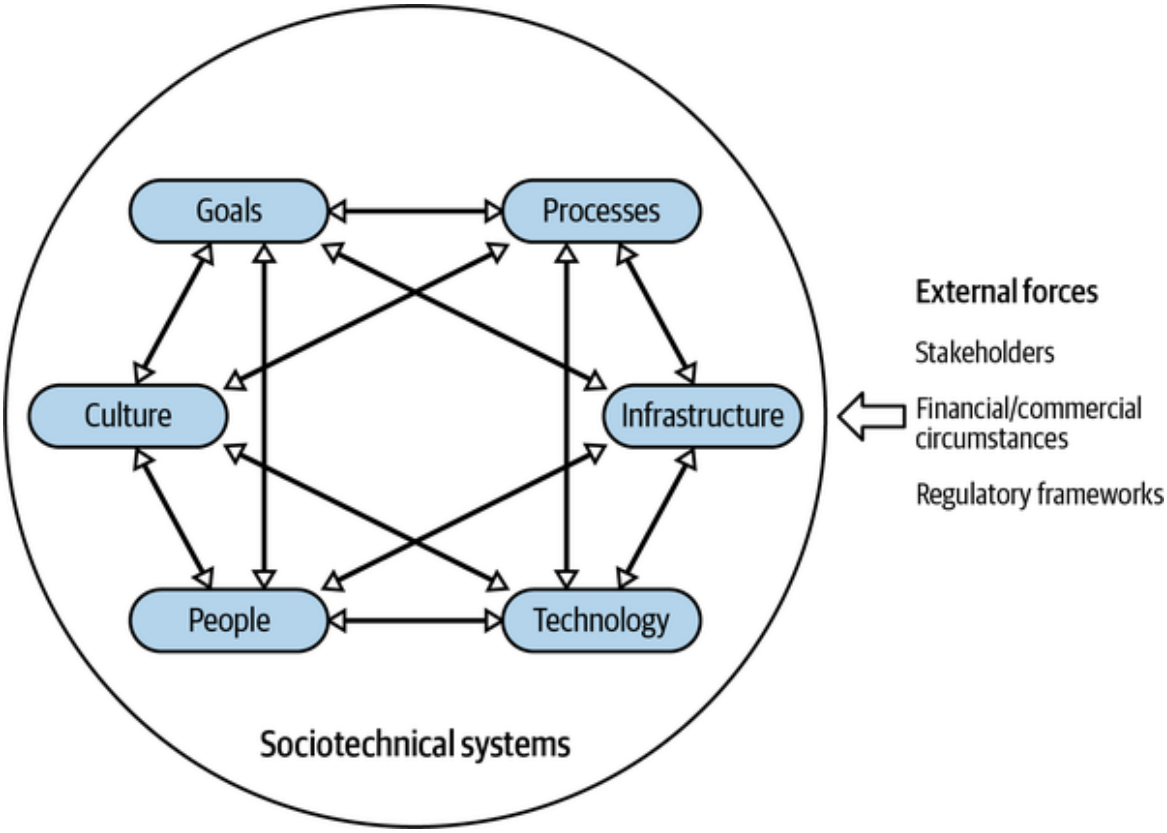


Figure 13-1. A hexagonal model for sociotechnical systems

The hexagonal model breaks down the idea of a sociotechnical system into six discrete concepts:

People

> The people doing the work and using the software (developers and users).

Culture

The social environment in which the work is done. For example, some cultures might be more risk adverse than others or might ignore the input from people in marginalized groups.

Goals

The incentives for the people creating, operating, and using the system.

Technology

The software being written and the supporting tools being used.

Infrastructure

Generically, this relates to the physical infrastructure at play. This could extend to buildings and roads, for example, but in the context of distributed systems, this concept will more likely cover computing hardware, networks, power generation, and so on.

Processes

These are the mechanisms put in place that guide how the work is done. This concept could include your preferred style of software development—for example, Scrum or Six Sigma.

People, culture, and goals are the societal aspects, whereas technology, infrastructure, and processes are on the technical side of things. These all sit within a wider ecosystem of influences that you will have limited to zero ability to control:

Stakeholders

This category would at a minimum encompass the people the system is being built for, but it could include a wider range of interested parties.

Financial/economic circumstances

Your company could be flush from a cash injection from a new investor, or it could be operating against the backdrop of a national recession or

regional conflict.

Regulatory frameworks

Rules and regulations that might apply to the industry you work in. This might mean adhering to the Payment Card Industry standards when handling credit cards, for example, or complying with regulations such as the General Data Protection Regulation (GDPR) for companies operating in the EU.

A change to any one of these elements will have a knock-on impact elsewhere. For example, a culture in which people are empowered to make local decisions might result in a proliferation of tools being used (technology). Targets that aim for low levels of bugs may result in the people building the software not wanting to report issues when they arise (goals).

Let's look at each of these sociotechnical factors in more detail so that we can better understand how they might improve or impair our system's resilience.

OTHER METHODS FOR UNDERSTANDING SOCIOTECHNICAL SYSTEMS

While the hexagonal model I've presented here helps us reason about the constituent parts of a sociotechnical system, other concepts exist to help us analyze and understand them. The two most well-known concepts appear to be the Functional Resonance Analysis Method (FRAM), developed by Erik Hollnagel,[5] and Systems-Theoretic Accident Model and Processes (STAMP), developed by Nancy Leveson.[6]

Hollnagel and Leveson are both experts in the field of safety-critical systems. Their models give us different ways to interpret the reality of our sociotechnical systems and give us different tools to help make sense of them. Both models are widely used in incident management to help understand the broader factors that may have contributed to the outcomes. I am not enough of an expert to speak to the merits of one model over the other, but I wanted to make you aware of them in case you wanted to explore this topic in more detail.

Of course, other theories look at the interplay between organization and technology. Conway's Law is a well-known concept whereby the software architecture of a system is said to follow the structure of the organization that builds it. While in my experience I've found Conway's Law to often be true, at best it speaks to only a small part of the interactions between the social and technical parts of the systems we build.

People

No part of our distributed system is as important as the people who build, operate, and use the system. The other aspects of the sociotechnical system we will look at greatly impact how the people in our system are able to perform. Having people in our system who are able to learn, think critically,

communicate with one another, and contribute what they discover to the wider organization is essential.

You can of course influence the people in your organization through recruitment, the rotation of people through different parts of your company, and, in extreme circumstances, moving people on. But fundamentally, you'll get the best out of your people by creating an environment in which they can thrive and feel able to be their best selves.

Culture

If your culture tends toward the bureaucratic, where people tend to stick to specific roles, this can make it difficult for you to react to unfamiliar situations. Rigid structures and hierarchies tend to guide people to very specific behaviors in a corporate structure. Rigid hierarchies tend to be shaped by what has happened before, not by being ready for the unknown. This tendency can materially impact your ability to gracefully extend your system in the face of surprise.

A culture that punishes mistakes is going to be one in which people won't admit mistakes. This will result in issues being covered up, right up until the moment they cause a big enough problem that they can't be ignored. On the other hand, if your corporate culture is more open and honest about issues, then you are more likely to find and fix things before they become problematic. This is why it is generally accepted that cultures that promote psychological safety tend to operate better than those that don't. I'll explore the topic of psychological safety in more detail in Chapter 15.

One manifestation of a positive culture comes from Etsy, which wanted to encourage a culture of learning and sharing. Every year the company awarded an actual three-armed sweater to the engineers who had been involved in causing the biggest incidents—or had, in Etsy's words, turned "mishaps into opportunities for innovation and growth." I'm not sure if the award is still going, but at least up until 2023, the award winners were announced publicly. From speaking to at least one previous Etsy employee,

I'd say that the people there considered the award not only a badge of honor but something that really reinforced the notion of sharing mistakes.

Goals

The phrase "you get what you measure" is used frequently, often in hindsight when goal setting has led to an undesirable outcome, sometimes even the opposite outcome from what was desired. One of the most famous (and potentially apocryphal) stories of this nature includes a bounty established in India during the days of the British Raj that was intended to reduce the population of cobras. People soon realized that breeding cobras and then turning their bodies in to the authorities to collect the bounty was a surefire way to make money. When it was discovered that people were breeding cobras rather than catching wild ones, the bounty was abolished, at which point the cobra breeders released their snakes, resulting in an increase of cobra numbers overall.[7]

Setting goals has, by definition, a massive trickle-down impact. We've already seen across the industry that a rapid push to adopt AI, often driven by senior leadership, has ended up increasing instability in our systems. I've also been involved in multiple projects where releases were pushed out the door before they were ready in order to meet some business goals—and in more than one case this resulted in significant fallout.

On the other hand, goals can also help to improve the resiliency of a system. A drive to reduce change failure rate, supported by leadership, could give teams the air cover they need to improve both their processes and their technology.

Technology and Infrastructure

I've combined technology and infrastructure together since we have covered these two issues throughout the many chapters leading up to this point, and thus there is no need to dig into them further here. But so far you've seen how small things like getting your timeouts right can help avoid a system collapsing, how DR ensures that you can keep your system

running when the worst happens, how you can mitigate progressive collapse, how you can make retrying safe, and so much more. So in this chapter, we'll focus on the other aspects of the sociotechnical system.

Processes

Some formal processes can codify roles to the extent that there exists little opportunity for creativity or thinking outside the box. In this way, the culture and goals of an organization often end up driving the process (and vice versa).

A lot of existing processes in organizations are like scar tissue. When things went wrong in the past, processes were changed after the fact to ensure that the issue wouldn't happen again. The problem is that this scar tissue can exist long after the original injury, even if the context has changed so much that the mitigation no longer makes sense. The most extreme example I've seen of this was an organization that, because of a previously botched performance test that was accidentally run on production, had banned performance tests entirely!

Some processes can structurally inhibit our attempts to build more resilient systems. Delivery processes that involve lots of handoffs can result in knowledge getting lost, while also extending the time taken to get to production. Another example I've seen of this was a company that still required creating a manual ticket in Jira for any change to the infrastructure, even though the infrastructure had long since been moved over to AWS. The self-service nature of AWS being completely hidden from the teams who wanted to make infrastructure changes added lots of delay.

External Factors

Whereas the factors that we've looked at so far are at least theoretically under your control, there are a number of influences on your sociotechnical system that you just have to deal with. Regional conflicts can make data centers unavailable, as has happened during the recent conflict in Iran. The growth of AI has led to equipment shortages and price hikes, meaning that

you might not be able to get the hardware you need, and the people you might want to hire may have their heads turned by fancy startups.

New regulations are unlikely to catch you unaware, but they can have a significant impact on what you need to do and on how you need to do it. For example, the EU's recent Digital Operational Resilience Act (DORA) requires organizations to have a framework for managing IT risks around resilience. I am far from an expert, but my own read on DORA is that the requirements are not particularly onerous and arguably don't go far enough in some regards. However, even if your organization does most things sensibly, making sure that what you do maps to what DORA prescribes is going to involve a lot of work. It can also cause pressure in other ways, as the stakes get raised. An internal issue leading to the regulator fining you or perhaps calling you out publicly is going to add some extra spice to your internal procedures.

In some parts of wider society, we are seeing a backlash against AI more broadly and more specifically around the construction of data centers in different parts of the world, due to concerns around pressure on local water and power supplies and around environmental pollution, among other issues. This could well have a knock-on effect in terms of the plans you might have for expanding your use of AI.

The Therac-25 Radiation Machine

We've taken a look at a number of sociotechnical factors, and we will explore some more real-world examples of how these ideas play out in the remaining chapters. Before that, however, I want to use our newfound sociotechnical lens to look at a case where a medical device ended up killing some of the patients it was supposed to help.

The Therac-25 radiation therapy machine was a medical device that was created to help treat people with cancer through the use of X-rays.[8] Between 1985 and 1987, massive radiation overdoses were delivered to at least six patients, resulting in the deaths of at least three people and life-

changing impacts for the others. The Therac-25 was based on earlier devices that had a series of physical safeguards to ensure that the machine could not be set up in invalid configurations that could lead to harmful consequences. The newer Therac-25 device, however, moved away from these physical controls and was instead controlled entirely by a computer. The main benefit of this approach was that the time taken to set up the device could be drastically reduced.

The software for the Therac-25 was based on the software for the earlier Therac-6 and Therac-20 devices, but unfortunately it had not been updated to account for the removal of the physical safeguards. This meant that the software allowed for the device to be used in an unsafe manner. In the wake of individual incidents, patients alerted the operators of the machine, but their concerns were repeatedly dismissed until the resulting symptoms made their claims impossible to ignore.

This isn't the place for a full-blown study of what happened in this situation (for that, look at the work done by Nancy Leveson et al.[9]), but a host of issues were discovered as a result of various investigations and lawsuits resulting from these incidents. We can take some of the many observations made and map them back to the model I shared earlier, showing how a confluence of issues likely led to the incidents:

Software

> The machine lacked safeguards to prevent it from being operated unsafely. Poor error messages made it difficult to diagnose the issue when a problem occurred.

Infrastructure

> Physical safeguards were removed from the previous design; no ability for the software to verify that the machine's sensors were working properly.

Culture

Patients were ignored by medical professionals when reporting their concerns.

Goals

Reduction in the time taken to set up the machine for each patient.

People

The software was largely reused and written by a single engineer. Minor issues occurred so frequently that operators learned to mostly ignore error messages.

Processes

The new software and hardware were not tested until the machines were assembled in hospitals.

Seeing the interconnected nature of these issues again undermines the concept of a single root cause of failure, a topic we'll return to in Chapter 14.

Summary

Hopefully, I've made the case by now that it's impossible to build a resilient system without thinking more broadly about the nature of the system. The infrastructure and technology we use is vital, but so too are the people who build and operate the system, the culture they operate in, and the goals they set. This theme will continue in Chapter 14 when we look at how to handle incidents, both in the moment and after the dust has settled.

---

1 Diane Vaughan, *The Challenger Launch Decision: Risky Technology, Culture, and Deviance at NASA* (University of Chicago Press, 1996).

2 And arguably for much longer than that—you can absolutely see the construction of the pyramids as a sociotechnical system.

3 Readers of my previous books will know that I *love* a hexagon.

4 Matthew C. Davis et al., "Advancing Socio-Technical Systems Thinking: A Call for Bravery", *Applied Ergonomics* 45, no. 2 (2014): 171–80.

5 Erik Hollnagel, *FRAM: The Functional Resonance Analysis Method—Modelling Complex Socio-Technical Systems*. London: CRC Press, 2012.

6 Nancy Leveson, "A New Accident Model for Engineering Safer Systems", *Safety Science* 42, no. 4 (2004): 237–70.

7 This frequently told story doesn't actually have much evidence to back it up, but similar examples did happen around pest control initiatives in Vietnam in 1902, and in the US in 2008, relating to rats and wild pigs, respectively.

8 Adam Fabio, "Killed by a Machine: The Therac-25", *Hackaday*, October 26, 2015.

9 Nancy G. Leveson, *Safeware: System Safety and Computers* (Addison-Wesley, 1995).

Chapter 14. Incident Management and Post-Incident Response

> *You may encounter many defeats, but you must not be defeated. In fact, it may be necessary to encounter the defeats, so you can know who you are, what you can rise from, how you can still come out of it.*
>
> —Maya Angelou

Incident response has two clear goals. The first is to get the system up and running as quickly as possible. The second is to capture enough information to identify ways in which the system (in a sociotechnical sense) can learn and adapt.

Broadly speaking, this chapter is split into two parts. First, I'll look at the process of handling an incident and its immediate aftermath. Second, I'll look at the process of post-incident review, and at how to make sure you create the best opportunities to learn from what happens.

I also want to be clear that this chapter is not intended to be an exhaustive discussion of incident management. Where applicable, I've added references to other resources from which you can learn more. My main goal in this chapter is to help you see incident management as an extension of the ideas of resilience engineering as a whole, a process in which dealing with surprise and continual learning is key to building more resilient systems. I'll also show how incident management doesn't follow a simple, structured workflow—like our distributed systems, it can be a messy, multifaceted affair.

Before that, though, we should probably define what an incident is. Oddly enough, this is not as clear-cut as you might think.

What Do You Mean, “Incident”?

Some incidents are obvious, while others are not. Something doesn’t look right; is it an issue? Poke around—is it getting worse? Did it go away? Maybe that quick fix worked. Some so-called incidents are just business-as-usual activities that never go any further. Other incidents become major issues.

There isn’t a universal set of rules defining what an incident is. What is important is that you decide for yourself what constitutes an incident, based on your context. Have a discussion with your team, and try to understand existing requirements within the organization. For example, you might have a regulatory requirement to deal with certain issues promptly, which would require raising them as incidents. Or perhaps issues that impact certain customer groups need to be flagged with a higher severity, as those customers are more valuable.

Specifying which SLOs being breached constitutes an incident is sensible. Relatedly, you may decide to raise an incident based on the extent to which your users are impacted. If, for example, a small number of users are seeing a slowdown in how long it takes to render a page, that isn’t great—but when a large number of users see the same problem, you may need a more formal incident. It’s not just about how many people are impacted, though; even a small number of users being unable to access their bank accounts might require a formal incident due to the significant impact this would have on those users.

It’s also worth noting that you shouldn’t limit yourself to declaring an incident only when something catastrophic has occurred. The more incidents your team works on, the more experienced it becomes, and the more opportunity for you to adapt the guidance and tooling your colleagues rely on in incident management. Of course, you can practice incident management during simulated exercises like game days—and this is a topic I’ll discuss more in Chapter 15.

Whatever guidelines you put in place for an incident (and this will be a recurring theme throughout this chapter), make sure there is room for

discretion. Experienced system operators can often have intuition about things "not feeling right," so giving them the space to explore what is happening is very important.

Also remember that the absence of a formal incident doesn't mean that incidents aren't happening. It might mean you just haven't spotted one yet!

Not Linear

You have probably seen variations of the incident management workflow presented in Figure 14-1. While each of the activities shown in the workflow is likely a thing you will do as part of incident response, this diagram is lying to you. Incident management is not linear, nor is it fixed. It is a process of exploration, discovery, and circling around various activities in order to find sensible resolutions.



Figure 14-1. A linear, structured approach to handling incidents

When you're handling an incident, it doesn't feel neat—it's messy. Trying to impose a linear or highly structured process on incident management isn't about making the messiness go away; it's about pretending it doesn't exist. Halfway through incident response, you might discover new information that makes you reexamine previous steps. New people might join the incident-handling process and bring new information, or the scope of the incident might expand, requiring more diagnosis.

The problem with the lie of this linear flow is really a more general one with traditional incident management in IT. Often the process of incident management is highly structured, with checklists that have to be followed. There is little to no leeway to do anything outside this process (often because the process is itself quite heavyweight). But as we've learned, being resilient is about knowing how to deal with surprise—it's about being

able to embrace the concept of *graceful extensibility*, to use David Woods's term that I introduced in Chapter 1. As John Allspaw puts it:

> *Incidents come in all shapes, sizes, volumes, and colors. They have multiple tangled dilemmas, illusions you're certain were absolutely real, and needles in haystacks wrapped up tight into red herrings you can only understand in hindsight.*[1]

A more accurate pictorial description of how incident management works can be found in Figure 14-2. This representation is based on research into NASA's fault management process[2] and does a better job of recognizing the messy nature of incident management.

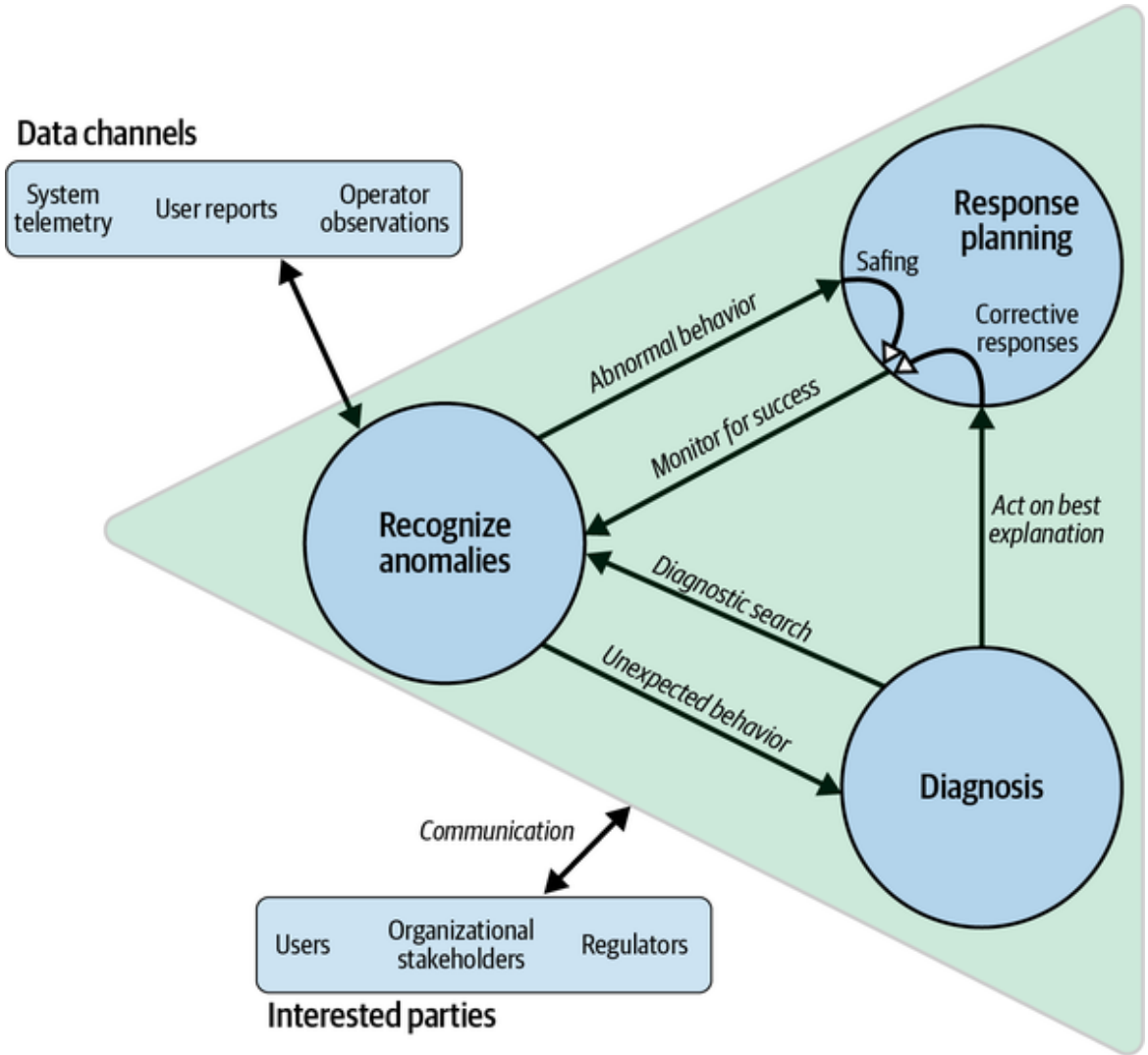


Figure 14-2. A more accurate representation of incident response

This diagram attempts to articulate the lack of clear distinction between the activities outlined here. It is, in essence, an attempt to portray the things that responders do as they work toward a resolution. The diagram is at best an approximation of the cognitive activities that responders may undertake; it should not be assumed to be a perfect summation of how every incident will end up being handled.

To explain the process outlined in Figure 14-2, we could turn to the example of a simple incident caused by a breached SLO, where the problem is well understood. The SLO breach results in an alert. This abnormality is well-known by the operator, who then moves straight into response planning, as we see in Figure 14-3. The issue is one we have seen before and is likely something we know how to deal with. In this image, the "safing" step refers to the immediate and often automated actions we undertake to try to stabilize a system in the event of an anomaly. This represents the remediations we carry out to bring the system back into a more reliable mode of operation. After safing, we will need to monitor the system to ensure our actions have been successful in stabilizing the system.

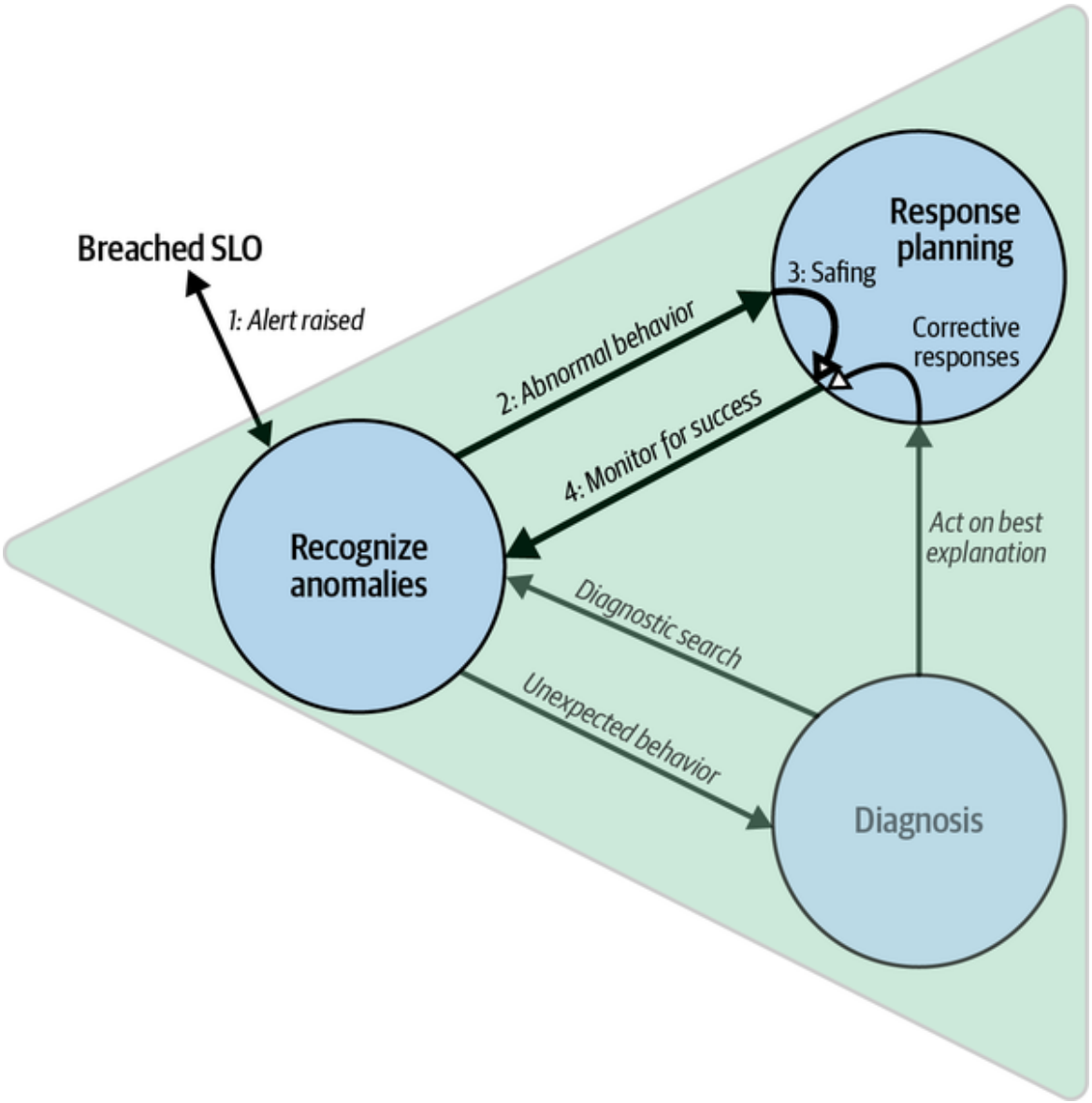


Figure 14-3. An anomaly that is quickly identified

If the anomaly is not expected, a period of diagnosis is required to work out what might be going on, as Figure 14-4 shows. We might not have experienced a Godzilla attack before. Perhaps we should search for historical records? Did anything work before? This may well lead to the anomaly being better identified, in which case we can move through normal safing activities. On the other hand, we might have to guess at a solution and see what happens—perhaps Mothra can help? This would then require a period of assessing the impact of the fix and deciding what to do next if

that fix hasn't worked (or has made things worse—although Mothra is pretty relaxed, as far as I can tell).

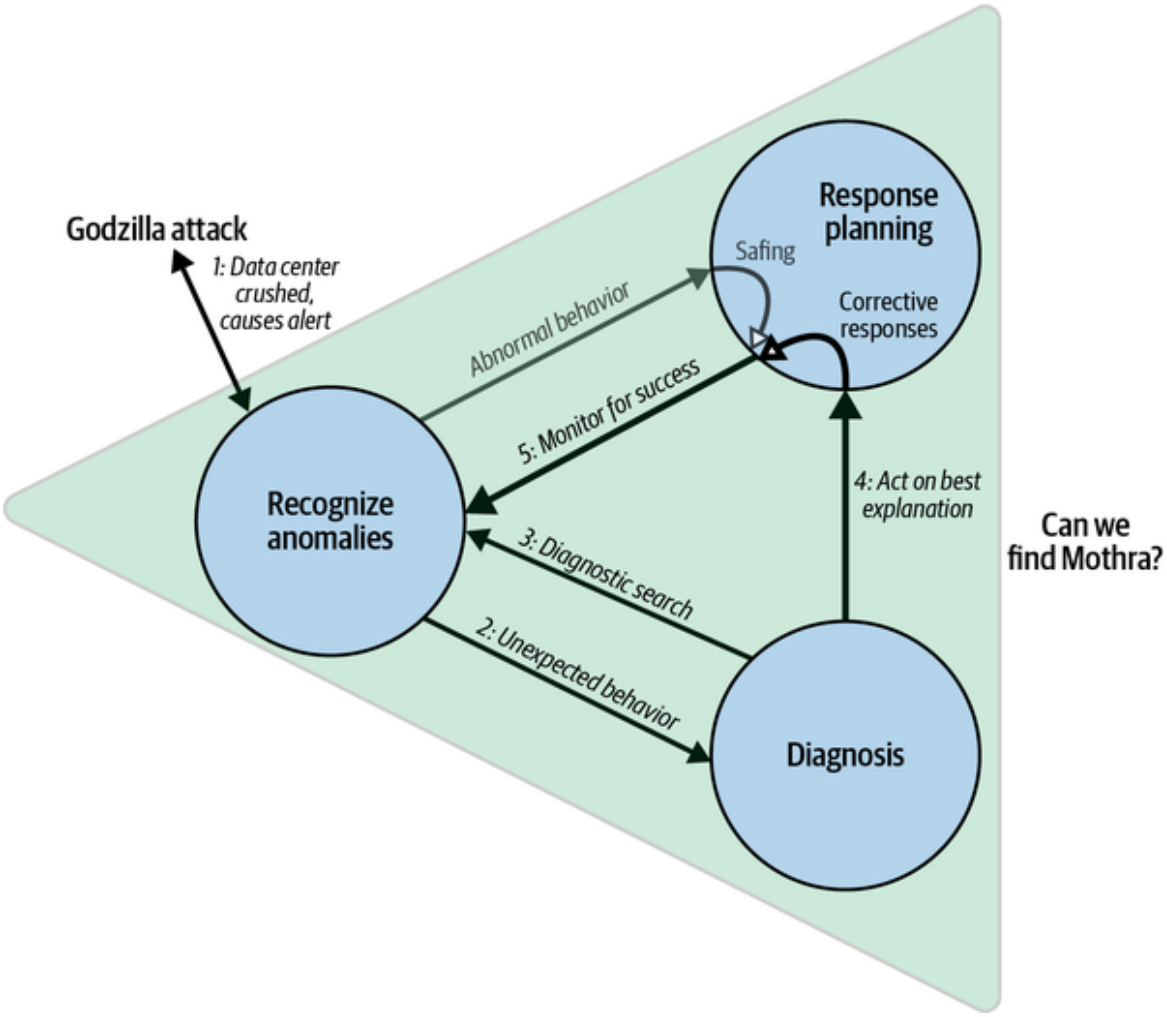


Figure 14-4. An anomaly that represents unexpected behavior

It's worth noting that this diagram doesn't indicate all the activities that are undertaken as part of incident management as a whole; it's more a general model to help show how the process of incident response is often iterative, involving lots of exploration and even experimentation.

Making Incident Response Easier

In advance of an incident, you can take concrete steps to make effective incident response that much easier. Let's touch on these next.

Right People, Right Place

On July 19, 1989, United Airlines Flight 232 suffered the explosive disintegration of one of its engines. This explosion severed all three hydraulic lines in the DC-10, essential for control of the aircraft. This was considered an incident with a near-zero chance of probability, and no process existed for landing an aircraft in the wake of an incident like this. As a result of the pilots' loss of control, the plane started to bank to the right, with the nose dropping.

Once the pilots realized that they no longer had control of the flaps and rudders, they enlisted the help of another pilot who was on the flight as a passenger. Working together, the flight's captain, Alfred C. Haynes, first officer William R. Records, engineer Dudley J. Dvorak, and passenger (and experienced pilot) Dennis E. Fitch were able to identify that by changing the throttle on the two remaining engines, they could steer the aircraft.

The plane made a crash landing at Sioux Gateway Airport. Of the 296 people on board, 112 died. Despite the tragic circumstances, the outcome could have been so much worse. The skills of the flight crew were key to ensuring that so many people survived.

Having the right people with the right skills and temperament at the right place at the right time might sometimes look like luck, but it isn't. The crew of Flight 232 were highly trained and highly experienced.

Having people with the right skills to handle incidents is key. As Allspaw put it in his talk at SREcon24 Americas:

> *The best scenario when it comes to responding to an incident is:*
>
> 1. *The people responding can recognize immediately what is happening, and*
> 2. *know exactly what to do about it.*[3]

Making sure that the organization arms incident responders with the skills, tooling, and support they need to respond effectively is vital. Unfortunately, this step often doesn't get the attention it deserves. Effective incident

response is often not rewarded, especially when it results in incidents never being noticed. It can be much easier to get funding to go from 99.99% to 99.99**9**% availability for a service than it is to improve the capabilities of incident responders, but the former is likely to be much less impactful than the latter.

Some of the activities I'll touch on in Chapter 15 will give you pointers on how to foster an environment of continual learning. But the incident itself becomes a great lens through which to assess the efficacy of your incident response. During the post-incident response, make sure to reflect on the experiences of responders during the incident, and try to identify what actions can be made to help further support responders. Simulated incidents can also be incredibly useful as a form of deliberate practice to help responders grow skills.

Clear Lines of Ownership

> *When ownership is unclear, everyone responds and nobody decides.*
>
> —Dileep Kumar Pandiya, principal software engineer at ZoomInfo

With a distributed system that has multiple components, knowing who owns what is very important. During an incident, the responders may need to pull in people with expertise on the impacted components. If it's unclear who should be contacted, then that may undermine your ability to get the right people working on an incident.

Obviously, in an interconnected distributed system, understanding who should be called in is difficult. If a failing user flow cuts across 15 services, do you pull in the owners of all 15? In practice, the first responder will do their best to identify where they think the issue is and pull the right people in, but of course as an investigation continues, the number of people they need to reach out to will grow.

Having a good, up-to-date service catalog can definitely help the first responder work out who to pull in. A registry that maps running services to information about them can be incredibly useful. Better still, if it's a short

step from the catalog to being able to contact the on-call person for the service in question, then incident response can be further sped up.

However, if some of these services have confused ownership, then the scope of people who might get pulled into an incident will increase massively—further increasing the impact of coordination during incident response. Having more people involved in incident response is often highly detrimental. We'll look at the cost of coordinating responders in a bit more detail shortly.

Make Systems Easily Observable

As we can see in Figure 14-2, having good telemetry from your system is going to be key. The more observable your system is, the easier it will be for you to spot anomalies. But the importance of observability doesn't stop there. As you carry out remediations for the incident, you need to be able to observe whether the fixes have worked. And you also need to be able to use both current and past telemetry to try and spot correlations, search for previous similar incidents, and share data with fellow responders.

Having business metrics available in your production telemetry is also key. Incidents often manifest themselves in terms of end-user impact. Being able to see how telemetry that is meaningful to the user experience is being impacted and correlating this with other operational data can make diagnosis and remediation much easier.

This requires that your system is emitting useful signals, a subject we explored in more detail back in Chapter 2. But it also requires that responders know how to use the tooling in the first place. You really don't want to have to work out how to use your observability tooling during an incident.

Exposing potential responders to your observability tools as often as possible is vital. Using the same tooling for development, test, and production purposes can greatly help. Running simulated exercises and shadowing experts are also great ways to get some experience in using the tools.

Remember also that incidents by their nature are somewhat unexpected. Given that, it's not always possible to predict in advance what questions you might want to ask of your production system. So adopting tooling that supports flexible querying, supported by high-cardinality data, is going to set you up well.

Incident handling is also normally a collaborative process. So make sure your tooling supports easy sharing of data and, ideally, collaboration within the tool itself. Collaboration not just with other fellow responders but also with AI assistants is useful. Honeycomb's Canvas feature, for example, is a shareable, collaborative workspace in which humans and AI can work together to explore the telemetry as part of incident response; although Honeycomb seems to be the first to offer something like this, I'm sure it won't be the last.

Implement Fast Rollback

The key initial goal of incident response is to get the system back to where it needs to be for its users. In some situations, this may mean fixing the problem by deploying a code change or making a configuration change. Perhaps it might involve tweaking the infrastructure—for example, provisioning more capacity. But sometimes the best approach may be to just roll back.

If an incident occurs shortly after a change, the change itself becomes a potential culprit, and the nature of that change is going to undergo a lot of scrutiny as part of the incident response. Sometimes the safest thing to do is just roll back the change to see if the incident can be resolved, at least in the short term.

For this approach to work, we need the rollback to be well tested, trusted, and ideally fully automated. A complex and potentially error-prone rollback mechanism is just as likely to make the incident worse.

If the rollback resolves the initial problem, you can continue your investigation in a much less stressful environment. You can use the data gathered during the incident to try and work out what caused the problem

before you try deploying the change again. Having access to a fast, well-trusted rollback mechanism is a very useful part of the incident response toolbox.

Incident Roles

When an incident is raised, you need someone who is responsible for the incident handling. This role is normally called an *incident commander*, although you may use other terms inside your own organization. The person in this role may have the help of an assistant incident commander, a role that may be required for sizable incidents.

The incident commander will work with a team of responders who are involved in dealing with the incident. These will typically be on-call people who are owners of different parts of the system. Additionally, you can expect other interested parties to be part of incident response, such as internal stakeholders, who will want to know what is going on; subject matter experts; representatives from third-party vendors; your customer support team, to ensure it can liaise with customers; internal security and compliance folks; communications; and, if things get really bad, perhaps even your legal team and external regulators.

It's worth noting that these roles may not be fixed. As incidents become broader, more people might need to be called in, and in some situations it might be appropriate for the incident commander to hand over their role to someone else. We'll explore that in more detail shortly.

Key Activities in Incident Response

So incident management isn't linear, and having a fixed process for how incidents are handled can be problematic. That doesn't mean you can't prepare and have well-understood processes for how certain activities are done already in place.

Courtney Nash, CEO and founder of the VOID, a database of public incident reports, has written a report on the importance of having a prepared incident response plan. In the report, she details five key aspects of effective incident response that you can think about and plan for in advance:

Communication

: Knowing who will be kept informed about the incident, and how

Delegation

: Handing off work to other responders and potentially to third parties

Escalation

: Involving more senior colleagues based on the severity of the incident

Coordination

: Organizing the people involved in incident response

Saturation

: Dealing with being overwhelmed

Let's explore each aspect in more detail.

Communication

Keeping people in the know when an incident is ongoing can be vitally important. Aside from any regulatory requirements you might have, the communication around an incident has the potential to either reduce or magnify the impact of an incident.

A well-informed user may be able to perform workaround actions if given enough information. They will also appreciate being given a timeline if one is available. Business stakeholders who are concerned about the impact of the incident might start getting involved in unhelpful ways if they aren't kept in the loop.

How you want to communicate, and with whom, is something that should absolutely be decided beforehand. During an incident, stress levels are high, as there's a lot going on. At that moment, developing an effective communication plan on the fly will be difficult!

Thus it's important to decide on a few factors ahead of time:

Format

Email, SMS, phone call, website update, etc.

Frequency

Some interested parties in a response may want a constant stream of information, while others might prefer less frequent, more summarized information.

From who

Who will be expected to send out the communication? Can it be automated?

Audience

Is this for internal or external stakeholders? What kind of language should be used for different types of audiences?

It's very likely that you'll have multiple communication channels. At the bare minimum, you will likely have one channel for communication with customers and a separate one for internal communication. For some incidents, you may have a channel that initially excludes C-level execs, as informing them too early is often not desirable—especially if you can get everything resolved quickly!

Delegation

With more than one person working on an incident, deciding who plays what role is key. This is especially important for long-running incidents,

where roles may need to be handed over.

Delegation can include bringing in subject matter experts—people who know specific subsystems, for example, or support for vendors. In a situation where regulatory issues need to be dealt with, you may have an appropriate point person who needs to be involved in the incident. Having some guidance as to what requires pulling in extra help, especially from external parties, may be key. For example, you might have a rule that states that any suspicion of a data breach requires pulling in security and legal.

Escalation

When the scope of an incident broadens, it may be appropriate to escalate the incident response to someone with more seniority. This step may be required to bring more resources to bear in order to help remove blockers.

Escalation can be a risk! It changes the dynamic of the incident, as new personnel will become involved. You're likely escalating at a time when your stress levels are already high. So being clear ahead of time as to what situations will cause escalation is important.

Coordination

As the number of parties involved in an incident grows, coordination becomes important. This responsibility would normally fall to the incident commander. As the scope of the incident grows and more parties become involved, the effort around coordination can become significant, in which case having an assistant incident commander dedicated to this task can be appropriate.

Saturation

Handling an incident can be stressful. Responders can quickly become overburdened by the often chaotic nature of incident response as well as the pressure felt from the impact to the system.

One thing that can be done to delay saturation is to give people breaks. Although this might be difficult to consider during an incident, finding times for people to take at least a short break can be useful to help them decompress. This is especially important during a long incident. As an incident commander, you should make sure this is happening not just for your team but also for yourself. Responders are likely to take their cue from you, so you should model the behavior you want to see by taking short breaks and letting people know you are doing this. A simple "Going to take a 10 min break to grab some food and clear my head" statement on Slack can go a long way toward making sure other people follow suit.

Once responders become saturated, you have several options. The first is to *do fewer things*. Just as our systems can load shed when overwhelmed, humans can do the same. If there is too much to be done, making a conscious effort *not* to do some of it may well be the smart play.

Another option is to *be less thorough*. In a similar way to load shedding, rejecting work can be the right call. Making a conscious decision to be less detailed around an activity can give you the cognitive bandwidth to handle more important things. This is why strict checklists can be problematic; they can end up making you think that everything needs to be done, and in a particular order. If you are going to prepare checklists, make it clear that they are only advisory in nature, and flag those aspects that can be considered optional.

You can also *ask for help*. Bringing in more people to help with an incident is a great way to make sure your responders aren't overly saturated. This could also involve escalating the incident to access more support, though as we've touched on, that isn't always a trivial activity. Of course, you have to be aware that adding more people can increase the coordination work!

You may also realize that you are not best placed to handle the role you are playing, in which case you may need to *hand off* the incident to someone else. This can be tricky, as some might feel that this represents a failure. Having some guidelines in place around when handing off the role may make sense can remove the emotion of the moment and take the sting out of

decision making. For example, if you've been incident commander for more than six hours, or if the severity of the incident has increased, this might be a trigger point for at least a discussion around whether changing the commander is appropriate. Having a fresh incident commander may well be the best way to help the team find a solution for a long-running incident.

Finally, you can *delay work*. This can be especially useful for longer-running tasks, where you are going outside of normal working hours. Being able to push work to the start of the workday, when more people might be available (or when current responders might have had a chance to get some sleep), can be hugely beneficial.

Post-Incident Recovery

So the white-hot heat of the incident has passed, the immediate issues have been remediated, and, for the moment at least, the system is back to where it needs to be. What happens next?

After an incident, a lot of people are likely going to be suffering the aftereffects of the stress and adrenaline. As these feelings dissipate, it can be tempting to think you can just get back into the thick of it, but over time, constantly going through the ringer can dramatically impact people and pose negative consequences. People's effectiveness diminishes rapidly as they work under continual stress, and this can lead to burnout and to people leaving the company, among other things. Making sure your post-incident response considers this and creates time for recovery is vital if you want your system to remain resilient over the medium to long term.

Prior to an incident, you should establish guidelines to outline how much recovery time responders get for dealing with incident response, especially if it goes outside of normal business hours. For example, you might state that anyone who receives an overnight page and has to act as first responder gets paid time off in compensation. Of course, it's not always possible to take this time off immediately after the incident, especially if the post-incident report is required promptly. In such a situation, it's important to

schedule this time off as early as possible, and if you're in a leadership position, you should make sure that the responder takes this time. It's there to make sure the most vital parts of your system, humans, have a chance to recover, decompress, and come back as effective as possible. These guidelines need to be clearly communicated within the company. Often the people involved in incident response have responsibilities elsewhere in the organization—for example, they might be working in a team where a product owner is pushing for some features to be shipped. In such a situation, the product owner has to understand that the organization expects the person to take a break.

Providing resources to help responders assess how they are coping in the wake of an incident can be useful. Dianing Yudono, a technical coach who has worked in IT for more than 20 years, has suggested making use of an anonymous self-assessment questionnaire to spot the signs of burnout. The British Medical Association provides one such questionnaire that is targeted at caregivers but could be very useful to others, according to Yudono—the questions seem fairly generic and could easily be applicable to most roles. The fact that this is independent from the company can help people give honest answers, and then you as an organization can provide resources to help responders based on what they've learned about themselves.

Post-Incident Review

> *Failure is the condiment that gives success its flavor.*
>
> —Truman Capote

No aspect of incident response gets as much attention (and can generate as much angst) as the *post-incident review* (PIR). For some, the process of collating and sharing a PIR can be as stressful as the incident itself.

The PIR process begins once the incident is resolved. It should be structured to maximize the amount an organization can learn from an incident, to identify work to improve the system in the future, and to

provide transparency into what happened to stakeholders (both internal and external).

A bad PIR can snatch defeat from the jaws of victory, becoming an exercise in blame and recrimination and a tool for political maneuvering—or perhaps an opportunity to whitewash mistakes and pretend they never happened, storing up trouble for the future when the chance to address issues is left by the wayside.

But done right, a good PIR can deliver catharsis—not simply drawing a line under the incident but creating the chance for those involved to have their say, surface actions that can help reduce the chance of future incidents, and make the system stronger. Sharing the PIR more widely can help provide transparency into the hard work that goes on to keep distributed systems running, earn trust from your users, and help the industry as a whole get better. After all, I've cited multiple examples of incidents throughout this book and used them to illustrate issues and remediation; that would not have been possible if the results of PIRs weren't made more widely available.

So what's involved in setting up your PIR for success?

POSTMORTEM VERSUS POST-INCIDENT REVIEW

You'll notice I am using the term *post-incident review* rather than *postmortem*. For some, the two terms are interchangeable, but given that *postmortem* is often used by some as the name for the document detailing what happened and proposing remedies, I thought it helpful to use the term *post-incident review* to define all the activity around the review itself.

Do You Need One?

Not all incidents will require a full PIR. Some incidents may be very minor and thus can be resolved quickly. In such situations, the responder might want to more informally share what happened and what they did, but that

might be all that's required. As the severity of an incident's impact increases, however, so too does the requirement for a fuller PIR.

A sensible starting point may be to tie the requirement for a review to the severity level of the incident, although you should be careful about this. On the one hand, such an approach seems sensible, but as we've already touched on a few times in this book, you get what you measure. An environment in which people might want to avoid a full PIR could result in them consciously or unconsciously decreasing the severity of an incident. Tying your severity level to quantitative data, such as the number of customers impacted or the period of downtime, might remove some of these concerns.

Also note that some lower-severity "near miss" incidents might warrant a full review. For example, you might have a low-severity incident triggered by a database approaching its storage limit. If you resolve the incident quickly, perhaps by dynamically scaling storage, the temptation might be to just move on with your day. However, if this was your critical database, and you realize that the issue could recur, then you may well want to have a more formal review to ensure the issue gets appropriate attention going forward.

Who Is Involved?

A PIR will need a point person who is responsible for running the review and preparing the final report. Depending on the scope of the incident and the review, that person may need one or more people assisting them. Responders who worked on the incident would help contribute to the report and would also be involved in reviewing the draft and proposing actions. Additionally, a host of other stakeholders might need to be involved at various stages. Some might just want to see the final report, whereas others might want to be consulted about proposed mitigations.

In an ideal world, the point person in charge of preparing the review won't have been involved in the incident itself—this gives you the best chance of the review being fair and objective. This might not always be possible,

though. In a smaller organization, it's likely that everyone who could do a post-incident review was involved in the incident. Also, in a highly political, blame-driven organization, asking someone from outside the team to do a post-incident review may lead to a more toxic process. In a situation like that, you might be best off getting someone from outside the company to help.

If you do need to have one of the incident responders prepare the review, then strongly consider choosing someone who was more on the periphery of the incident. They are less likely to be suffering from the emotional fallout of the response itself.

In a regulated environment, you may also need to involve regulators in the PIR. For example, firms that operate under the Digital Operational Resilience Act (DORA) must submit documentation sharing the outcomes of the PIRs for major incidents, and the regulator may well follow up on any associated action plans.

Gather Data

Building up a picture of what happened and what was done to resolve an incident is vital if you want to be able to learn from the event and identify actions for the future.

Valuable data doesn't come just from your system telemetry; it may also include Slack messages, user emails, and recordings (and perhaps transcripts) of video calls. Collating all this information can help you create a timeline of what happened and what was done in response.

All responders should have the chance to view and add their own information to a shared timeline of the incident. It's quite possible that an initial draft of a timeline can be constructed automatically by a helpful AI assistant.

While some data might be available for a long time, other data will start to degrade. Older Slack messages may be harder to retrieve, for example. Most importantly, people's recollections may start to fade. Even if the

formal review of the PIR is going to happen much later, starting the collation of data and construction of the timeline in the immediate aftermath is key.

Learning, Not Blame

The goal of a PIR is to learn and to improve the system. This underscores why the process of analysis in the wake of an incident needs to default to a blameless approach. Creating an environment of psychological safety isn't something that can start during the PIR; it's something that needs to cut across everything you do. How can you make it easy for people to speak up, to share what happened? This type of culture requires lots of ongoing work. We'll look at that topic in more detail in Chapter 15.

I also still come back to the retrospective prime directive as a way to frame the PIR:

> *Regardless of what we discover, we understand and truly believe that everyone did the best job they could, given what they knew at the time, their skills and abilities, the resources available, and the situation at hand.*
>
> —Norman L. Kerth, *Project Retrospectives: A Handbook for Team Review* (Addison-Wesley)

This type of sentiment, together with making it explicit that the goal is about organizational learning and improvement, can contribute greatly toward making the PIR as blameless as possible.

The importance of creating a blameless environment for a PIR is key. Aside from the longer-term poisonous impacts of using a PIR to assign blame, it can also result in problems being laid at the feet of individuals. The issues with that are twofold.

Firstly, in an environment where people immediately get singled out and castigated when they make a mistake, everyone is incentivized to cover up any mistake and not share what happens. This in turn means that minor

incidents might never get looked at and underlying problems might not be identified, and so the system misses a chance to be improved.

The second problem is that when we say that an incident was caused by "human error," we are ignoring the system as a whole. If a system really can be brought down by one person making a mistake, then how robust is that system in the first place? Expecting to avoid future recurrences of that incident by just making sure that people never make a mistake again seems foolish. It would be much better to understand how the system contributed to allowing the mistake to occur, and why the system wasn't robust enough to withstand a simple human mistake.

It is worth calling out that adopting a blameless stance is in general highly sensible; however, this is going to apply primarily in the context of mistakes and accidents. In the case of serious malfeasance by an employee or actions undertaken by malicious external parties, that stance may need to be reexamined. Even in such rare circumstances, however, there may still be wider things to learn from the incident itself.

Root Cause(s)?

As we've touched on multiple times, major incidents are never caused by one thing. This is why it is highly problematic to look for a singular "root cause." Returning to the collapse of the Ronan Point tower block in Chapter 11, multiple factors were at play that resulted in a minor gas explosion causing the partial collapse of the building.

The five whys

The *five whys* is often cited as an iterative technique to help surface multiple root causes behind an incident. The idea is that you ask a question —for example, "Why did the website go down?"—and then follow up as you get each answer: *"The disk on the customer service side was filled up."* "Why did that happen?" *"The logs were filling up."* And so on and so forth . . .

I see value in the five whys technique, especially as a teaching aid. However, it does cause me concern, as I think the way it is used often presents an overly simplistic view of cause and effect. Firstly, it can be used to promote the idea that there is a single underlying root cause, and you just have to keep digging until you find it. Secondly, I think that it is overly simplistic in that it assumes a linear structure behind a failure. It implies that A was caused by B, which in turn was caused by C. In reality, A was caused by B and C, and C happened because of D, which in turn triggered E and F—oh, and we forgot about G, H, and I.

Thinking broadly

In Chapter 13, when discussing the interwoven nature of a sociotechnical system, I shared a few examples of how multiple factors can end up contributing to a failure. For example, the case against simplistic root cause analysis was highlighted strongly in the aftermath of the Therac-25 accidents. From an article by Nancy Leveson and Clark Turner:

> *Accidents are seldom simple—they usually involve a complex web of interacting events with multiple contributing technical, human, and organizational factors. One of the serious mistakes that led to the multiple Therac-25 accidents was the tendency to believe that the cause of an accident had been determined . . . without adequate evidence to come to this conclusion and without looking at all possible contributing factors. Another mistake was the assumption that fixing a particular error (eliminating the current software bug) would prevent future accidents. There is always another software bug.*[4]

To help break me out of this simplistic thinking, I like to come back to the hexagonal model for sociotechnical systems that was introduced in Chapter 13. This can help remind us that there are multiple forces involved in our system, and that when things go wrong, there can often be multiple reasons why. Does your PIR cover the impact of organizational goals? Does it look at how your processes may have impacted things? Does it consider the ways that the corporate culture might have helped (or hindered) incident response?

Other useful questions to ask include “Where were we surprised?” and “Where were we lucky?” Such questions can help surface not just what went wrong but also what went right. If you can find ways to take those lucky moments and build them into a more robust aspect of the system, you may start making your own luck. This approach can also help you spot where an incident ends up being much less severe than it could have been. For example, you avoided deleting a production database because someone got a typo in a command they were running. This incident as a whole might be seen to have a lower degree of severity (the database wasn’t deleted), but the fact that it was only luck that rescued you may well increase the priority of any identified fixes.

Visualizing root causes

Ishikawa diagrams, also known as fishbone or herringbone diagrams, present a more realistic representation for reasoning about root causes that in turn can help us think more broadly about cause and effect. As shown in Figure 14-5, the visual makes it clear that multiple forces can be at play, and it can help break people out of linear thinking.

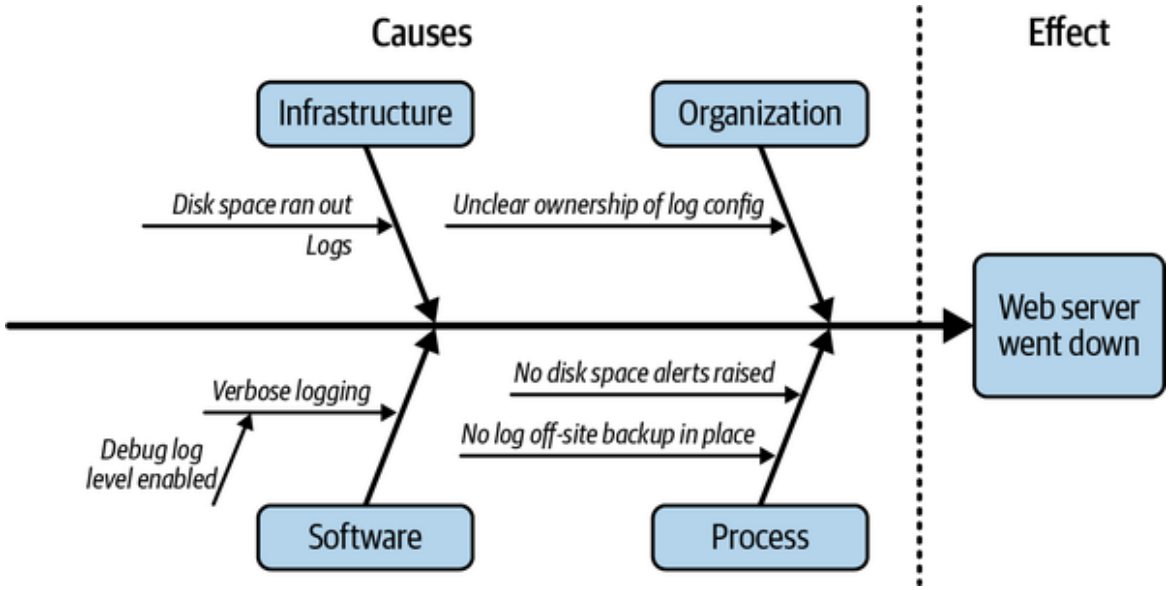


Figure 14-5. An Ishikawa, or fishbone, diagram showing cause and effect

An Ishikawa diagram consists of multiple categories of causes that lead to an end effect. Which categories of causes you use varies greatly and will

likely be impacted by the context you work in. To keep it simple, I've selected four categories derived from the hexagonal sociotechnical model we explored in Chapter 13, but there are a number of other commonly used sets of categories. In the Toyota Production System, for example, which has been highly influential in Lean manufacturing circles, the company made use of the five M's—Manpower/Mindpower, Machines, Materials, Methods, and Measurement. Having these different categories of causes helps you think more broadly about which causes might be at play. You may well decide to tweak these categories over time to make sure you are capturing the right things.

Within each category, you list out the causes, as well as capturing any secondary causes. This is where a five whys style of inquiry can help you dig deeper to discover underlying issues. For example, in Figure 14-5, under "Software," you can see that verbose logging was itself exacerbated by the debug log level being enabled.

For me, these types of diagrams go further than most in highlighting the messy and nonlinear causes that can end up causing incidents. That said, these diagrams are not without flaws, as they imply a hierarchical structure to the incident, whereas I think the structure is closer to a directed acyclical graph. In Figure 14-5, for example, the debug log level being enabled and not turned off might in turn be caused by an unclear ownership of the log configuration.

Action-Oriented

A PIR that generates a set of general recommendations with no owners or timescales for when work should be done can end up being a borderline waste of effort. You spend the time on the PIR, taking time from people who are likely already stretched, and because the PIR happened, people assume that lessons were learned and the system will improve. But without actual actions and follow-up, you can end up with the illusion of systemic learning and improvement without any of that actually happening.

So your PIR needs to generate real, useful actions. The SMART framework for actions can be useful to remember here. Actions should ideally be as follows:

Specific

: The action should have a concrete definition of what you want to achieve and who will own it.

Measurable

: When the action is done, you should be able to confirm this with data.

Achievable

: Listing unrealistic goals is going to be dispiriting. If the work that you think needs to be done seems like too much, try breaking it down into smaller chunks.

Relevant

: This should be a worthwhile activity. You may decide not to undertake an action that might be complicated if it only allows you to hedge against low-risk issues, for example.

Time-based

: You should have a timescale for completing the work. This should result in the action making its way into future planning exercises.

Share a Draft

Once you have a draft of the PIR report ready, it's important that you share it with responders before a final review. Give them some time to take a look and suggest additions before any final sign-off occurs. This will allow them time to decompress after the incident and reflect on what happened. This also gives you the best chance to make sure that people don't feel blindsided by what was discovered or by the proposed actions.

Share the Final Results

If you decide to share the PIR results publicly, it's likely that due care will have to be given to individual privacy and security concerns, at the very least. Individual names will likely need to be redacted, and you may want to obfuscate useful information about internal system details to prevent malicious parties from getting their hands on it. Typically, organizations will create an abridged blog post or something similar, highlighting what happened. A full incident report might spread across multiple pages; but if you want people outside the organization to read it, you'll likely want to produce a much more condensed version!

If you want to see examples, the aforementioned VOID has collated hundreds of postmortems into a public database. Dan Luu has also created his own curated list of postmortems, which he hosts at GitHub—although less exhaustive than the VOID, each entry has a brief summary that makes it easier to sift through to find something you're interested in.

Summary

In this chapter, we've explored how handling an incident can be chaotic, surprising, and stressful. We also talked about how to capture the lessons learned once you've come out the other side. A resilient system cannot sit still—you need to continue to absorb new information so that you can adapt your system as the world around you changes. Taking the opportunity to carry out proper PIRs is one vital aspect of this process.

Constant learning can be done in other forms, though, and this will be the focus of Chapter 15, where we will explore everything from the benefits of game days to the importance of psychological safety.

---

1 John Allspaw, "No matter how many guides are written that insist otherwise, incident response does not follow a linear and predictable sequence", LinkedIn, March 7, 2026.

2 Leila Johannesen and David D. Woods, "Human Interaction with Intelligent Systems: Trends, Problems and New Directions", in *Conference Proceedings 1991 IEEE International*

Conference on Systems, Man, and Cybernetics (IEEE, 1991).

3 John Allspaw, "Real Talk: What We Think We Know—That Just Ain't So", conference lecture, SREcon24 Americas, March 20, 2024, posted April 18, 2024, by USENIX, YouTube.

4 Nancy G. Leveson and Clark S. Turner, "An Investigation of the Therac-25 Accidents", *Computer* 26, no. 7 (July 1993): 18–41.

Chapter 15. Constant Learning and Adaptation

Failure seldom stops you. What stops you is the fear of failure.

—Jack Lemmon

A resilient system isn't one that is rigid; it's one that can bend, flex, and grow. For a system to be resilient, it needs to modify itself as the world around it changes, rather than remaining stagnant. For that to happen, you need to build your system in such a way that you can constantly learn and adapt.

Learning involves more than having some e-learning resources and the odd retrospective. It requires an environment in which everyone feels safe enough to share and challenge. It also requires a relentless focus on continual adaptation.

In this chapter, I'll share a number of aspects required to achieve sustained adaptability and promote a culture of learning. I'll look at how game day exercises and other forms of deliberate practice can help sharpen our skills, how chaos engineering can hold us to account, and why AI may be making critical thinking more difficult. Before all of that, though, we need to look at how we can create the right type of working environment for learning and change.

Psychological Safety

Be excellent to each other.

—William "Bill" S. Preston, Esq.

To create the best opportunity to learn and adapt as an organization, you need to operate in an environment with a high degree of psychological

safety. If you are looking for a broader introduction to fostering psychological safety within a company, then Amy C. Edmondson's book, *The Fearless Organization*,[1] is a good place to start. However, I want to give you a brief overview of what psychological safety is and why it's important in the context of building resilient systems.

What Is Psychological Safety?

When you have a high degree of psychological safety, you feel able to ask questions and share your ideas (and mistakes). You also feel able to challenge others, including people in positions of seniority, without fear of repercussion.

A psychologically safe environment requires empathy. For someone to be able to experience psychological safety themselves, they need to feel welcomed and accepted. Working environments also benefit hugely from individuals being able to communicate about what they would need to feel more psychologically safe.

Psychological safety does not mean that everyone agrees—if anything, the fact that people feel safe enough to be able to raise their concerns means that disagreements might happen more often. However, the environment in which these disagreements surface means that issues can be discussed more openly, and resolutions are much easier to achieve.

Why Psychological Safety Is Vital

If you feel unable to admit when you made a mistake or to raise an issue you have spotted, then your system as a whole will be missing some important information that could help it be more resilient. This can occur at both the macro and micro level. If people feel unable to bring up new ideas or potential solutions during an incident, for example, then your incident response will lack potentially vital information, reducing the quality of response. At a macro level, an unsafe environment will lead to a general deadening of useful sources of information that are vital for adaptation.

Small issues not raised initially can snowball to become major problems. Opportunities to resolve issues earlier are missed.

Multiple studies into psychological safety have shown that it has a material impact in many different areas, from helping reduce patient safety incidents in healthcare to improving innovation in organizations.

But just think for a moment about the impact on the individual. We've already seen the importance of the humans who build and operate our distributed systems. Do you think you are getting the best out of them if they feel that they have to keep quiet? Are they going to feel that they are welcomed? Are they likely to want to stay in an environment like that?

An environment in which people feel psychological safety is an environment in which there is more sharing of information and more learning. Individuals will have the chance to be their best selves at work and will also have the chance to grow.

It's Not Universal

It's important to understand that one person feeling that they have a high degree of psychological safety doesn't mean that other members of the team feel a similar way. For example, someone with a lot of tenure and authority in an organization may well feel able to express themself freely, without fear that this will cause them issues at work. But that psychological safety may come from their position of power in the organization rather than from a general culture that prioritizes the psychological safety of everyone.

> **WARNING**
>
> Don't assume that just because *you* have psychological safety everyone else does too.

Taking that into consideration, it's also important to acknowledge that people can process information and interact in very different ways from one another. According to a 2024 report by the Tech Talent Charter, 53% of

people working in tech identify as being some form of neurodivergent.[2] This is in stark contrast to the 3% of the workforce recorded as being divergent according to employers' own figures. This speaks to a significant underreporting of neurodivergence in the wider tech workforce. Neurodivergent people often don't let their employers know because they don't see the benefit, and because of fears of stigma.[3] This means you may be unaware of the extent to which your colleagues are neurodivergent.

Neurodivergent is a catchall term for people who process information "differently" and covers people with a broad range of conditions, such as autism, ADHD, and dyslexia. Someone with ADHD is likely to have different challenges from someone with autism and will likely bring different benefits, too.

I am far from an expert in this domain—I view myself purely as a conscious incompetent (I know I don't know much!). If you want to know more about how to create a more neuroinclusive environment, "Embracing Neurodiversity in the Workplace" from the UK's National Health Service would be a useful starting place. I also highly recommend Dianing Yudono's talk from LDX London 2026 on designing neuroinclusive incident management.

AI and the Rise of Cognitive Debt

In Chapter 1, I introduced the concept of cognitive debt. A prepublication paper from June 2025 that explored the impact of using LLMs to aid essay writing had this to say about cognitive debt:

> *When individuals fail to critically engage with a subject, their writing might become biased and superficial. This pattern reflects the accumulation of cognitive debt, a condition in which repeated reliance on external systems like LLMs replaces the effortful cognitive processes required for independent thinking.*
>
> *Cognitive debt defers mental effort in the short term but results in long-term costs, such as diminished critical inquiry, increased vulnerability to manipulation, decreased creativity. When participants reproduce suggestions without evaluating their accuracy or relevance, they not only forfeit ownership of the ideas but also risk internalizing shallow or biased perspectives.*[4]

In a software delivery context, a reduction in the ability to engage in critical inquiry and a decrease in creativity seem especially problematic, assuming these observations carry over to the world of software development. From a resilience point of view, we need people who are able to handle surprise and adapt. We need people who can learn, inquire, and improve. If a reliance on AI pushes us away from this, we may see resilience suffer in the long term.

Cognitive debt also seems to manifest itself in the form of a poorer understanding of the system as a whole. When suffering from cognitive debt, an individual might have a good idea about the part of the system they are building but won't necessarily understand how it fits into the whole. Work being done by others is opaque, even within the same team.

But the drive to go faster, to justify the cost of AI, is also causing stress.

As Margaret-Anne Storey points out in her article "How Generative and Agentic AI Shift Concern from Technical Debt to Cognitive Debt", the use of AI agents is encouraging us to work on more things in parallel, increasing our cognitive load:

> *This dynamic echoes a classic lesson from Fred Brooks' Mythical Man-Month. Adding more agents to a project may add more coordination overhead, invisible decisions, and thus cognitive load. Of course, agents can also be used to manage cognitive load by summarizing what changes have been made and how, but the core constraints of human memory and working capacity will be stretched with the push for speed at all costs.*

So if the use of AI is accelerating our cognitive debt, and cognitive debt is making us even less aware of the wider system we are working in, we will be poorly placed to take a sociotechnical view to system resilience. We could end up being more focused within a narrow silo of understanding—going fast perhaps, but at the cost of wider system stability and resilience. We might lose the bigger picture, limiting our ability to improve our resilience.

The *2025 DORA State of AI-Assisted Software Development Report* notes that while the use of AI is increasing the rate at which new functionality is getting shipped, it is also correlated with increased instability. Some have argued that an increase in instability caused by the use of AI tooling might be outweighed by other improvements in productivity, but the report still sees significant problems with increased instability:

> *[I]nstability still has significant detrimental effects on crucial outcomes like product performance and burnout, which can ultimately negate any perceived gains in throughput.*

The Acceleration Whiplash, a report published by Faros in 2026, goes further. Featuring real data gathered from four thousand teams over a two-year window, the report found that while task completion has improved,

> *[b]ugs per developer are up 54%. The incidents-to-PR [pull request] ratio has more than tripled. Median review time has increased 5×. 31% more PRs are merging without any review. The relationship between AI adoption and production quality is worsening, not stabilizing, as adoption deepens.*

I think we are still too early in our use of AI to know whether the correlation of instability with AI use is due to growing cognitive debt, but at

the very least it seems an association worth considering. So what can we do if we're worried that we might be suffering from cognitive debt? Returning to Storey's article, she has some practical advice to help us spot when we might be experiencing cognitive debt:

> *[W]e need better ways to detect cognitive debt before it becomes crippling. Warning signs include: team members hesitating to make changes for fear of unintended consequences, increased reliance on "tribal knowledge" held by just one or two people, or a growing sense that the system is becoming a black box. These may be signals that the shared theory is eroding.*

So what can we do? The use of AI isn't going away, so how do we make sure that we are still able to think critically, work together, and see the bigger picture? While I'm sure we'll learn more about the best ways to work with AI, and I'm sure AI itself will improve, concern about the impact of cognitive debt places even more importance on providing opportunities for people to learn, collaborate, and think critically.

In the rest of this chapter, I'll outline some techniques for creating opportunities for deliberate practice and collaborative working, which may help combat cognitive debt and ensure that you can learn and adapt the system around you.

Build Learning into the Day Job

Several years ago, during a Q&A at a conference called Craft Conf, someone asked me, "How do I find time to learn new skills at work?" By that point, I had been working for myself for a number of years, but this question took me back to my days as a full-time employee. I was lucky that many of my previous employers had given me plenty of time to learn as an individual, but I know from experience that this isn't the case for everyone. The answer I gave was perhaps not super helpful to the person asking the question, but it was correct, at least to my mind. I told them that if their company expected its employees to learn "on their own time," then that was a clear indication that the company didn't really care about learning.

If you are really committed to building a resilient system, setting aside time to help your people learn, grow, and adapt is vital. Creating opportunities to learn at work, both individually and collectively, is key. It's also not as simple as just giving people a learning budget or some online resources (although that can help); it's about being intentional about what learning you are trying to promote.

Let's look at some things you can do to create a working environment that is set up to help people learn.

Collaborative Working

I saw a developer recently argue that one of the main benefits of their AI coding assistant was that it let them discuss how they could solve problems. While I agree with the sentiment, it feels that having discussions with your AI coding assistant is an extremely poor substitute for talking to humans. To give one example of the shortcomings of an AI agent as a partner for collaboration, a key concept of psychological safety is creating an environment in which people can raise ideas and potentially challenge the status quo. Your sycophantic AI agent is unlikely to do a good job of challenging you, and you certainly won't get much experience in practicing empathy and communication skills from talking with your AI buddy.

> **WARNING**
>
> Speaking to your AI coding agent is a poor replacement for talking to your colleagues.

When people talk, ideas are exchanged. This in turn drives us toward debate, discussion, exploration, and critical thinking. If there is concern that the use of AI is accelerating cognitive debt, then making sure you have opportunities to work frequently with other colleagues is really important.

Make sure you have brainstorms around architectural decisions. Design docs are great, but they shouldn't be the only means of working and

collaborating together. If you've not done it before, try *pair programming* (with another human!) on tricky problems—you'll be amazed at how much better your understanding of the problem and the possible solutions will be, and how much you can learn from your human partner. And don't worry—you can still add an AI coding agent to the mix! At worst, you and your pairing partner can have some good problem-solving discussions while the agent burns all your tokens.

Ensemble programming (aka mob programming)[5] can also be very useful. With ensemble programming, a larger group of people (perhaps the whole team) work together on a change. It's primarily about collectively working on a problem and getting input from a large number of individuals.

Of the teams I've spoken to that make use of ensemble programming, most use it only occasionally for specific, tricky problems or important changes, but a lot of development gets done outside of the ensemble exercise as well. As such, while the ensemble programming exercise would likely provide sufficient review of the changes being made during the exercise itself, and in a very synchronous manner at that, you'd still need a way to ensure a review of changes made outside of the mob.

Not only does working together with other humans help you learn from one another and come up with better solutions, but it also helps combat cognitive debt by ensuring that you improve your sense of how everything fits together. It is also a great opportunity to help nurture psychological safety.

Learning from Incidents

As I showed in depth in Chapter 14, the act of carrying out and sharing a PIR is a key opportunity to learn from what happened and put plans in place to change the existing system. But you don't have to stop at learning from your incidents. Many organizations publicly share the postmortems from their incidents. In fact, you can find a large collection of them at the Void.

I'd also strongly suggest setting up the appropriate subscriptions so you can receive the postmortems from any key technology vendors you use.

> **TIP**
>
> Reading other companies' incident reports is a great way to learn from other people's mistakes!

Looking at incident reports from companies that operate in a sector, scale, or technology space similar to yours can create the opportunity to learn from other people's mistakes. In turn, these reports might give you inspiration for running game day exercises or specific chaos engineering experiments (both of which we'll discuss later) to see if your system might suffer from the same issues that other companies face. Just bear in mind that publicly shared incident reports will likely differ quite a bit from the internal versions. If you can find a way to speak with people at a company whose incident report you are interested in, you may well get a much more nuanced description of what went on—this is even more important if the company is one you rely on.

Naive, Purposeful, and Deliberate Practice

Learning through practice is powerful, but not all practice is equal. In his work on building expertise, psychologist K. Anders Ericsson made the case for three main types of practice:

Naive practice

: Growing expertise by doing the same thing repeatedly

Purposeful practice

: Being pushed beyond your current expertise to achieve a specific goal, with feedback as to your progress

Deliberate practice

> Going beyond purposeful practice to include the use of an expert in the area to provide feedback and guidance

We all learn during the course of our day. While carrying out normal activities, we pick things up and work out how to get better at those activities. This sort of basic learning at the job more or less just happens. This is often where a lot of *naive practice* takes place. It's important, though, to recognize that naive practice takes you only to a certain point. Ericsson's research shows that once you achieve an acceptable level of performance, more naive practice in that area doesn't help you improve further.

Purposeful practice takes things further, with the learner working toward a specific target, being pushed more and also receiving feedback as they go. I've seen this done in normal day-to-day activities when a learner is assigned work on a new technology or area of the system specifically to help them grow their skill set. This type of practice is also much more likely to test critical thinking and can help stave off at least some aspects of the AI-induced cognitive debt that I touched on earlier.

Finally, *deliberate practice* requires a clear distinction between expert and novice in a well-defined area of expertise. The expert can guide and coach the novice to help them improve. Training can fall into this bucket, but so too can activities like pair programming, when structured appropriately. Deliberate practice can require more structure and investment but can also deliver better outcomes.

Ericsson explains the distinction between purposeful and deliberate practice thus:

> *We are drawing a clear distinction between purposeful practice—in which a person tries very hard to push himself or herself to improve—and practice that is both purposeful and informed. In particular, deliberate practice is informed and guided by the best performers' accomplishments and by an understanding of what these expert performers do to excel. Deliberate practice is purposeful practice that knows where it is going and how to get there.*[6]

Whereas naive practice can happen without any real focus or investment, both purposeful and deliberate practice require time and resources. In the following sections, we'll look at examples of both purposeful and deliberate practice that you can carry out to help you and your colleagues grow your expertise.

Game Days

Waiting for an incident to learn from isn't great. Firstly, it means an incident has occurred. Secondly, and hopefully, incidents are rare, which means you may not have many chances to learn from one.

Game day exercises are simulated scenarios that give you the chance to test your people and processes in the wake of realistic but fictional incidents. During my time at Google, game day exercises were a fairly common occurrence for various systems, and I certainly think that many organizations could benefit from having these sorts of exercises regularly. Some of these exercises would be small in scale—for instance, an SRE might make use of a "Wheel of Misfortune" to randomly identify a problem and then work with the team to decide how it could be resolved (there is even an open source version of this that you can use!).

Google goes beyond simple tests to mimic server failure, and as part of its DiRT (Disaster Recovery Test) exercises, it has simulated large-scale disasters such as earthquakes.[7]

Game days can be used to probe suspected sources of weakness in the system. In his book *Learning Chaos Engineering*,[8] Russ Miles shares a

game day exercise he facilitated that had been designed in part to examine overreliance on a single staff member called Bob. For the game day, Bob was sequestered in a room, unable to help the team during the simulated outage. Bob was observing, though, and ended up having to step in when the team, in its attempt to fix issues with the "fake" system, ended up logging into production by mistake and was in the process of destroying production data. One can only assume that lots of lessons were learned in the aftermath of that exercise.

Incident.io runs game day exercises regularly. For each game day, there is a specific aspect the company wants to explore. For example, during the outage of US-EAST-1, while incident.io itself was still able to operate, some of its vendors were impacted. So it created a game day to specifically probe this aspect. The first half of a typical incident.io game day is the exercise itself, with the second half dedicated to a team debrief. The company makes it a fun day by providing snacks and even handing out custom stickers specific to the game day.

In incident.io's case, the exercise itself is run by "villains," or bad actors. These are people within incident.io who not only help define the scenario but also role-play as stakeholders during the incident. This allows them to play out various stakeholder management scenarios while also being able to play back observations at the end of the day. The villains have their own Slack channel on which they can coordinate their activities and no doubt dream up nasty surprises for the team dealing with the simulated incident.

For a game day to become deliberate practice, it would require the oversight of an expert. At Google, SREs often play that role, but in a smaller organization you may not have the benefit of an in-house expert, in which case external facilitation can be useful. But even without the benefit of an expert to guide the game day, you can accomplish a huge amount with a self-organized exercise. If you'd like to explore this topic more, I recommend the talk "Don't Wait for an Outage to Improve Your Reliability" by incident.io's Leo Papaloizos, from LDX London 2026, for a very practical guide to running your own game day exercises.

Online Learning Resources

Many organizations unfortunately end up using online learning resources as little more than window dressing. Employees are given access to a website with a bunch of videos but are given no time to make use of them, nor do they receive support or guidance in using them well. The good learning teams I have worked with over the years endeavor to understand what their colleagues want to know and where there may be organizational capability gaps. Those teams will then identify the right sorts of resources to help their colleagues and give them time to use those resources.

Monitoring use of online learning platforms can be especially useful—not to spy on individual employees, but to understand what resources are being heavily used. This might be a reason for some more targeted help and support. I've delivered a few heavily customized training courses for organizations after they realized a lot of their developers were reading my book on O'Reilly's platform, for example. They recognized there was interest and a need and found a way to bring some more targeted help to bear.

More recently, companies such as Uptime Labs have raised the bar in terms of what is possible for online learning. With Uptime Labs, you can simulate an incident; you can see an example drill in Figure 15-1. After the drill, you can get expert feedback on how you did (placing this very clearly in the space of deliberate practice). The fact that this drill can be done online in a time-boxed fashion makes it easy to arrange around other delivery activities.

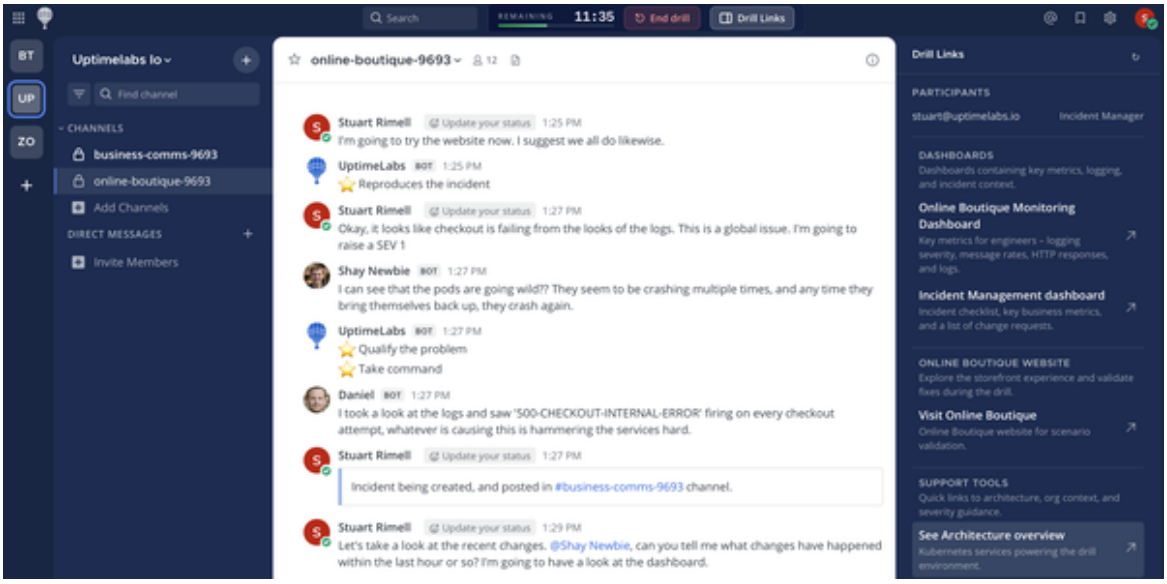


Figure 15-1. An in-progress Uptime Labs drill for a simulated incident

Architectural Katas

> *So how are we supposed to get great architects, if they only get the chance to architect fewer than a half-dozen times in their career?*
>
> —Ted Neward

One of the concerns raised by the growth of AI-induced cognitive debt is that it materially impacts our ability to think critically. Critical thinking requires us to take in lots of different sources of information, explore what we find, and make informed choices.

Critical thinking plays a major part in deciding how our system needs to change. Critical thinking isn't something you are born with. It's something you can practice and get better at. Some of the activities I've already discussed, like game day exercises or other types of deliberate practice such as Uptime Labs' incident simulation, will also naturally test your critical thinking. Personally, I've found the best place to practice my critical thinking is in collaboration with coworkers, and this is where architectural katas come in.

An *architectural kata* consists of a team working together to create a "paper" architecture. They're given a scenario, including both business and

technical context, and a set of problems that need solving, and then they have to work together to come up with an appropriate architecture. In a way, this is a game day exercise, albeit for architecture. A coach guides the team through the scenario and provides feedback.

Ted Neward, who first came up with the concept, has a list of scenarios for you to try out (as well as some guidelines for how to run your own kata). It's also easy enough to come up with your own scenarios to help explore different challenges. As an example, I ran a kata recently in which I had participants come up with an architecture for remote fish farms. The goal was to explore how to handle high latency and frequent partitions in a distributed system that would often be impacted by bad weather.

One of the main benefits of architectural katas is that in the "real" world, architecture evolves slowly, so discussions around trade-offs are few and far between. But an architectural kata, which you could complete in a day or less, gives you lots of scope for discussion and for exploring different ways of carrying out some collective critical thinking.

Chaos Engineering

The term *chaos engineering* was originally inspired by work that Netflix was doing internally. Netflix's system was designed to handle various failure modes, but the Netflix engineers realized that actual infrastructure failures were quite rare. So the idea was to create a real infrastructure failure so they could make sure their systems were able to cope. They hoped that by creating a real but controlled exercise, they would be able to not only validate that these scenarios could be handled successfully but also hold themselves to account to make sure that their services held up to scrutiny.

The most famous of the tools that Netflix created was Chaos Monkey, which during certain hours of the day would turn off random machines in production. Knowing that this can and will happen in production means that the developers who create the systems have to be prepared for it. It's

important to note that this was still a controlled exercise; Chaos Monkey would run only during the workday and away from peak times.

Although the term *chaos engineering* is fairly well-known, the concept suffers somewhat due to competing (and occasionally contradictory) definitions. For many, it means "run a tool on my software and see what it says," and it's not surprising that many of the most vocal proponents of chaos engineering are often themselves selling tools to run on your software to do, well, chaos engineering.

Getting a clear definition of what chaos engineering means to its practitioners is difficult. The best definition (at least to my mind) that I've come across is this on the Principles of Chaos Engineering website:

> *Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production.*

Two words are doing a lot of work in this statement—*experimenting* and *system*. When formulating an experiment, you need to be clear on what you expect to happen and then define an experiment to test this prediction. More appropriately, if you were practicing the scientific method, you would define an experiment to test a hypothesis and then run the experiment. For example, you might have a hypothesis that your system can survive half its machines being taken offline, so you create an experiment to remove half of your machines and confirm whether your hypothesis holds.

In terms of the *system*, some will view this narrowly as the software and hardware components. But in the context of resiliency engineering, it's important that we see the system as being the entirety of the people, processes, culture, and yes, the software and infrastructure that go into creating our product. This means we should view chaos engineering more broadly than just "let's turn off some machines and see what happens." Coming back to our discussion about purposeful and deliberate practice, the use of chaos engineering will need some structure and goals set around it, rather than you just pointing a vendor tool at production and hoping for the best.

So chaos engineering tools very much have their place, not just to test our system but to help us learn about what our system can actually do. Although a lot of the attention on chaos engineering focuses on running these tools in production settings, I'd caution you against loosing chaos engineering tooling on production until you've played around with it a bit in a safer environment! If you want to explore this in more detail, I can recommend the book *Chaos Engineering* by Casey Rosenthal and Nora Jones.[9]

Summary

For us to build truly resilient systems, it's vital that we find ways to learn and improve. This can't be an afterthought, nor should it be relegated to an activity we carry out only after an incident. Waiting for a major incident before deciding to learn and improve is a great recipe for ensuring you have more incidents in the first place.

As I've explained in this chapter, you cannot rely on learning on the job—at best, this delivers a form of naive practice, which very much has its limits. Instead, you need to create space for both purposeful and deliberate practice. Things such as game days, online drills with expert feedback such as those offered by Uptime Labs, architectural katas, and similar exercises can raise the bar in terms of learning outcomes.

Throughout this book, we've explored the concepts of robustness, rebound, graceful extensibility (dealing with surprise), and sustained adaptability. In this chapter, we've finished our exploration of the last of those four concepts, sustained adaptability. You've seen how putting continual learning and adaption at the heart of what your organization does will help you create a system that can be resilient in both the short and the longer term.

Before I leave you entirely, though, I'll summarize everything we've covered in the book in my next and final chapter.

1 Amy C. Edmondson, *The Fearless Organization: Creating Psychological Safety in the Workplace for Learning, Innovation, and Growth* (Wiley, 2018).

2 The Tech Talent Charter is no longer operating, but you can still access an archive of the report.

3 Kaia Newman et al., "Disclosure of Neurodivergence in Software Workplaces: A Mixed Methods Study of Forum and Survey Perspectives", in *Proceedings of the 27th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '25)* (Association for Computing Machinery, 2025), Article 82, 1–17.

4 Nataliya Kosmyna et al., "Your Brain on ChatGPT: Accumulation of Cognitive Debt When Using an AI Assistant for Essay Writing Task", preprint, arXiV, 2025.

5 Although the term *mob programming* is more widespread, I dislike the picture it paints, preferring instead *ensemble programming* as it makes clear that we have a group of people working together rather than an unruly collection of people throwing petrol bombs and breaking windows. I'm not 100% sure who came up with this renaming, but I think it was Maaret Pyhäjärvi.

6 K. Anders Ericsson and Robert Pool, *Peak: Secrets from the New Science of Expertise* (Houghton Mifflin Harcourt, 2016).

7 Kripa Krishnan, "Weathering the Unexpected", *ACM Queue* 10, no. 9 (2012).

8 Russ Miles, *Learning Chaos Engineering* (O'Reilly, 2019).

9 Casey Rosenthal and Nora Jones, *Chaos Engineering* (O'Reilly, 2020).

Chapter 16. Conclusion

Over this book's first 15 chapters, I have shared a mix of concrete practical advice and discussion of larger themes related to building resilient distributed systems. Given that we are at the end of this journey together, I thought now would be the perfect time to recap some of the major themes of the book before summarizing some of the patterns and practices we looked at.

Distributed Systems

The systems we build are distributed—multiple computers communicating over networks that cannot be 100% reliable. This means we have to embrace three fundamental truths with any distributed system we build:

1. You cannot beam information instantaneously between two points.
2. Sometimes you can't reach the thing you want to talk to.
3. Resource pools are not infinite.

While these rules are simple and easy enough to understand in isolation, as the system becomes more complex, the implications of these rules give us a number of headaches.

Resilience Engineering

Trying to make sure nothing ever goes wrong is a problem if that's *all* you do. You can't control everything, and the unexpected will happen. Resilience engineering is all about accepting that fact and focusing instead on *making sure that as many things go right as possible*.

David Woods gives us *four concepts for resilience* that are woven through this book:

Robustness

: The ability to absorb expected issues, such as handling a machine crashing

Rebound

: Recovering after a traumatic event

Graceful extensibility

: How well you can handle the unexpected

Sustained adaptability

: Continual learning and transformation

Remember that our desire to make our systems more resilient can backfire—as we increase the complexity of our system to handle more situations, that new complexity can itself become the source of further issues.

Sociotechnical Systems

The systems we build aren't just computers and networks. Each one also involves the people who use and operate the system, the culture they operate in, and the goals of the organization, as shown in Figure 16-1.

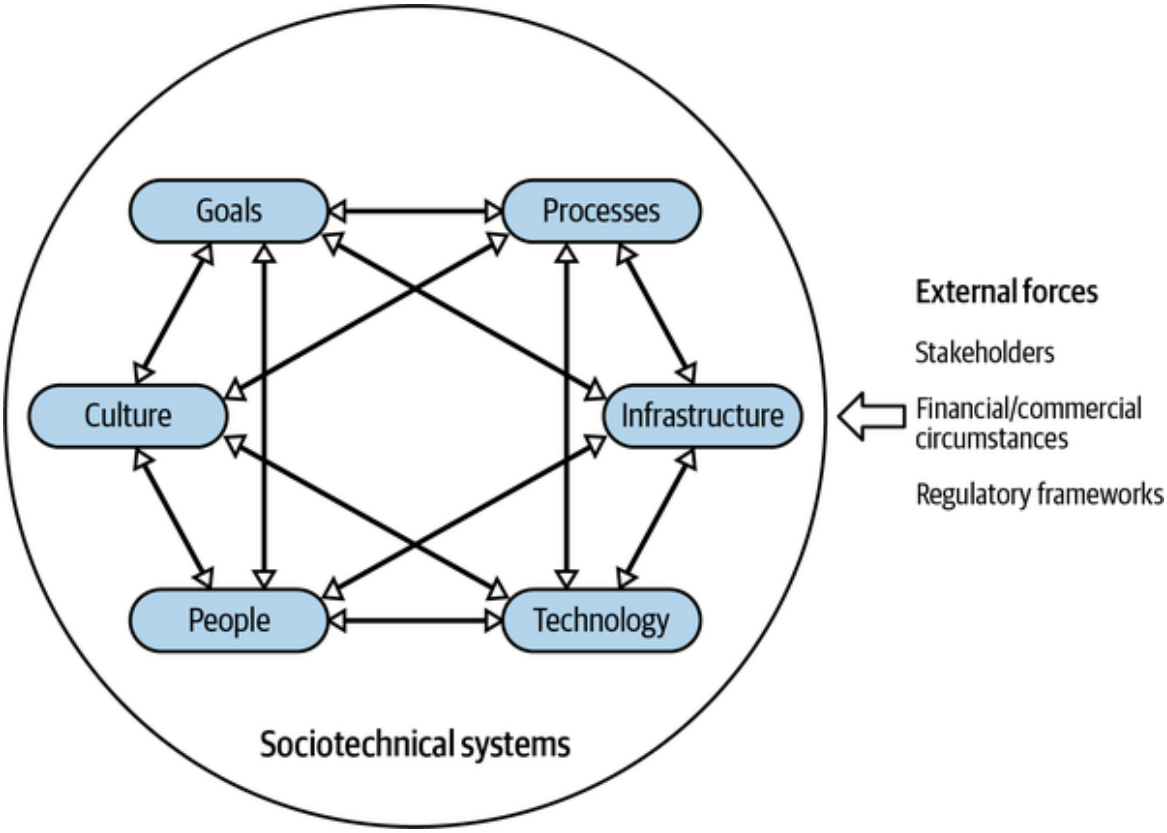


Figure 16-1. A hexagonal model for sociotechnical systems

If you focus solely on technical aspects, you are likely missing more than half the picture.

Resilience and AI

The use of AI to help develop and manage software while also making it part of the products we build appears to be a double-edged sword. The potential upsides are huge, but the industry as a whole is struggling with the implications.

The use of AI tooling seems to make cognitive debt worse, and the use of AI for software delivery is correlated with increased instability, according to at least two industry-wide reports. The increased expense and lack of availability of infrastructure is also causing issues across the industry.

On the other hand, AI coding agents are great at helping us write and understand code, pulling together useful information during incidents, and

making it much easier to navigate our system telemetry.

We are still very much in the midst of the AI revolution, and it's too early to draw any long-term conclusions. Given that, it becomes even more important to experiment and explore the space, while also being very conscious of how we use AI.

Patterns, Practices, and Concepts

Aside from the broader themes, we've covered a number of specific topics in this book. Summarized here are the highlights of the patterns and practices that we looked at.

Observability

Unless we have the right telemetry from our system, we will be unable to know how it is behaving in production. This robs us of the ability to diagnose and deal with anomalies, as well as making it very difficult to assess the suitability of our system to achieve the desired degree of resilience.

Use OpenTelemetry to send information about what your system is doing to a suitable observability tool that supports a rich array of high cardinality data. This will allow you to slice and dice the data you have to ask questions of your system you never knew you needed to ask. Define service level objectives (SLOs) backed by your telemetry to make sure that you'll know if your system is behaving appropriately.

Timeouts

Computers can fail to respond, networks can get overly congested, and sometimes things just take longer than you expect. This is why the *timeout* pattern is so important. For more interconnected systems, you may want to *propagate the timeout* across multiple services to ensure that you stop working once other parties have already given up.

Retries and Idempotency

If you aren't sure whether something worked the first time, you may want to *retry* it. But make sure you have a gap between retries, along with some artificial jitter to make sure all the clients don't retry at the same time. You'll also want an upper limit on your retries to ensure you aren't tying up too many of those precious computing resources.

You also need to be able to make sure your retries are safe—you don't want to accidentally pay someone twice! By making your operation *idempotent*, you ensure that applying it more than once doesn't change the result. The easiest way to make operations idempotent is to use a *client-generated idempotency key*, but if you can't change the client-server protocol, consider the use of *server-side request fingerprinting* instead.

Thundering Herds

Systems can and will be overwhelmed by too much load. These thundering herds can be caused by a variety of forces. Malicious parties can unleash DDoS attacks. The failure of a component such as a cache can cause an internal stampede. And sometimes a product that is too popular can be its own worst enemy, attracting way too much interest. How you deal with these thundering herds depends very much on what caused them in the first place.

Rate Limiting

Controlling the rate at which load can hit part of your system is a key defense against thundering herds. As a simple starting point, if a service has too much load, it can just *load shed*, dropping the excess work it cannot handle.

When a service appears to be erroring, why keep sending it work? With a *circuit breaker*, a client can go from failing slow to failing fast, giving the server time to recover. A circuit breaker is typically either open or closed, though, making it a blunt instrument as a rate-limiting technique; a *token*

bucket approach is trickier to configure but can be a much more elegant solution, especially if you want to avoid the boom and bust problems associated with a circuit breaker.

Machine Failure

The fact that individual computers can fail is something that we understand well. It therefore makes sense to make our systems robust enough to tolerate at least some degree of machine failure.

As machines come and go, using an appropriate *dynamic service discovery* implementation can be vital. *Server-side load balancing* can ensure that clients are unaware that an individual machine is unavailable, as long as other machines are there to handle the load. For more complex tasks, you might want to look at *client-side load balancing* instead. Finally, use *desired state management* to automatically bring the infrastructure back to the desired state when machines fail.

Message Brokers

Message brokers can help shift the responsibility for guaranteed delivery from your application code to a dedicated system. Changing your code to use a message broker can be a complicated affair, however, and it's vital that you operate the broker correctly if you want to secure the guarantees it offers.

If you want to make sure that a message gets put on a broker in the first place, a *transactional outbox* can keep your message safe even if the program tasked with putting the message on the broker fails. To deal with a client crashing when consuming a message, a *consumer-managed inbox* is the answer. If a message in a broker is proving impossible to process, it may be safer to put it in a *dead letter channel* to avoid causing instability, and a *heartbeat message* can help clients know that the broker is still working even if no messages are being sent.

Disaster Recovery

It's not great when a single machine fails. It can be a lot worse when a whole data center goes down. Working out what disaster recovery (DR) approach is best for you requires that you identify your *recovery point objective* (RPO)—how much data loss you can tolerate—and your *recovery time objective* (RTO)—how long you can be down for. Different patterns of increasing complexity can help you achieve the RPO and RTO you want, as shown in Figure 16-2.

Backup and restore is where you need to start, but you could be offline for hours or even days. With a *pilot light* setup, data is replicated to a standby site to keep RPO low, but it might take time to provision the infrastructure you need when the failover happens. A *warm failover* means that some of your system may work almost immediately. If you want the lowest RPO and RTO possible and can justify the cost and expense, then *cross-site active-active* is the way to go.

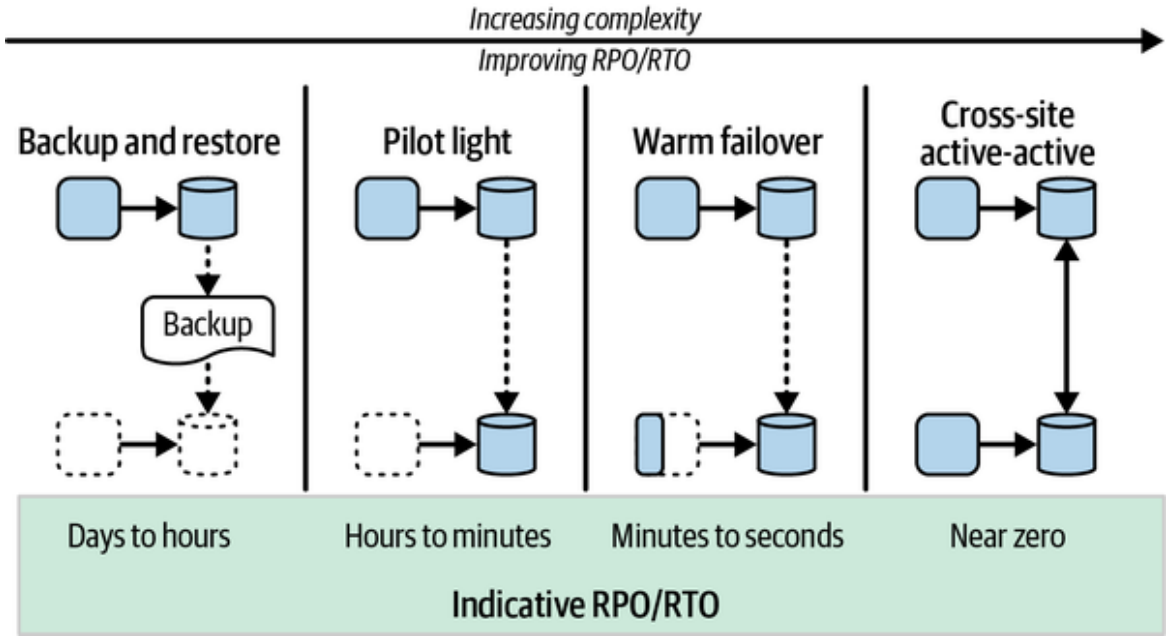


Figure 16-2. The key objectives to define as part of selecting an appropriate DR strategy

Progressive Collapse

A progressive collapse describes the phenomenon of a small issue that snowballs, triggering other problems in the system. The small inciting

incident results in an outsized impact. Progressive collapses can be mitigated in three main ways:

Reduce hazards

Reduce the causes of component failure.

Strengthen individual components

Reduce the chance of a single component failing.

Reduce interconnection

Ensure that the failure cannot cascade across to other components.

Using a *bulkhead* can reduce interconnection, as can *distributing across failure planes* to ensure that a failure in one area doesn't take out the whole system. *Cell-based architectures* take these ideas further, creating isolated subsets of the overall system. When things do go wrong, you need to *gracefully degrade* your system's functionality to minimize the impact on the user. Finally, a *critical standby system* with no shared code or infrastructure may be the ultimate defense against a progressive collapse.

Formal Methods

When parts of your system fail, you have to decide how you want your system to adapt. *CAP theorem* describes the trade-off between consistency and availability that must occur when you have a partition (which prevents one part of your system from speaking to another). *PACELC theorem* extends CAP to show that even when there is no partition, you have to trade off between consistency and latency. Of course, you can make different types of trade-offs for different parts of your system.

Managing resource pools is important to keep your system resilient, which is why processing work quickly is better than processing work slowly. When you process work quickly, you free up computing resources for other needs. *Little's Law* highlights the interplay between work coming in,

processing time, and work in progress, helping show where the levers are for you to adapt your system's load and throughput.

Incident Management

When things go wrong, fixing the problem as quickly as possible is key. So too is making sure you keep people informed as to what is going on. *Incident response*, though, is messy, involving a number of different activities: you need to make sure you have the right people who are able to deal with incident response, and that they are supported, have training, and have tools they know how to use.

Avoid fixed checklists, as these can constrain incident response. Instead, make sure that you discuss beforehand how you are going to handle *communication* about the incident, *delegating* work to others, *escalation* when required, *coordinating* with the rest of the team, and dealing with *saturation* when you are overwhelmed.

After the dust has cleared, gather data to build a timeline of the incident, and gather testimony from those involved to build a picture of what happened. Discuss what changes you want to make in the future, and share the postmortem with interested parties.

Constant Learning

For a system to be resilient, you need to learn and adapt. *Psychological safety* is vital—everyone needs to be able to ask questions, share their ideas, and even challenge one another without fear of repercussions. Without that environment, your desire to learn and grow will be massively impacted.

Learning on the job is useful but rarely sufficient. Look into structured learning opportunities such as ensemble programming, game days, and architectural katas. Online self-guided learning can also be useful, but you need to make sure people are given the time and guidance to use these resources well.

Final Thoughts

The advice in this book is meant to be practical and useful today. But the world can and will change. The key to being resilient is to adapt and change. I've laid many breadcrumbs throughout this book to help you continue your own learning, but if you take away only one thing from this book, I hope it is that staying curious, exploring, questioning, and experimenting are key to staying resilient over the long term. Just because you have finished this book doesn't mean your own journey into the world of resiliency has finished.

Appendix. Pattern Index

backup and restore	A disaster recovery pattern that involves restoring data from a backup and loading it into a replacement system.
bulkhead	A mechanism to isolate the impact of a failure to help mitigate the risk of a progressive collapse.
cell-based architecture	An approach in which a collection of collaborating services are deployed and communicate within an isolated failure plane.
circuit breaker	Breaks the connection from a client to a server when an error condition is identified in the server, allowing subsequent calls from the client to fail fast.
client idempotency key	An ID generated on the client side to allow a server to identify whether this operation has already been processed.
client-side load balancing	A client is made aware of the available server nodes and makes its own determination on which node to send work to.
consumer-managed inbox	A consumer-managed store of messages that have been delivered but not yet processed.
critical standby system	An alternative system providing only critical functionality that shares as little as possible with the primary system and is used in failover scenarios.
cross-site active-active	The system is distributed across two or more sites, with writes being able to be sent to any site.

dead letter channel	A place where messages that cannot be delivered or safely processed can be quarantined.
desired state management	Continually adapting a system to ensure that its current state matches the desired state.
distribute across failure planes	Spreading a workload across multiple planes to ensure that a failure in one plane doesn't take out all functionality.
dynamic service discovery	Clients can receive new addresses for servers as they change.
graceful degradation	Reducing the functionality of a system while still continuing to operate in reaction to part of the system's functionality being unavailable.
hardcoded IP resolution	IP address for a networked endpoint is hardcoded in the client.
idempotent operation	An operation that can be applied multiple times without changing the result.
leaky bucket rate limiting	Ensures that the rate of calls from a client to a server can remain consistent to avoid overloading the server.
load shedding	Work that would otherwise overwhelm a server is rejected.
pilot light failover	A disaster recovery pattern that constantly syncs data to a failover site, with infrastructure provisioned only after

	failover.
producer heartbeat message	A regular message sent by a producer to let the consumers know that the message channel is working and that the producer is capable of sending messages.
retry remote operation	Reattempting an operation on a remote endpoint when you cannot confirm that the previous attempt succeeded.
server-side load balancing	Load sent to a server is transparently distributed across a number of nodes.
server-side request fingerprinting	The server stores a representation of a request to spot when identical requests are subsequently received.
timeout	A time-based threshold after which an operation will be terminated.
timeout budget propagation	Propagate the timeout budget across a series of operations to ensure that work is terminated when the overall timeout expires.
transactional outbox	A store for messages that are yet to be sent over a broker, managed within producer-local transactional boundaries.
warm failover	A disaster recovery pattern that syncs data to a failover site, with at least some infrastructure already provisioned to handle traffic.

Glossary

accord-based backpressure

A form of backpressure that requires the client and server working together.

alert

A notification that something is wrong.

availability zone (AZ)

A location within a public cloud region, within which computing resources can be provisioned. Each AZ is geographically close to another to reduce latency.

backpressure

Reducing the amount of work being sent when a threshold is reached.

blackhole routing

Silently dropping traffic, often used to mitigate against a DDoS.

Border Gateway Protocol (BGP)

Defines the best route for traffic over a network.

bulkhead

A component that stops failure spreading in a system.

cache stampede

A specific type of thundering herd problem, caused when a recently restarted cache that has yet to refill is overwhelmed by traffic.

CAP theorem

A formal theory stating that in a network partition in a distributed system, you have to trade off between consistency and availability.

cellular architecture

A form of sharded architecture, with each shard consisting of a set of collaborating services and data, where calls are routed locally within each shard.

chaos engineering

A discipline that involves experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production.

circuit breaker

Breaks the connection from a client to a server when the client identifies a persistent problem in communication with the server, allowing subsequent calls from the client to fail fast.

client idempotency key

An ID generated on the client side to allow a server to identify whether this operation has already been processed.

client-only backpressure

Backpressure triggered purely by the client. A circuit breaker is an example of client-only backpressure.

client-side load balancing

Load balancing that is performed by each client.

clock drift

The phenomenon explaining how two different clocks in two different computers can indicate different times, even if they were at one point synchronized.

cloud region

A public cloud concept that refers to a geographic region made up of multiple availability zones (AZs). Some cloud resources can be provisioned at the region level, others at the AZ level.

conflict-free replicating data type (CRDT)

A data type that can be replicated, have its copies changed independently from one another, and then be merged back into a predictable state.

consumer

A program that is consuming the functionality of a service.

consumer-managed inbox

A consumer-managed store of messages that have been delivered but not yet processed.

content delivery network (CDN)

A system that caches web content close to the user, reducing load on the origin and speeding up delivery of content. Some CDN services will also provide protection against denial of service (DoS) attacks.

critical standby system

An alternative system providing only critical functionality that shares as little as possible with the primary system, to be used in failover scenarios.

dead letter channel

A quarantine for messages that cannot be processed.

deliberate practice

: Goes beyond purposeful practice to include the use of an expert in the area to provide feedback and guidance.

desired state management

: Provided by a tool or platform to continually ensure that your infrastructure/system matches your desired state as closely as possible.

disaster recovery (DR)

: The process of recovery of key systems (primarily IT), in the wake of a major incident, such as a data center failure.

distributed denial of service (DDoS) attack

: A coordinated effort to use a network of machines to attack your system and interrupt its ability to function.

distributed system

: A system consisting of two or more computers that communicate with one another via networks.

Domain Name System (DNS)

: Allows for domain names to be mapped to IP addresses.

error budget

: A defined amount of allowable error, calculated in terms of a specific SLO.

error rate

: The frequency with which errors are generated.

eventual consistency

Describes where multiple replicas in a system will eventually converge to the same state, once all messages to those replicas for state changes have been processed.

graceful degradation

Reducing the functionality of a system while still continuing to operate in reaction to part of the system's functionality being unavailable.

graceful extensibility

A system's ability to handle the unexpected.

hardcoded IP resolution

IP address for a networked endpoint is hardcoded in the client.

idempotent operation

An operation that can be applied multiple times without changing the initial result.

incident commander

A point person responsible for an incident.

latency

How long something takes.

leaky bucket rate limiting

A pattern used to regulate the rate of calls made from a client to a server.

Little's Law

A theorem from queuing theory stating that Items in the system (L) = Arrival rate (λ) × Processing time (W).

load balancing

Attempting to distribute work across a number of machines such that work is evenly distributed.

load sharing

Sharing work across a number of machines without any guarantees that the work will be evenly distributed.

load shedding

Rejecting new work when a threshold is reached.

message broker

Software that allows programs to communicate among themselves by sending messages via a message channel, typically providing guarantees around both durability and delivery.

message channel

A mechanism(typically implemented as a queue or a topic) that allows one program to communicate with one or more recipients, hosted by a message broker.

naive practice

Growing expertise by doing the same thing repeatedly.

Network Time Protocol (NTP)

A standard protocol for reducing the skew between clocks in different computers.

observability

A property of a system that describes to what extent the internal behavior of the system can be understood by external signals.

OpenTelemetry (OTel)

An open framework for observability that makes it possible to instrument your system in different ways and then collect that information.

OpenTelemetry Protocol (OLTP)

The standard protocol used to send OpenTelemetry logs, metrics, and traces.

PACELC theorem

An extension of CAP theorem showing that in the absence of a network partition, you still need to trade off between latency and consistency.

partition

When one part of a distributed system can't see another part of the system.

pilot light failover

A disaster recovery pattern that constantly syncs data to a failover site, with infrastructure provisioned only after failover.

post-incident review (PIR)

The process that occurs after an incident is resolved; the details of the incident and its resolution are shared, and actions are identified to improve the system in the future.

postmortem

For some people, this is used interchangeably with a PIR. I use this to refer to the specific document produced by a post-incident review.

Precision Time Protocol (PTP)

A more precise alternative to NTP, PTP normally makes use of specialist hardware to decrease the clock drift between machines even further.

producer

Primarily used in a message broker context, it refers to the service that is generating messages to be consumed by a consumer.

producer heartbeat message

A message sent by a producer over a message channel to let the consumers know that the message channel from the producer is still working.

progressive collapse

A small inciting incident that causes an outsides failure in the wider system.

psychological safety

The extent to which an individual feels able to ask questions, share ideas, and challenge colleagues without fear of repercussion.

purposeful practice

Being pushed beyond your current expertise to achieve a specific goal, with feedback as to your progress.

rate limiting

Limiting the amount of work that a system is allowed to do.

rebound

The ability of a system to recover after a problem has occurred.

recovery point objective (RPO)

Defines the maximum amount of time since data the last recovery point for data.

recovery time objective (RTO)

Defines how long it's acceptable for your service to be unavailable in the wake of an incident.

resilience

The ability of a system to withstand problems, recover from issues that impact the system, and continue to evolve in order to maintain resilience even as the wider context of the system changes around it.

resilience engineering

A discipline focusing on making sure as many things go right as possible.

responder

A person involved with actively helping investigate and resolve an incident.

retry

Trying something for a second (or subsequent) time after the initial attempt failed.

robustness

The ability of the system to deal with an expected issue.

Safety-I

A traditional view of safety management that focuses on reducing adverse events (e.g., failures) to reduce adverse outcomes.

Safety-II

A focus on making sure as many things go right as possible. Resilience engineering is the discipline that helps us implement Safety-II.

safing

The immediate actions taken to stabilize the system after an anomaly. These actions are often, although not always, automated.

saturation/saturated

In the context of computing resources, something is "saturated" when no more of the computing resource is left, as it is all being consumed by existing work.

server-side request fingerprinting

The server stores a representation of a request to spot when identical requests are subsequently received.

service level agreement (SLA)

An agreement with an end user regarding what the system must provide. This is often, but not always, a contractual obligation with a paying customer.

service level indicator (SLI)

An individual measurement of some aspect of your system's service level, such as latency or availability. Provides measurement against a specific SLO.

service level objective (SLO)

A target that a team signs up to that defines an acceptable service level over a period of time. For example, an SLO could define a 99.9% uptime over a three-month period.

service mesh

A network of proxies that sit between services to provide shared behavior around discovery, connection, and communication.

sociotechnical system

A system that consists of people working with technology.

stateful processing

: State required for processing is held in memory, with memory persisted between processing work items.

stateless processing

: Each item of work is treated entirely independently, with any required state for the processing either provided in the work item itself, or can be fetched from remote locations.

sustained adaptability

: Adapting a system in reaction to what you've learned and changing circumstances to ensure that the system remains resilient over the long term.

thundering herd

: A flood of work that threatens to overwhelm part or all of the system.

timeout

: A duration after which an operation should be terminated if not yet completed.

timeout budget propagation

: A shared timeout is propagated across a series of operations to ensure that work is terminated when the budget is exhausted.

transactional outbox

: A store for messages that are yet to be sent over a broker, managed within producer-local transactional boundaries.

universally unique identifier (UUID)

: A standard for generating a unique 128-bit number that can be generated entirely locally.

warm failover

: A disaster recovery pattern that syncs data to to a failover site, with at least some infrastructure already provisioned to handle traffic.

Bibliography

1. Abadi, Daniel. “Consistency Tradeoffs in Modern Distributed Database System Design: CAP Is Only Part of the Story.” *IEEE Computer* 45, no. 2 (2012): 37–42. *https://doi.org/10.1109/MC.2012.33*.

2. Allspaw, John. “No matter how many guides are written that insist otherwise, incident response does not follow a linear and predictable sequence.” LinkedIn post, March 2026. *https://www.linkedin.com/posts/jallspaw_lorin-hochsteins-right-no-matter-how-many-share-7436237040674992129-vOS5*.

3. Allspaw, John. “Real Talk: What We Think We Know—That Just Ain’t So.” SREcon24 Americas, USENIX. YouTube video, 2024. *https://www.youtube.com/watch?v=eiRsqjfO7zI*.

4. Amazon Web Services. “Disaster Recovery (DR) Objectives.” AWS Well-Architected Framework: Reliability Pillar. Accessed July 7, 2026. *https://oreil.ly/dwjOU*.

5. Amazon Web Services. “Summary of the Amazon DynamoDB Service Disruption in the Northern Virginia (US-EAST-1) Region.” AWS Message, October 20, 2025. *https://aws.amazon.com/message/101925*.

6. Antoniuk, Daryna. “Russian Top-Level Internet Domain Suffers Massive Outage.” *The Record*, January 31, 2024. *https://therecord.media/russia-top-level-domain-internet-outage-dnssec*.

7. Beyer, Betsy, Chris Jones, Jennifer Petoff, and Niall Richard Murphy, eds. *Site Reliability Engineering: How Google Runs Production Systems*. O’Reilly, 2016.

8. Bourget, Erik, Toby Reyelts, and Xavier Shay. “Incident Summary: 2017-03-16.” *Square Corner Blog*, March 17, 2017. *https://medium.com/square-corner-blog/incident-summary-2017-03-16-2f65be39297*.

9. Dai, T., J. He, X. Gu, and S. Lu. “Understanding Real-World Timeout Problems in Cloud Server Systems.” In *Proceedings of IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018. *https://doi.org/10.1109/IC2E.2018.00022*.

10. Davis, M. C., R. Challenger, D. N. W. Jayewardene, and C. W. Clegg. “Advancing Socio-Technical Systems Thinking: A Call for Bravery.” *Applied Ergonomics* 45, no. 2 (2014): 171–180. *https://doi.org/10.1016/j.apergo.2013.02.009*.

11. DORA. *State of AI-Assisted Software Development 2025*. Google, 2025. *https://oreil.ly/HBFqw*.

12. Down, Aisha. “Cloudflare Apologises After Latest Outage Takes Down LinkedIn and Zoom.” *The Guardian*, December 5, 2025. *https://oreil.ly/1i6aE*.

13. Edmondson, Amy. *The Fearless Organization: Creating Psychological Safety in the Workplace for Learning, Innovation, and Growth*. Wiley, 2018.

14. Ellingwood, Bruce R., Robert Smilowitz, Donald O. Dusenberry, Dat Duthinh, H.S. Lew, and Nicholas J. Carino. *NISTIR 7396: Best Practices for Reducing the Potential for Progressive Collapse in Buildings*. NIST, 2007.

15. Ericsson, K. Anders and Robert Pool. *Peak: Secrets from the New Science of Expertise*. Houghton Mifflin Harcourt, 2016.

16. Fabio, Adam. “Killed by a Machine: The THERAC-25.” *Hackaday*, October 26, 2015. *https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25*.

17. Fastly. “Fastly DDoS Protection Datasheet.” Accessed August 25, 2026. *https://www.fastly.com/resources/datasheets/security/fastly-ddos-mitigation-datasheet*.

18. Fox, Armando, and Eric A. Brewer. “Harvest, Yield, and Scalable Tolerant Systems.” In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS ’99)*. IEEE Computer Society, 1999.

19. Gilbert, Seth, and Nancy Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services.” *ACM SIGACT News* 33, no. 2 (June 2002): 51–59. *https://doi.org/10.1145/564585.564601*.

20. Golab, Wojciech. “Proving PACELC.” *ACM SIGACT News* 49, no. 1 (2018): 73-81.

21. GSMA. *The Mobile Economy 2026*. GSMA, 2026. *https://oreil.ly/l3r_M*.

22. *The Guardian*. “Much of West and Central Africa Without Internet after Undersea Cable Failures.” March 14, 2024. *https://oreil.ly/2KFiL*.

23. Helland, Pat. “Don’t Get Stuck in the CON Game (V3).” *Scattered Thoughts on Distributed Systems* (blog), June 30, 2021. *https://pathelland.substack.com/p/dont-get-stuck-in-the-con-game-v3*.

24. Hidalgo, Alex. *Implementing Service Level Objectives*. O’Reilly, 2020.

25. Hohpe, Gregor, and Bobby Woolf. *Enterprise Integration Patterns*. Addison-Wesley, 2003.

26. Hollnagel, Erik. *FRAM: The Functional Resonance Analysis Method: Modelling Complex Socio-Technical Systems*. CRC Press, 2012.

27. Hollnagel, Erik, Robert L. Wears, and Jeffrey Braithwaite. *From Safety-I to Safety-II: A White Paper*. University of Southern Denmark, University of Florida; Macquarie University, 2015.

28. Johannesen, Leila and David D. Woods. "Human Interaction with Intelligent Systems: Trends, Problems, and New Directions." In *Conference Proceedings 1991 IEEE International Conference on Systems, Man, and Cybernetics*. IEEE, 1991. *https://doi.org/10.1109/ICSMC.1991.169874*.

29. Kleppmann, Martin, and Chris Riccomini. *Designing Data-Intensive Applications*. 2nd ed. O'Reilly, 2026.

30. Konieczny, Bartosz. *Data Engineering Design Patterns*. O'Reilly, 2025.

31. Kosmyna, Nataliya, Eugene Hauptmann, Ye Tong Yuan et al. "Your Brain on ChatGPT: Accumulation of Cognitive Debt When Using an AI Assistant for Essay Writing Task." Preprint, arXiv, 2025. *https://arxiv.org/abs/2506.08872*.

32. Krishnan, Kripa. "Weathering the Unexpected." *ACM Queue* 10, no. 9 (2012). *https://queue.acm.org/detail.cfm?id=2371516*.

33. Leveson, Nancy. "A New Accident Model for Engineering Safer Systems." *Safety Science* 42, no. 4 (2004): 237–270. *https://oreil.ly/Vnmx-*.

34. Leveson, Nancy G. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

35. Leveson, Nancy G., and Clark S. Turner. "An Investigation of the Therac-25 Accidents." *Computer* 26, no. 7 (July 1993): 18–41.

36. Majors, Charity, Liz Fong-Jones, and George Miranda. *Observability Engineering*. 2nd ed. O'Reilly, 2026.

37. McMahon, Liv. "Amazon Apologises to Customers Impacted by Huge AWS Outage." *BBC News*, October 23, 2025.

https://www.bbc.co.uk/news/articles/cvgvnp77dy9o.

38. Miles, Russ. *Learning Chaos Engineering*. O'Reilly, 2019.

39. Mohan, Gayathri. *Full Stack Testing*. 2nd ed. O'Reilly, 2026.

40. Morales, Jowi. "Amazon's Middle East Data Centers Damaged by Iran Drone and Missile Attacks Will Be Down for Several Months during Repairs." *Tom's Hardware*, May 3, 2026. *https://oreil.ly/HtUFB*.

41. Morris, Kief. *Infrastructure as Code*. 3rd ed. O'Reilly, 2025.

42. National Transportation Safety Board. *MIR 25-10: Safeguarding Bridges from Vessel Strikes: Need for Vulnerability Assessment and Risk Reduction Strategies*. NTSB, 2025.

43. National Transportation Safety Board. *MIR 25-40: Contact of Containership* Dali *with Francis Scott Key Bridge and Subsequent Bridge Collapse*. NTSB, 2025.

44. National Transportation Safety Board. *Safety Recommendation Report: Assumptions Used in the Safety Assessment Process and the Effects of Multiple Alerts and Indications on Pilot Performance*. NTSB, 2019.

45. Naur, Peter. "Programming as Theory Building." Originally published 1985. Reprint in *Computing: A Human Activity*, ACM, 1992.

46. Newman, Kaia, Sarah Snay, Madeline Endres, Manasvi Parikh, and Andrew Begel. "Disclosure of Neurodivergence in Software Workplaces: A Mixed Methods Study of Forum and Survey Perspectives." In *Proceedings of the 27th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '25)*. Association for Computing Machinery, 2025. *https://doi.org/10.1145/3663547.3746334*.

47. Newman, Sam. *Monolith to Microservices*. O'Reilly, 2019.

48. Nygard, Michael T. *Release It!* 2nd ed. Pragmatic Bookshelf, 2018.

49. OECD, *Health at a Glance 2025: OECD Indicators*. OECD Publishing, 2025. *https://doi.org/10.1787/8f9e3f98-en*.

50. Organisation for Economic Co-operation and Development. "Health Spending." Accessed July 7, 2026. *https://www.oecd.org/en/data/indicators/health-spending.html*.

51. O'Toole, Garson. "Quote Origin: No Plan Survives First Contact with the Enemy." *Quote Investigator*, May 4, 2021. *https://quoteinvestigator.com/2021/05/04/no-plan*.

52. President's Commission on the Accident at Three Mile Island. *The Need for Change: the Legacy of TMI: Report of the President's Commission on the Accident at Three Mile Island*. The Commission, 1979.

53. RabbitMQ, "Credit Based Flow Control." Accessed August 26, 2026. *https://oreil.ly/S20cj*.

54. Reuters. "Twitter Is Refusing to Pay Its Google Cloud Bills, Platformer Reports." June 10, 2023. *https://www.reuters.com/technology/twitter-is-refusing-pay-its-google-cloud-bills-platformer-2023-06-10*.

55. Rosenthal, Casey, and Nora Jones. *Chaos Engineering: System Resiliency in Practice*. O'Reilly, 2020.

56. Schroeder, Bianca, Adam Wierman, and Mor Harchol-Balter. "Open Versus Closed: A Cautionary Tale." In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation (NSDI '06)*. USENIX Association, 2006.

57. Shankar, Shreya, and Hamel Husain. *Evals for AI Engineers*. O'Reilly, 2026.

58. Shorrock, Steven. "Alarm Design: From Nuclear Power to WebOps." *Humanistic Systems*, October 16, 2015.

https://humanisticsystems.com/2015/10/16/fit-for-purpose-questions-about-alarm-system-design-from-theory-and-practice.

59. Stidham, Shaler, Jr. "Technical Note—A Last Word on $L = \lambda W$." *Operations Research* 22, no. 2 (1974): 417–21.

60. Tech Talent Charter. *Diversity in Tech*. Tech Talent Charter, 2024. *https://oreil.ly/fCqaf*.

61. Ukis, Vladyslav. *Establishing SRE Foundations: A Step-by-Step Guide to Introducing Site Reliability Engineering in Software Delivery Organizations*. Addison-Wesley Professional, 2022.

62. Vaughan, Diane. *The Challenger Launch Decision: Risky Technology, Culture, and Deviance at NASA*. University of Chicago Press, 1996.

63. Wikipedia. "Perverse Incentive." Accessed July 7, 2026. *https://en.wikipedia.org/wiki/Perverse_incentive*.

64. Wikipedia. "Quantum Entanglement." Accessed July 7, 2026. *https://en.wikipedia.org/wiki/Quantum_entanglement*.

65. Woods, David D. "Four Concepts for Resilience and the Implications for the Future of Resilience Engineering." *Reliability Engineering & System Safety* 141 (2015): 5–9. *https://doi.org/10.1016/j.ress.2015.03.018*.

66. Zhu, Jiamin, Hongjiang Lv, and Yan Feng. "The Effect of Psychological Safety on Innovation Behavior: A Meta-Analysis." In *Proceedings of the 2022 7th International Conference on Financial Innovation and Economic Development*. Atlantis Press, 2022. *https://doi.org/10.2991/aebmr.k.220307.503*.

Index

Symbols

3 rules of distributed systems, The Three Golden Rules of Distributed Systems-Resource pools are not infinite, Distributed Systems

4 concepts for resilience, The Four Concepts for Resilience-Sustained Adaptability, Resilience Engineering

“5 whys”, The five whys

A

Abadi, Daniel, PACELC, Record Shop and Ticketing Examples Revisited

“The Acceleration Whiplash” (Faros AI), AI and the Rise of Cognitive Debt

accord-based backpressure, Accord-Based Backpressure, For client-only or accord-based backpressure, Glossary

action-oriented PIRs, Action-Oriented

active-active setup, Active-Active, Cassandra

active-passive setup, Active-Passive

adaptability, Sustained Adaptability

adaptation, Constant Learning and Adaptation (see also learning and improving)

AdvertCorp example, Case Study: AdvertCorp-A Confluence of Events, Case study: AdvertCorp

AI (see artificial intelligence)

"Alarm Design: From Nuclear Power to WebOps" (Shorrock), Toward Better Alerting

alarms, Alert Fatigue

alert fatigue, Alert Fatigue

alerts, Alerts-Separation of Responsibility Considered Harmful?, Glossary

alignment, Better alignment

Allspaw, John, Not Linear, Right People, Right Place

anomaly detection, Automated Anomaly Detection

architectures

- cellular, Glossary
- ecommerce, Example: Online Record Shop-It's Not All or Nothing
- microservice, What Is a Distributed System?

artificial intelligence (AI)

- backlash against, External Factors
- challenges of, AI Challenges-Yet Another Financial Bubble?
- cognitive debt created by, Distributed Systems Aren't Just Computers, AI and the Rise of Cognitive Debt
- handling uncertainty, How to Handle Uncertainty-Cost Transparency
- LLM- versus ML-based, Resiliency and AI, Automated Anomaly Detection
- and observability, Observability to the Rescue of AI?
- resiliency of AI-powered software, Resiliency of AI-Powered Software-Observability, Resilience and AI

software delivery and operations with, Using AI for Software Delivery and Operations-Incident Response

and writing, Writing Process and Use of AI

at least once delivery guarantees, At Least Once

at most once delivery guarantees, At Most Once

auditing (see monitoring)

automated anomaly detection, Automated Anomaly Detection

availability, Maintaining Availability, CAP Theorem

availability zones (AZs), Cloud Region, Data Center, or Site?, Context, Glossary

AWS Architecture Blog, Further Reading

AWS US-EAST-1 Outage (2025), Case Study: AWS US-EAST-1 Outage, October 2025-Impact

B

backoffs, Delays Between Retries

backpressure pattern

accord-based backpressure, Accord-Based Backpressure, For client-only or accord-based backpressure, Glossary

client-only backpressure, Client-Only Backpressure, For client-only or accord-based backpressure, Glossary

definition of, Glossary

handling too much work with, Reduce the Work Being Sent (Backpressure)

overview of, Backpressure

backup and restore pattern, Pattern: Backup and Restore-When to use it, Pattern Index

Beyer, Betsy, The “Golden” Signals, Separation of Responsibility Considered Harmful?

blackhole routing, Degradation functionality and blackhole routing, Glossary

blame-free environments, Learning, Not Blame

blast radius, reducing, Multiple networks, Summary

boom and bust cycle, Boom and bust, Boom and Bust Cycle

Border Gateway Protocol (BGP), Fault tolerance, Glossary

broad thinking, Thinking broadly

broadcast communication, Point-to-Point Versus Broadcast

brokers (see message brokers)

Brooker, Marc, Fault Injection and Testing Timeouts, Further Reading, Variations, It’s Not All or Nothing

bulkhead pattern, Pattern: Bulkhead, Pattern Index, Glossary

bureaucratic culture, Culture

C

cache stampede, Glossary

cache-aside pattern, Example: ShopCorp Catalog Cache Failure

call chains, Timeouts and Call Chains

CAP theorem, CAP Theorem-It’s Not All or Nothing, Glossary

case studies/examples

backups, When to use it

cache failure, Example: ShopCorp Catalog Cache Failure-Disable the service

CAP theorem, Example: Online Record Shop-It's Not All or Nothing

cell-based architecture, Case studies

circuit breaker pattern, Case study: AdvertCorp

critical standby systems, How It Works

DDoS attacks, Examples-Web application firewalls and content delivery networks

degradation functionality, Degradation functionality and blackhole routing

Little's Law, A (Contrived) Real-World Example-Limitations of Little's Law

load balancing, Example: Pet Service-Reduce failure rate

mismatched capacities, Mismatched Capacities-Improve communication

PACELC theorem, Record Shop and Ticketing Examples Revisited

progressive collapse, Real-World Examples-Impact

sociotechnical systems, The Therac-25 Radiation Machine

streamlining incident response, Right People, Right Place

timeouts, Case Study: AdvertCorp-A Confluence of Events

Cassandra, Cassandra

CDN (see content delivery networks)

cell-based architecture pattern, Pattern: Cell-Based Architecture-When to Use It, Pattern Index

cellular architecture, Glossary

chaos engineering, Chaos Engineering, Glossary

Chaos Monkey, Chaos Engineering

chapter overviews, Navigating the Book-Part III: People, Process, and Culture

circuit breaker pattern

- application of, How It Works-When to Use It
- definition of, Pattern: Circuit Breaker, Glossary
- description of, Pattern Index
- handling too much work with, Reduce the Work Being Sent (Backpressure)

client idempotency key pattern

- application of, How it works
- definition of, Pattern: Client Idempotency Key
- making operations idempotent with, How It Works
- using with server-side fingerprinting, Using Client Idempotency Keys Together with Server-Side Fingerprinting

client-only backpressure, Client-Only Backpressure, For client-only or accord-based backpressure, Glossary

client-side load balancing pattern, Pattern: Client-Side Load Balancing-When to use it, Pattern Index, Glossary

clock drift, When to Use It, Glossary

clock skew, When to Use It

cloud regions, Cloud Region, Data Center, or Site?, Glossary

Cloudflare, Examples

CockroachDB, Relational databases

Code Spaces example, When to use it

cognitive debt, Distributed Systems Aren't Just Computers, AI and the Rise of Cognitive Debt

column-oriented databases, Cassandra

comments and questions, How to Contact Us

communication

- during incident management and response, Communication
- improving, Improve communication, Message Brokers
- point-to-point versus broadcast, Point-to-Point Versus Broadcast

components

- challenges of introducing new, Progressive Collapse
- inevitability of failure of, Failure Is Inevitable
- strengthening individual, Strengthen Individual Components

concert ticketing example, Example: Concert Ticketing-In the real world, Record Shop and Ticketing Examples Revisited, Little's Law and Ticketing Systems

concurrency, How Many Requests Can You Handle?

concurrent duplicate requests, Handling concurrent duplicate requests

confirmation of processing, Guaranteed Delivery Versus Guaranteed Processing

conflict-free replicating data type (CRDT), Message brokers, Glossary

connection failures, Sometimes you can't reach the thing you want to talk to

connection pools, Page Load Times

connection timeouts, Setting a Timeout

consistency, CAP Theorem, Latency Versus Consistency

Consul, Consul, etcd, et al

consumer groups, Consumer Groups

consumer-managed inbox pattern, Pattern: Consumer-Managed Inbox-When to use it, Pattern Index, Glossary

consumers, Message Broker Basics, Glossary

content delivery networks (CDNs), Web application firewalls and content delivery networks, Glossary

Conway's Law, The Hexagonal Model for Sociotechnical Systems

coordination, Coordination

corporate culture, Thinking broadly

correctness, of AI-powered software, Correctness

costs

    of AI-powered software, Cost

    transparency in, Cost Transparency

CRDT (see conflict-free replicating data type)

critical standby system pattern, Pattern: Critical Standby System-When to Use It, Pattern Index, Glossary

critical thinking, Architectural Katas

cross-site active-active pattern, Active-Active, Pattern Index

culture

- bureaucratic, Culture
- of constant learning, Constant Learning and Adaptation
- horizontal, Better alignment
- of psychological safety, Psychological Safety-It's Not Universal

D

Dai, Ting, Other Timeout Considerations, Missing Timeouts

dashboards, static, Slicing and Dicing

data

- gathering for post-incident reviews, Gather Data
- replicating, Multisite Data Replication-Further Reading
- synchronizing, Synchronizing data

data centers, Cloud Region, Data Center, or Site?, External Factors

"Data Engineering Design Patterns" (Konieczny), Further Reading

databases

- column-oriented, Cassandra
- distributed relational databases, Relational databases
- relational, Relational databases

vector, Vector databases

DDoS (see distributed denial of service attacks)

dead letter channel pattern, Pattern: Dead Letter Channel-When to use it, Pattern Index, Glossary

deadline propagation, How It Works

declarative programming, Using Client Idempotency Keys Together with Server-Side Fingerprinting

degradation functionality, Degradation functionality and blackhole routing, Summary

delegation, Delegation

deliberate practice, Naive, Purposeful, and Deliberate Practice, Glossary

delivery guarantees, Delivery Guarantees, Types of Delivery Guarantees-Guaranteed Delivery Versus Guaranteed Processing

denial of service (DoS) attacks, Customer-Driven Denial of Service

“Designing Data-Intensive Applications”, Further Reading

desired state management pattern

application of, How It Works

definition of, Pattern: Desired State Management, Glossary

description of, Pattern Index

tools for, Use desired state management

determinism, Observability to the Rescue of AI?, Nondeterminism

diagrams for visualizing root causes, Visualizing root causes

Digital Operational Resilience Act (DORA), External Factors

DiRT (Disaster Recovery Test) exercises, Game Days

disaster recovery (DR)

- backup and restore pattern, Pattern: Backup and Restore-When to use it
- definition of, Disaster Recovery, Glossary
- determining requirements for, How Good Does Your DR Need to Be?-DR Objectives and SLOs
- highlights of, Disaster Recovery
- infrastructure availability, Infrastructure Availability
- multisite active-active pattern, Pattern: Multisite Active-Active-When to use it
- multisite data replication, Multisite Data Replication-Further Reading
- patterns for, Disaster Recovery Patterns
- pilot light failover pattern, Pattern: Pilot Light Failover-When to use it
- terminology used, Cloud Region, Data Center, or Site?
- testing, Testing DR
- warm failover pattern, Pattern: Warm Failover-When to use it

Disaster Recovery Test (DiRT) exercises, Game Days

distribute across failure planes pattern, Pattern: Distribute Across Failure Planes-When to Use It, Pattern Index

distributed denial of service (DDoS) attacks, Distributed Denial of Service Attacks-Web application firewalls and content delivery networks, Glossary

distributed relational databases, Relational databases

distributed systems

challenges of, Preface

definition of, Glossary

golden rules of, The Three Golden Rules of Distributed Systems, Distributed Systems

inevitability of failure in, Failure Is Inevitable

learning goals, What You Will Learn

potential failure of, How Distributed Systems Can Fail (Us)

role of humans in, Distributed Systems Aren't Just Computers

role of resiliency in, What Is a Distributed System?-What Is a Distributed System?

separating services on different networks, Multiple networks

Dolfing, Henrico, Example: Healthcare.gov

Domain Name System (DNS), Domain Name System (DNS), Finding server endpoints, Glossary

DORA (Digital Operational Resilience Act), External Factors

DORA State of AI-Assisted Software Development Report (DORA), AI and the Rise of Cognitive Debt

DoS (denial of service) attacks, Customer-Driven Denial of Service

draft report, sharing with responders, Share a Draft

duplicate requests, Handling concurrent duplicate requests

durability, Maintaining Durability

Dvorak, Dudley J., Right People, Right Place

dynamic provisioning, Dynamically Provision More Resources

dynamic scaling, Scaling your way out of trouble

dynamic service discovery pattern, Pattern: Dynamic Service Discovery-When to use it, Pattern Index

DynamoDB, What Happened

E

ecommerce systems, Example: Online Record Shop-It's Not All or Nothing, Record Shop and Ticketing Examples Revisited, Little's Law and Ticketing Systems-Limitations of Little's Law

Engineering Equipment and Materials Users Association (EEMUA), Toward Better Alerting

ensemble programming, Collaborative Working

"Enterprise Integration Patterns" (Hohpe), Queues Versus Topics

error budget, Error Budgets, Glossary

error codes, A Better Type of Error

error rate, Error Rates, Glossary

escalation protocols, Escalation

"Establishing SRE Foundations" (Ukis), Separation of Responsibility Considered Harmful?

etcd, Consul, etcd, et al

"Evals for AI Engineers" (Shankar), Model Changes

eventual consistency, Glossary

exactly once delivery guarantees, Exactly Once

examples (see case studies/examples)

exponential backoffs, Exponential Backoff

extensibility, Graceful Extensibility, Slicing and Dicing, Not Linear, Glossary

extreme programming, Better alignment

F

failure (see also machine failure)

- cache collapse, Cache Collapse-Disable the service
- catastrophic, Disaster Recovery, Real-World Examples (see also disaster recovery; progressive collapse)
- communication failures, Improve communication
- connection failures, Sometimes you can’t reach the thing you want to talk to
- inevitability of, Failure Is Inevitable
- instance failure, Handling instance failure, In Distributed Systems
- load balancer domino, Load Balancer Domino-Reduce failure rate
- in message sending, Failure in message sending (see also message brokers)
- mismatched capacities, Mismatched Capacities-Improve communication
- overly simplistic approach to, A Confluence of Events, Thinking broadly
- partial failure with circuit breakers, Partial failure
- popularity surges, Roaring Success-Better alignment
- reducing rate of, Reduce failure rate

failure planes, Pattern: Distribute Across Failure Planes-When to Use It

fall over, Just Let It Fall Over

fast rollback, Implement Fast Rollback

fault injection, Fault Injection and Testing Timeouts

fault tolerance, How Resilient Do You Need to Be?, Fault tolerance

fingerprints (see server-side request fingerprinting pattern)

fishbone diagrams (see Ishikawa diagrams)

Fitch, Dennis E., Right People, Right Place

“five whys”, The five whys

fleeting messages, Fleeting Messages

“Four Concepts for Resilience and the Implications for the Future of Resilience Engineering” (Woods), The Four Concepts for Resilience

FRAM (Functional Resonance Analysis Method), The Hexagonal Model for Sociotechnical Systems

Francis Scott Key Bridge (Baltimore), Real-World Examples

“From Safety-I to Safety-II” (Hollnagel), From Safety Management to Resilience Engineering

“Full Stack Testing” (Mohan), Model Changes

Functional Resonance Analysis Method (FRAM), The Hexagonal Model for Sociotechnical Systems

G

game day exercises, Game Days

goals, Goals, Incident Management and Post-Incident Response, Learning, Not Blame

“golden” signals for distributed systems, The “Golden” Signals-Correlation Is Key

Google Pub/Sub, Exactly Once

graceful degradation pattern, Degradation functionality and blackhole routing, How it works, Pattern: Graceful Degradation-When to Use It, Pattern Index, Glossary

graceful extensibility, Graceful Extensibility, Slicing and Dicing, Not Linear, Glossary

guaranteed processing, Guaranteed Delivery Versus Guaranteed Processing

Guardian example, Degradation functionality and blackhole routing, How It Works

H

hallucinations, Correctness

hardcoded IP resolution pattern, Pattern: Hardcoded IP Resolution, Pattern Index, Glossary

harvest and yield model, It’s Not All or Nothing

Haynes, Alfred C., Right People, Right Place

hazards, reducing, Reduce Hazards

Healthcare.gov, Example: Healthcare.gov

herringbone diagrams (see Ishikawa diagrams)

hexagonal model for sociotechnical systems, The Hexagonal Model for Sociotechnical Systems-External Factors, Thinking broadly

high availability, How Resilient Do You Need to Be?

Hildago, Alex, Separation of Responsibility Considered Harmful?

Hollnagel, Erik, From Safety Management to Resilience Engineering, The Hexagonal Model for Sociotechnical Systems

Honeycomb, Traces, Correlation Is Key, Make Systems Easily Observable

horizontal culture, Better alignment

“How Generative and Agentic AI Shift Concern from Technical Debt to Cognitive Debt” (Storey), Distributed Systems Aren’t Just Computers, AI and the Rise of Cognitive Debt

HTTP status codes, A Better Type of Error

human error, Learning, Not Blame

Husain, Hamel, Model Changes

I

idempotency, Exactly Once (see also client idempotency key pattern)

idempotency keys, Pattern Index, Glossary

idempotent operations (see also retries)

- and change of state, What happens when state changes between attempts
- creating, How It Works
- definition of, Pattern: Idempotent Operation, Glossary
- description of, Pattern Index
- handling concurrent duplicate requests, Handling concurrent duplicate requests

highlights of, Retries and Idempotency

patterns for, Making Operations Idempotent-Testing Retries and Idempotency

responding to repeated attempts, How to respond to repeated attempts

testing, Testing Retries and Idempotency

when to use, When to Use It

“Implementing Service Level Objectives” (Hidalgo), Separation of Responsibility Considered Harmful?

inbox pattern (see consumer-managed inbox pattern)

incident commander, Incident Roles, Glossary

incident management and response

definition of incident, What Do You Mean, “Incident”?

goals of, Incident Management and Post-Incident Response, Learning, Not Blame

highlights of, Incident Management

incident roles, Incident Roles

key activities in, Key Activities in Incident Response-Saturation

learning from incidents, Learning from Incidents

non-linearity of, Not Linear-Not Linear

post-incident recovery, Post-Incident Recovery

post-incident review (PIR), Post-Incident Review-Share the Final Results

streamlining for ease of, Making Incident Response Easier-Implement Fast Rollback

tools for incident response, Incident Response

incident response plan, Key Activities in Incident Response

incident.io, Toward Better Alerting, Testing Observability, Incident Response, When to Use It, Game Days

independent thinking, AI and the Rise of Cognitive Debt

inference, Automated Anomaly Detection

infrastructure availability, Infrastructure Availability

“Infrastructure as Code” (Morris), Use desired state management

instance failure, Handling instance failure, In Distributed Systems

interconnection, reducing, Reduce Interconnection-Reduce Interconnection

invite codes, Limiting sign-ups

Ishikawa diagrams, Visualizing root causes

isolation, increasing, Increasing isolation

J

Java Virtual Machine (JVM), Improve communication

jitter, Jitter

K

Kafka, When to Use It, How Brokers Work (in a Nutshell), Fleeting Messages, Exactly Once, When to use it

Kerth, Norman L., Learning, Not Blame

Kleppmann, Martin, Further Reading

Konieczny, Bartosz, Further Reading

Kubernetes, When to Use It

L

latency, Latency, Tail Latencies-Tight Latency Bounds, Impact of Latency, Latency Versus Consistency, Glossary

leaky bucket rate limiting pattern

- application of, How It Works-When to Use It
- definition of, Pattern: Leaky Bucket Rate Limiting, Glossary
- description of, Pattern Index
- handling too much work with, Reduce the Work Being Sent (Backpressure)

learning and improving

- AI and cognitive debt, AI and the Rise of Cognitive Debt
- architectural katas, Architectural Katas
- building learning into day jobs, Build Learning into the Day Job
- chaos engineering, Chaos Engineering
- collaborative working, Collaborative Working
- game day exercises, Game Days
- highlights of, Constant Learning
- learning from incidents, Learning from Incidents
- naive, purposeful, and deliberate practice, Naive, Purposeful, and Deliberate Practice
- online learning resources, Online Learning Resources
- promoting a culture of learning, Constant Learning and Adaptation

psychological safety and, Psychological Safety-It’s Not Universal

“Learning Chaos Engineering” (Miles), Game Days

learning goals, What You Will Learn

Leveson, Nancy, The Hexagonal Model for Sociotechnical Systems, Thinking broadly

linear thinking, Visualizing root causes

Little's Law, Little’s Law-Limitations of Little’s Law, Glossary

load balancing

challenges of, Load Balancer Domino-Reduce failure rate

client-side load balancing pattern, Pattern: Client-Side Load Balancing-When to use it, Glossary

definition of, Glossary

versus load sharing, Load Balancing

for scaling and resiliency, Load Balancing

server-side load balancing pattern, Pattern: Server-Side Load Balancing-When to use it

load sharing, Load Balancing, Glossary

load shedding pattern

application of, How It Works-When to Use It

and DDoS attacks, Rate limiting

definition of, Pattern: Load Shedding, Glossary

description of, Pattern Index

handling too much work with, Throw Away Some of the Work (Load Shedding)

and Kubernetes, When to Use It

versus quota limits and backpressure, Accord-Based Backpressure

load testing, Better load testing, More realistic load testing

logs, Logs

luck, Thinking broadly

Luu, Dan, Share the Final Results

M

machine failure

desired state management pattern, Pattern: Desired State Management-When to Use It

handling, Handling Machine Failure

highlights of, Machine Failure

load balancing for scaling and resiliency, Load Balancing

service discovery, Service Discovery-When to use it

stateful versus stateless processing, Stateful Versus Stateless Processing-General guidance

testing, Testing Machine Failure

Majors, Charity, Three Pillars Not Sufficient?, Separation of Responsibility Considered Harmful?

message brokers

basics of, Message Broker Basics-Consumer Groups

benefits of using, Why Use a Message Broker?-Scaling

consumer-managed inbox pattern, Pattern: Consumer-Managed Inbox-When to use it

data replication, Message brokers

dead letter channel pattern, Pattern: Dead Letter Channel-When to use it

definition of, Glossary

delivery guarantees, Types of Delivery Guarantees-Guaranteed Delivery Versus Guaranteed Processing

highlights of, Message Brokers

how brokers work, How Brokers Work (in a Nutshell)-Fleeting Messages

producer heartbeat message pattern, Pattern: Producer Heartbeat Message

self-hosted versus fully managed, Self-Hosted Versus Fully Managed Brokers

timeouts and retries, Scaling

transactional outbox pattern, Pattern: Transactional Outbox-When to use it

useful patterns for, Useful Patterns

message channels, Queues Versus Topics, Glossary

message hospital pattern, When to use it

metrics, Metrics

microservice architecture, What Is a Distributed System?

Microsoft Azure, Examples

Miles, Russ, Game Days

mob programming, Collaborative Working

Mohan, Gayathri, Model Changes

monitoring

- auditing messages, Retaining sent messages
- connections versus requests, Choosing what to track
- “golden” signals to capture, The “Golden” Signals-Correlation Is Key
- versus observability, Observability Versus Monitoring

Monzo example, How It Works

Morris, Kief, Use desired state management

multimodel/multivendor approach, Multimodel, Multivendor

multisite active-active pattern, Pattern: Multisite Active-Active-When to use it

multisite data replication, Multisite Data Replication-Further Reading

N

naive practice, Naive, Purposeful, and Deliberate Practice, Glossary

Nash, Courtney, Key Activities in Incident Response

Network Time Protocol (NTP), When to Use It, Glossary

networking

- failures of in distributed systems, Sometimes you can’t reach the thing you want to talk to

separating services on different networks, Multiple networks

neurodivergence, It's Not Universal

Neward, Ted, Architectural Katas

non-linear workflows, Not Linear-Not Linear

nondeterminism, Observability to the Rescue of AI?, Nondeterminism

normalization of deviance, The Sociotechnical Systems

NTP (Network Time Protocol), When to Use It, Glossary

Nubank example, Case studies

Nygard, Mike, Pattern: Circuit Breaker

O

observability

3 pillars of, The Three Pillars of Observability?-Three Pillars Not Sufficient?

and AI, Observability to the Rescue of AI?, Observability

alerts, Alerts-Separation of Responsibility Considered Harmful?

definition of, Glossary

error budgets, Error Budgets

"golden" signals to capture, The "Golden" Signals-Correlation Is Key

highlights of, Observability

versus monitoring, Observability Versus Monitoring

OpenTelemetry standards, OpenTelemetry

role in resilience, Fundamentals of Observability

SLAs, SLOs, and SLIs, SLAs and SLOs and SLIs, Oh My!-Service Level Indicators

streamlining incident response, Make Systems Easily Observable

testing, Testing Observability

"Observability Engineering" (Majors), Three Pillars Not Sufficient?, Separation of Responsibility Considered Harmful?

off-site backups, When to use it

online learning resources, Online Learning Resources

online record shop example, Example: Online Record Shop-In the real world, Record Shop and Ticketing Examples Revisited

open standards, OpenTelemetry

OpenTelemetry (OTel), OpenTelemetry, Glossary

OpenTelemetry Protocol (OTLP), Glossary

overload, Overload

ownership, clear lines of, Clear Lines of Ownership

P

PACELC theorem, PACELC-Latency Versus Consistency and Resiliency, Glossary

page loading, Page Load Times

pair programming, Collaborative Working

partition tolerance, CAP Theorem

partitions, CAP Theorem, PACELC, Glossary

patching, Keeping things patched

patterns

backup and restore, Pattern: Backup and Restore-When to use it, Pattern Index

bulkhead, Pattern: Bulkhead, Pattern Index

cache-aside, Example: ShopCorp Catalog Cache Failure

cell-based architecture, Pattern: Cell-Based Architecture-When to Use It, Pattern Index

circuit breaker, Pattern: Circuit Breaker-When to Use It, Pattern Index

client idempotency key, How It Works, Pattern: Client Idempotency Key, Using Client Idempotency Keys Together with Server-Side Fingerprinting

client-side load balancing, Pattern Index

consumer-managed inbox, Pattern: Consumer-Managed Inbox-When to use it, Pattern Index

critical standby system, Pattern: Critical Standby System-When to Use It, Pattern Index

cross-site active-active, Active-Active, Pattern Index

dead letter channel, Pattern: Dead Letter Channel-When to use it, Pattern Index

desired state management, Use desired state management, Pattern: Desired State Management-When to Use It, Pattern Index, Glossary

distribute across failure planes, Pattern: Distribute Across Failure Planes-When to Use It, Pattern Index

dynamic service discovery, Pattern Index

graceful degradation, How it works, Pattern: Graceful Degradation-When to Use It, Pattern Index, Glossary

hardcoded IP resolution, Pattern Index, Glossary

idempotent operation, Pattern: Idempotent Operation, Pattern Index, Glossary

idempotent operations, Pattern Index

leaky bucket rate limiting, Reduce the Work Being Sent (Backpressure), Pattern Index, Glossary

load shedding, Rate limiting, Pattern: Load Shedding, Pattern Index, Glossary

message hospital, When to use it

multisite active-active, Pattern: Multisite Active-Active-When to use it

pilot light failover, Pattern: Pilot Light Failover-When to use it, Pattern Index, Glossary

producer heartbeat message, Pattern: Producer Heartbeat Message, Pattern Index, Glossary

request fingerprinting, How It Works

retry remote operation, Pattern: Retry Remote Operation, Pattern Index

server-side load balancing, Pattern: Server-Side Load Balancing-When to use it, Pattern Index

server-side request fingerprinting, Pattern: Server-Side Request Fingerprinting-Where to use it, Pattern Index

timeout, Pattern: Timeout, Pattern Index

timeout budget propagation, Pattern: Timeout Budget Propagation, Dynamic Retry Limit, Pattern Index, Glossary

transactional outbox, Pattern: Transactional Outbox-When to use it, Pattern Index, Glossary

warm failover, Pattern: Warm Failover-When to use it, Pattern Index

Pet service example, Example: Pet Service-Reduce failure rate

pilot light failover pattern, Pattern: Pilot Light Failover-When to use it, Pattern Index, Glossary

point-to-point communication, Point-to-Point Versus Broadcast

popularity surges, Roaring Success-Better alignment

post-incident reviews (PIRs), Post-Incident Review-Share the Final Results, Glossary

postmortems, Post-Incident Review, Glossary

Precision Time Protocol (PTP), When to Use It, Glossary

prerequisites required, Prerequisite Knowledge

producer heartbeat message pattern, Pattern: Producer Heartbeat Message, Pattern Index, Glossary

producers, Message Broker Basics, Glossary

programming models

declarative programming, Using Client Idempotency Keys Together with Server-Side Fingerprinting

ensemble programming, Collaborative Working

extreme programming, Better alignment

pair programming, Collaborative Working

“Programming as Theory Building” (Naur), Distributed Systems Aren’t Just Computers

progressive collapse
bulkhead pattern, Pattern: Bulkhead
case study, Case Study: AWS US-EAST-1 Outage, October 2025-Impact
cell-based architecture pattern, Pattern: Cell-Based Architecture-When to Use It
critical standby system pattern, Pattern: Critical Standby System-When to Use It
definition of, What Is Progressive Collapse?, Glossary
distribute across failure planes pattern, Pattern: Distribute Across Failure Planes-When to Use It
in distributed systems, In Distributed Systems
graceful degradation pattern, Pattern: Graceful Degradation-When to Use It
highlights of, Progressive Collapse
mitigating, Mitigating Progressive Collapse-Reduce Interconnection
patterns for avoiding, Patterns for Avoiding Progressive Collapse
real-world examples of, Real-World Examples
“Project Retrospectives: A Handbook for Team Review” (Kerth), Learning, Not Blame
psychological safety, Psychological Safety-It’s Not Universal, Glossary
PTP (see Precision Time Protocol)
public cloud, Cloud Region, Data Center, or Site?
public cloud global tables, Public cloud global tables

purposeful practice, Naive, Purposeful, and Deliberate Practice, Glossary

Q

questions and comments, How to Contact Us

queues, Queue Up the Work, Queues Versus Topics, A (Contrived) Real-World Example-Limitations of Little's Law

quota limits, Accord-Based Backpressure

R

RabbitMQ, How Brokers Work (in a Nutshell)

Raft consensus algorithm, Consul, etcd, et al

ramp-ups, gradual, Gradual ramp-up

rate limiting

- avoiding hardcoding of, Config, Not Code
- comparing circuit breaker and token bucket, Simulating the Use of a Circuit Breaker and a Token/Leaky Bucket
- DDoS attacks and, Rate limiting
- definition of, Glossary
- handling too much work, Ways to Handle Having Too Much Work-Dynamically Provision More Resources
- highlights of, Rate Limiting
- other options, Stopping the Load in the First Place
- reducing versus stopping traffic, Reducing Versus Stopping Traffic-A Delicate Balance
- reducing work with, Reduce work, Rate limiting

RateLimiter, Existing implementations

rebound, Rebound, Glossary

Records, William R., Right People, Right Place

recovery point objective (RPO), Recovery Point Objective, Glossary

recovery time objective (RTO), Recovery Time Objective, Glossary

recovery time, improving, Improve recovery time, Summary

redundant computing resources, Reduce failure rate, Summary

relational databases, Relational databases

“Release It!” (Nygard), Pattern: Circuit Breaker

request fingerprinting, How It Works

request timeouts, Setting a Timeout

requests

- definition of, Timeouts
- determining maximum request number, How Many Requests Can You Handle?
- handling concurrent duplicate requests, Handling concurrent duplicate requests
- monitoring connections versus requests, Choosing what to track
- subject access requests (SARs), Handling inconsistent load from work, A Delicate Balance

requirements, setting clear, Summary

resilience

- attaining desired amount of, How Resilient Do You Need to Be?

definition of, Glossary

versus fault tolerance and high availability, How Resilient Do You Need to Be?

four concepts for, Resilience Engineering

four concepts of, The Four Concepts for Resilience-Sustained Adaptability

role of observability in, Fundamentals of Observability

resilience engineering, From Safety Management to Resilience Engineering, Resilience Engineering, Glossary

resiliency

in cell-based architecture, Cell resiliency

versus consistency and latency, Latency Versus Consistency and Resiliency

definition of, Defining Resiliency-Social

importance of, Why Resilience Matters

of AI-powered software, Resiliency of AI-Powered Software-Observability

role of distributed systems in, What Is a Distributed System?-What Is a Distributed System?

resources

dynamic provisioning of, Dynamically Provision More Resources

monitoring saturation, Saturation

use of in distributed systems, Resource pools are not infinite

responders, Glossary

response times, analyzing existing, Analyzing Existing Response Times-Tight Latency Bounds

retries (see also idempotent operations)

- adaptive retry mechanism, Variations
- application of, How It Works
- appropriate versus inappropriate, Should You Always Retry?
- and client libraries, When to Use It
- definition of, Glossary
- delays between, Delays Between Retries-Jitter
- determining correct number of, How Many Retries Are Appropriate?-Dynamic Retry Limit
- differentiating errors, A Better Type of Error
- highlights of, Retries and Idempotency
- and message brokers, Scaling
- retry remote operation pattern, Pattern: Retry Remote Operation, Pattern Index
- safety of, Is Retrying Safe?
- testing, Testing Retries and Idempotency

retrospective prime directive, Learning, Not Blame

Riccomini, Chris, Further Reading

robustness, Robustness, Glossary

roles, defining for incident response, Incident Roles

rollback, implementing fast, Implement Fast Rollback

Ronan Point residential tower (London), Real-World Examples

root causes, Root Cause(s)?-Visualizing root causes

Rose, Sam, Further Reading

round-robin DNS, Finding server endpoints

RPO (see recovery point objective)

RTO (see recovery time objective)

S

safety management, From Safety Management to Resilience Engineering

safety-critical systems, The Hexagonal Model for Sociotechnical Systems

Safety-I, Glossary

Safety-II, Glossary

safing, Not Linear, Glossary

sandwich shop example, A (Contrived) Real-World Example

SARs (subject access requests), Handling inconsistent load from work, A Delicate Balance

saturation, Saturation, Saturation, Glossary

scaling, Mismatched Capacities-Improve communication, Scaling your way out of trouble, Scale up, Scaling

server endpoints, Finding server endpoints

server-side load balancing pattern, Pattern: Server-Side Load Balancing-When to use it, Pattern Index

server-side request fingerprinting pattern, Pattern: Server-Side Request Fingerprinting-Testing Retries and Idempotency, Pattern Index, Glossary

service discovery

dynamic service discovery pattern, Pattern: Dynamic Service Discovery-When to use it

hardcoded IP resolution pattern, Pattern: Hardcoded IP Resolution

service level agreements (SLAs), Service Level Agreements, Glossary

service level indicators (SLIs), Service Level Indicators, DR Objectives and SLOs, Glossary

service level objectives (SLOs), Service Level Objectives, DR Objectives and SLOs, Not Linear, Glossary

service meshes, Using a service mesh, Glossary

Shankar, Shreya, Model Changes

shards, Case studies

ShopCorp example, Example: ShopCorp Catalog Cache Failure-Disable the service

Shorrock, Steven, Toward Better Alerting-Separation of Responsibility Considered Harmful?

sign-ups, limiting, Limiting sign-ups

simplistic thinking, A Confluence of Events, Thinking broadly

“Site Reliability Engineering” (Beyer), The “Golden” Signals, Separation of Responsibility Considered Harmful?

sites, Cloud Region, Data Center, or Site?

SLA (see service level agreements)

SLI (see service level indicators)

SLO (see service level objectives)

SMART framework, Action-Oriented

sociotechnical systems, Distributed Systems Aren't Just Computers, The Sociotechnical Systems-The Therac-25 Radiation Machine, Thinking broadly, Sociotechnical Systems, Glossary

software delivery and operations, Using AI for Software Delivery and Operations-Incident Response

Spanner, Relational databases

spans, Traces

Stability Sim, Fault Injection and Testing Timeouts, Challenges

STAMP (Systems-Theoretic Accident Model and Processes), The Hexagonal Model for Sociotechnical Systems

state, change of during idempotent operations, What happens when state changes between attempts

stateful processing, Stateful Versus Stateless Processing-General guidance, Glossary

stateless processing, Stateful Versus Stateless Processing, Glossary

static dashboards, Slicing and Dicing

status codes, A Better Type of Error

sticky sessions, Work routing

Storey, Margaret-Anne, Distributed Systems Aren't Just Computers, AI and the Rise of Cognitive Debt

strangler fig pattern, Case Study: AdvertCorp

subject access requests (SARs), Handling inconsistent load from work, A Delicate Balance

sustained adaptability, Sustained Adaptability, Slicing and Dicing, Glossary

system usage, throttling, Summary, Queue Up the Work

Systems-Theoretic Accident Model and Processes (STAMP), The Hexagonal Model for Sociotechnical Systems

T

tail latencies, Tail Latencies-Tight Latency Bounds

target audience, Who This Book Is For

temporal decoupling, Temporal Decoupling

theorems

- CAP, CAP Theorem-It's Not All or Nothing
- highlights of, Formal Methods
- Little's Law, Little's Law-Limitations of Little's Law
- PACELC, PACELC-Latency Versus Consistency and Resiliency
- understanding trade-offs, Using Formal Theorems to Understand Trade-Offs, It's Not All or Nothing, Summary

Therac-25 radiation therapy machine, The Therac-25 Radiation Machine, Thinking broadly

thinking styles

- broad thinking, Thinking broadly
- critical thinking, Architectural Katas
- independent thinking, AI and the Rise of Cognitive Debt
- linear thinking, Visualizing root causes
- simplistic thinking, A Confluence of Events

threads, Improve communication

thresholds, setting, Setting thresholds

throttling, Summary, Queue Up the Work

thundering herds

- cache collapse, Cache Collapse-Disable the service
- DDoS attacks, Distributed Denial of Service Attacks-Web application firewalls and content delivery networks
- definition of, What Is a Thundering Herd?, Glossary
- highlights of, Thundering Herds
- load balancing, Load Balancer Domino-Reduce failure rate
- mismatched capacities, Mismatched Capacities-Improve communication
- roaring successes, Roaring Success-Better alignment

ticket purchasing systems, In the real world, Record Shop and Ticketing Examples Revisited, Little's Law and Ticketing Systems

time to live (TTL), Domain Name System (DNS)

timeout budget propagation pattern

- application of, How It Works-When to Use It
- definition of, Pattern: Timeout Budget Propagation, Glossary
- description of, Pattern Index
- dynamic retry limits, Dynamic Retry Limit

timeouts

- analyzing existing response times, Analyzing Existing Response Times-Tight Latency Bounds

application of, How It Works-When to Use It

and call chains, Timeouts and Call Chains

case study, Case Study: AdvertCorp-A Confluence of Events

definition of, Pattern: Timeout, Glossary

determining maximum request number, How Many Requests Can You Handle?

highlights of, Timeouts

and message brokers, Scaling

missing, Missing Timeouts

need for, The Problem with Time

setting, Setting a Timeout

testing, Fault Injection and Testing Timeouts

timeout pattern, Pattern Index

timeout-related bugs, Other Timeout Considerations

and user expectations, Using User Expectations

token buckets, Variations

topics, Queues Versus Topics

traces, Traces

tracking (see monitoring)

trade-offs, understanding, Using Formal Theorems to Understand Trade-Offs, It’s Not All or Nothing, Summary

traffic, Traffic

transactional outbox pattern, Pattern: Transactional Outbox-When to use it, Pattern Index, Glossary

TTL (time to live), Domain Name System (DNS)

Turner, Clark, Thinking broadly

Twitter, Example: Twitter

U

Ukis, Vladyslav, Separation of Responsibility Considered Harmful?

uncertainty, handling, How to Handle Uncertainty-Cost Transparency

“Understanding Real-World Timeout Problems in Cloud Server Systems” (Dai), Other Timeout Considerations, Missing Timeouts

United Airlines Flight 232 example, Right People, Right Place

universally unique identifier (UUID), Glossary

Uptime Labs, Online Learning Resources

user growth, Roaring Success-Better alignment

V

vector databases, Vector databases

vendors

- instability among, Vendor Instability
- more frequent vendor analysis, More Frequent Vendor Analysis
- multivendor approach, Multimodel, Multivendor

virtual threads, Improve communication

VOID incident report database, Key Activities in Incident Response, Share the Final Results, Learning from Incidents

W

warm failover pattern, Pattern: Warm Failover-When to use it, Pattern Index, Glossary

web application firewalls (WAFs), Web application firewalls and content delivery networks

“Wheel of Misfortune”, Game Days

Woods, David, The Four Concepts for Resilience, Model Changes, Not Linear, Resilience Engineering

work

- distributing across planes, Distributing work
- handling too much, Ways to Handle Having Too Much Work-Dynamically Provision More Resources
- inconsistent loads, Handling inconsistent load from work
- prioritizing, Prioritizing work-Prioritizing work

worker pools, Managing the worker pool

Y

yield, It’s Not All or Nothing

About the Author

Sam Newman is a technologist focusing primarily in the areas of distributed systems architecture and continuous delivery—three topics that seem to overlap frequently. He provides consulting, training, and advisory services to startups and large multinational enterprises alike, drawing on his more than 25 years in IT as a developer, sysadmin, and architect. He works with companies big and small all over the world, helping solve problems with technology. Sam is the author of the best-selling *Building Microservices* and *Monolith to Microservices*, both from O'Reilly, and an experienced conference speaker.

Colophon

The animal on the cover of *Building Resilient Distributed Systems* is a European badger *(Meles meles)*.

The European badger is a powerfully built member of the weasel family found across much of Europe and parts of western Asia. It favors landscapes combining woodland or dense cover with open grassland, farmland, hedgerows, or gardens where it can forage. Adults typically measure 30–39 inches in length and weigh approximately 18–26 pounds, although weight varies seasonally and geographically. Their coarse gray coat, black underside and legs, and boldly striped black-and-white face make them distinctive.

Badgers are mainly nocturnal, living in mixed-sex social groups or clans. They usually search for food independently. They are adaptable omnivores whose diet often consists largely of earthworms, supplemented by insects, fruit, roots, and small vertebrates. They do not hibernate in the strict sense but may remain underground and become less active during cold weather. The IUCN classifies the species's population globally as Least Concern, so it is not presently endangered; however, road deaths, habitat loss, persecution, and population-control programs can cause serious local declines.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black-and-white engraving from *Natural History of Animals*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.