O'REILLY®

Software Architecture

Patterns, Antipatterns, and Pitfalls

Understanding Trade-Offs



Mark Richards, Neal Ford & Raju Gandhi

Software Architecture Patterns, Antipatterns, and Pitfalls

Understanding Trade-Offs

Mark Richards, Neal Ford, and Raju Gandhi



Software Architecture Patterns, Antipatterns, and Pitfalls

by Mark Richards, Neal Ford, and Raju Gandhi

Copyright © 2027 Neal Ford, Defmacro Software, LLC and Mark Richards. 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 Editor: Sarah Grey

Production Editor: Christopher Faucher

Copyeditor: Sonia Saruba

Proofreader: Dave Awl

Indexer: BIM Creatives, LLC

Cover Designer: Susan Brown

Cover Illustrator: Monica Kamsvaag

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

November 2026: First Edition

Revision History for the First Edition

- 2026-10-28: First Release

See *https://oreilly.com/catalog/errata.csp?isbn=9798341663923* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Software Architecture Patterns, Antipatterns, and Pitfalls*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors 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.

979-8-341-66392-3

[LSI]

Preface

Software architects constantly search for solutions to problems that arise on a regular basis, but many of those problems aren't unique — they have been solved before. When an architect solves a problem in a particularly elegant way, they document it as an architecture pattern.

Those new to software architecture or current practitioners who want to improve their craft turn to patterns books to see and understand these fundamental patterns, which this book provides. While most architecture patterns books focus on a narrow subject area, our book is designed to cover great breadth in uncovering solutions every architect should know.

How to Read This Book

This is a patterns book, which means it's fundamentally a catalog of architecture patterns. However, architects can use it in a couple of ways:

- As a catalog of architecture patterns to learn their contexts and trade-offs
- As a cross-cutting reference to help find patterns that help (or hurt) particular capabilities
- As a narrative to learn more about architecture patterns

Architecture Patterns Catalog

True to its nature, architects can use this book purely as a catalog of patterns. Because they vary so much in context, we didn't try to categorize them into sections but rather order them alphabetically.

However, architects must perform trade-off analysis as a key part of their job, and everything in software architecture suffers from trade-offs. To that

end, we provide a chapter called "Capability and Context Cross-Reference" (Chapter 2) that organizes the patterns not by name but by what capabilities they improve or degrade. Thus, if an architect seeks a communication pattern that offers high responsiveness, for example, they can go to that section ("Responsiveness") to see what patterns either improve responsiveness (making them candidates for the architect) or degrade it (things for the architect to avoid). Our consistent focus on trade-offs allows a cross-cutting comparison between patterns and their capabilities.

As a Narrative

One of the goals of our book, *Fundamentals of Software Architecture*, 2nd edition (O'Reilly, 2025), was to create a foundational guide for the architect role. Similarly, we mean this book to provide a foundational set of architectural patterns for those new to the subject. To that end, readers may want to read it serially, one pattern at a time. Our alphabetical organization helps keep the reader's interest; each next pattern may be from a different context.

Many of our patterns touch on the same core concepts. For example, several of our workflow patterns deal with compensating updates and ACID transactions. To keep from repeating those core concepts (including these two) in each pattern, we consolidated them to a final chapter on fundamental definitions (Chapter 37) and freely cross-referenced them within individual patterns.

Cross-Cutting References

We mean this book to be useful as a working guide for software architects, so we focused on comparability and searchability across useful dimensions. We have the usual table of contents and index, but we also have more ways to discover and contextualize patterns. In "Capability and Context Cross-Reference" (Chapter 2), we group the patterns by the capabilities they influence (either improved or degraded), and we also group them by *context*: because we cover a wide variety of patterns to provide more

breadth, each pattern is only applicable within one or more contexts. We cross-reference each pattern in "Patterns by Context" as to their applicable contexts.

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, and keywords.

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

> **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/software-architecture-patterns*.

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

Mark, Neal, and Raju would like to thank all the people who made this book possible. First and foremost, we would like to thank our excellent editor, Sarah Grey. Sarah's commitment to us and to the book was amazing, and we truly couldn't have written this book without her. We'd also like to thank the publishing team at O'Reilly for their endless patience and effort in creating all the custom tables, custom fonts, and special formatting we requested for this book. Finally, we want to thank all of our technical reviewers, all of whom produced great feedback in a very short period of time.

We'd like to give a shoutout to the wonderful audiences we've had throughout the world who attend our talks, workshops, and conferences, who constantly challenge us with their insightful questions and feedback, and have helped make this material relevant, and better. We are truly grateful for your engagement and support.

Acknowledgments from Mark Richards

In addition to the preceding acknowledgments, I would like to thank my lovely wife, Rebecca, whose patience and support are never-ending. I also want to thank my two coauthors, Neal and Raju, for making this long book project enjoyable and worthwhile.

Acknowledgments from Neal Ford

Thanks to Mark for letting me talk him into the (still) probably ill-advised adventure of writing two books at once, and Raju for collaborating with us despite tons of other commitments; it was a pleasure working with both of you. Mostly, however, I want to thank my immediate family, who are most impacted by my tucked-away writing stance so much of the time: Candy,

my wife of many years, and our cats since the pandemic: Amadeus, Fauci, and Linda Ruth.

Acknowledgments from Raju Gandhi

This is my second book with Mark and Neal, and I'm deeply grateful for the opportunity to work with them again. Learning from two of the best has been an incredible experience, and it has made the entire writing process both exhilarating and deeply rewarding. I also want to thank my wonderful partner Michelle, whose love and support sustain me in all of my endeavors, and our wonderful four-year-old daughter Delphine, who has often joined our calls to share her keen insights about software architecture and who makes all of this feel worthwhile. And finally, to our readers: thank you. Without you, none of this would mean nearly as much.

Chapter 1. Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls

The concept of *patterns* as a way of documenting common effective solutions within similar contexts has been popular since the publication of *Design Patterns* (Addison-Wesley, 1994), by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, often called the "Gang of Four book." The patterns perspective has proven useful across a wide variety of technical subjects, including software architecture.

Dozens of books exist covering architecture patterns; the topic is one of the most fruitful in the entire software architecture discipline. This is both encouraging and discouraging! The number of patterns is *encouraging* because it illustrates that a host of problems exist in the software architecture world and there are solutions for those problems in the form of patterns. However, it is *discouraging*, especially for new software architects, because of the sheer volume and specialization of these patterns.

Because there are so many architecture patterns, books generally specialize in particular categories. For example, Martin Fowler wrote *Patterns in Enterprise Application Architecture* (Addison-Wesley, 2002), whereas Bobby Woolf and Gregor Hohpe wrote *Enterprise Integration Patterns* (Addison-Wesley, 2003), highlighting the host of issues that arise for that context. This level of specialization is great if someone needs a specific context — but it's less useful for an overview.

That's where this book fits in. In the second edition of our book *Fundamentals of Software Architecture*, we tried to wrangle the wide variety of topics that comprise software architecture into a single place to

create a context for new architects to understand its facets. Toward that end, it covered a number of architectural styles (see "Styles Versus Patterns") that we consider fundamental and think every software architect should know.

While understanding how to choose an appropriate style is important, only a few dozen styles exist; they form the primary scaffolding of the architectural solution. Architects have created hundreds of architectural patterns over decades — how do new (and existing) architects make intelligent choices? While there are a number of specialized architecture patterns books available, no general guide covering a wide array of fundamental patterns existed until now.

First, we distinguish between an architecture *style* versus *pattern*, then between *design* and *architectural* patterns.

Styles Versus Patterns

When we first started teaching a software architecture fundamentals course, we made no distinction between *styles* and *patterns*. We treated both as contextualized solutions to problems. However, over time we realized their key distinguishing characteristic: patterns are more flexible than styles. An architecture *style* provides a name to a set of system design characteristics. *Patterns*, on the other hand, are contextualized solutions. For example, a communication pattern such as the Orchestration pattern (see Chapter 27) may apply to any distributed architecture, including event-driven and microservices.

Like a pattern, an architectural style is a contextualized solution, but it has a much narrower context than a pattern does. Patterns represent solutions with a particular context.

An architecture's style describes several different characteristics of that architecture, including its:

Component topology

An architectural style defines how components and their dependencies are organized. For example, a layered architecture organizes component layers by their technical capabilities, such as presentation, persistence, and user interface; whereas a modular monolith organizes its components around domains, such as *Catalog Checkout* and *Update Inventory*. The style determines the architect's philosophy — their underlying principles and goals — in creating the building block of the application.

Component dependencies

Architectural style dictates how components are "wired" to each other. For example, in a monolithic architecture, components may freely reference other components within the same address space. However, in distributed architectures, components must reference each other via network protocols, adding complexity and communication overhead.

Physical architecture

Architectural style also dictates the type of physical architecture: either monolithic or distributed. For example, a monolithic architecture typically has a single database that is deployed together in lockstep with the architecture, whereas an event-driven architecture may use a single or multiple databases, and a microservices architecture typically uses many databases. The philosophy of the architecture dictates how the architects partition its data.

Deployment model

A system's granularity and its deployment frequency correlate with its architectural style. Teams generally deploy monolithic architectures as a single deployment along with a single relational database. Conversely, highly agile distributed architectures, such as microservices, feature automated integration, automated provisioning, and sometimes automated deployments, and are generally deployed in pieces, with a much faster cadence.

Communication style

The architectural style also dictates how components communicate with each other. Monolithic architectures can make method calls within the same physical process, whereas distributed architectures communicate via network protocols, like REST and message queues.

Data topology

Just like its component topology, a system's data topology is usually dictated by its architectural style. Monolithic architectures tend to have a single database, whereas distributed architectures sometimes separate data — depending on the philosophy of the architectural style.

Reuse philosophy

The architectural style also dictates how architects answer the inevitable follow-up question when they answer, "It depends": "Depends on what?" Often, the architectural style's philosophy determines the answer. For example, the answer to the question, "Which is a better way to achieve reuse for this functionality — creating a shared component or duplication?" is "It depends." If it's a monolithic architecture, the answer is a *shared component*, which leverages the advantage of the monolithic nature of the code base. However, the answer is *duplication* for a microservices architecture, where the general rule of thumb is that duplication is preferable to coupling. While both have negative consequences, the underlying philosophy of microservices is to try to avoid intraservice coupling as much as possible, tipping the balance toward duplication.

Naming a style provides a concise way to describe these myriad factors. Each name captures a wealth of understood detail — that's one of the purposes of patterns in general. However, whereas a pattern captures a contextualized solution, an architectural style is more specific to

architecture, describing the architecture's topology and its assumed and default characteristics, both beneficial and detrimental.

Architecture Patterns Versus Design Patterns

Next, we distinguish between *design patterns* and *architecture patterns*. True to their name, *architecture patterns* provide one solution to a common problem within a certain *architectural* context. So, how are design and architecture patterns different? Design patterns deal with how developers implement solutions more elegantly. For example, the Strategy Design pattern provides a more readable and maintainable way of handling a long list of options than a giant `SWITCH` statement.

The book *Design Patterns* targeted the internal design characteristics of code, providing good solutions to common problems. This book provides a similar perspective: an architecture pattern provides a useful solution to a common problem that arises in software architecture. Just like design patterns, architecture patterns are implementation agnostic — not tied to a particular platform or implementation.

Both design and architecture patterns capture and name a good solution to a problem within a context. So how are architecture patterns different from design patterns?

Unlike design patterns, architecture patterns also:

Impact architectural structure

> The only real impact that design patterns have on structure is addition or subtraction of classes and relationships — after all, design patterns express design decisions in object-oriented languages. Architecture patterns, on the other hand, impact the structure of the system, at a minimum changing the relationship between components, all the way to restructuring the physical architecture.

Impact architectural capabilities

> In our various books on software architecture fundamentals, we split structural analysis in software architecture into *capabilities* versus *behavior*. The behavior of the system encompasses the problem domain, expressed in some form as requirements — the motivation for writing a software solution. The other part of structural analysis is defining and governing the *capabilities* of the system. Architects choose architecture patterns (or avoid antipatterns or pitfalls) based on behavior and/or capabilities, but each pattern changes capabilities, either improving or degrading them. In fact, the primary way to select and, more importantly, compare patterns lies in their impact on capabilities. To that end, each architecture pattern in our book features one or more trade-off charts, discussing how each capability either improves or degrades based on the pattern (or its absence).

Antipatterns

While building architects didn't immediately create the concept of antipatterns, software sure did. An antipattern isn't just a bad idea — the term "bad idea" covers that eventuality. Instead, an *antipattern* initially looks like a pattern, until the architect discovers that it leads to negative outcomes. With a metaphorical flourish, Martin Fowler describes an antipattern as "a solution that initially looks like an attractive road lined with flowers…but further on leads you into a maze filled with monsters."

Context matters a lot for antipatterns: an attractive solution in some contexts can lead to negative outcomes in others. Often, whether a solution is a pattern or an antipattern depends on the context in which the architect applies it. This is the second distinguishing factor of an antipattern: generally, more favorable alternatives exist for many contexts.

Documenting software architecture antipatterns is critical, because architecture solutions have many more antipatterns than design or process solutions do. Because architecture decisions involve so many trade-offs, identifying where a pattern works (or becomes an antipattern) creates lots of complexity.

Understanding how each pattern compares to related patterns, antipatterns, and pitfalls allows architects to steer their solutions toward the best (or sometimes just the least worst) combination of forces and hone their analysis skills.

Pitfalls

We distinguish between *antipatterns* and *pitfalls* in this book because they exhibit slightly different characteristics. An *architectural pitfall*, unlike an antipattern, only superficially seems like a good idea and leads to almost immediate negative consequences. Unlike antipatterns, there's typically no alternative except to stop doing it!

Pitfalls are quite numerous in software architecture; we only cover a few.

The Pattern, Antipattern, Pitfall Spectrum

We documented one of the inconvenient facts about software architecture in the second edition of *Fundamentals of Software Architecture* as our Third Law of Software Architecture.

> **THIRD LAW OF SOFTWARE ARCHITECTURE**
>
> Most architecture decisions aren't binary but rather exist on a spectrum between extremes.

While it would be nice to have clean delineations between patterns and antipatterns, like everything else, they reside on a spectrum. By definition, antipatterns look like a good idea and only turn bad later. But if an architect only uses the beneficial part of the antipattern, they never stumble into the metaphorical maze filled with monsters. In other words, many of the antipatterns we discuss here aren't bad initially, but grow worse over time, complexity, or some other aggravating factor.

The same holds true for pitfalls, but with a much shorter time until they turn bad. For example, the Grains of Sand Pitfall (see Chapter 21) describes building an overly granular microservices ecosystem. If an architect has a small problem with performance or scale requirements, they can survive with this pitfall…until they try to enhance it.

When architects consider antipatterns, realize that they exist on a spectrum: they *can* be beneficial in some situations but come with dangerous disadvantages. Part of maturity as a software architect lies in understanding when a solution is appropriate and when it might degrade into an antipattern.

How Is This Book Different from Other Patterns Books?

This book is markedly different from other patterns books, even those that cover architecture patterns, based on our experience of how software architects utilize patterns in the real world: in combination with numerous other forces, not in convenient isolation. Our book differs in the following ways:

How we selected patterns

> As we mentioned in the introduction, we chose the patterns in this book based not on a particular specialized depth but rather based on breadth — what are the fundamental patterns each software architect should be aware of when making choices? Much like our catalog of architecture styles in our *Fundamentals of Software Architecture* book, this book is meant to catalog fundamental architecture patterns and common things to avoid (antipatterns and pitfalls), which are extremely important to new architects. For example, antipatterns by definition *look* like good solutions but lead to undesirable outcomes.
>
> Because our selection of patterns is broader than most patterns books, for each pattern we supply metadata that includes the context for the pattern. Additionally, in the section on contexts in Chapter 2 (see "Patterns by Context"), we describe each context and the applicable patterns that fall into it. We ended up alphabetizing the patterns rather than trying to categorize them — many of them fall into multiple categories, which is best handled by the context cross-reference.

Different levels of abstraction

> Most patterns books that cover software architecture hyper-specialize because the topic is so broad. For example, architects can find books on enterprise architecture patterns, pattern of application architecture, distributed architecture patterns, and so on. Each book offers patterns at a certain level of abstraction. In the case of the distributed architecture book, every pattern exists within that context and with some base

assumptions. Conversely, we mean this book to be the first patterns book that a new software architect consumes, covering fundamental patterns in software architecture. Because of that, our patterns don't have a consistent context. For example, some of our patterns deal with distributed architecture, while others focus on reuse.

To make it easier for readers to make sure that a particular pattern applies, each pattern includes the contexts where it's applicable as part of the metadata for each pattern, listed at the beginning of each chapter.

Focus on trade-offs

We famously focus on trade-off analysis in our books on software architecture; the focus on pattern trade-offs sets this book apart. We catalog the improved and degraded capabilities for each pattern (and antipattern), providing charts that allow easy trade-off analysis. A shortcoming of most patterns books is a narrow focus on each pattern in isolation. However, as experienced software architects know, other choices they make may impact the trade-offs they chose the pattern for. So, in many patterns, we also include influences that shift the trade-offs in one direction or another and delineate what those changes are. This context is critical for architects to assess the holistic impact of their choices, not in an idealized stateless world.

Facilitating pattern comparison

Architects don't encounter patterns in isolation — they exist within the larger context of the entire system. Design patterns have the luxury of existing more or less in isolation, but architecture patterns can't. Thus, throughout the book, we make it easy for architects to compare relevant dimensions of the architecture and the impact of the pattern. For example, when we compare the Orchestration pattern (see Chapter 27) and the Choreography pattern (see Chapter 9), we compare them to each other: a workflow either has an orchestrator or it doesn't (making it choreography), so the relevant comparison is between the two patterns, not the absence of the pattern, which helps contextualize the decision.

Cataloging common influences

The other unique set of trade-off analyses in our book deal with how the influences of other decisions change the baseline trade-offs. For example, in many workflow patterns, the choice of synchronous or asynchronous communication may significantly change trade-offs like responsiveness and scalability. For each pattern with influences, we include sections of common ones and how they impact the trade-offs for that pattern.

Governing patterns

While many books describe patterns, we also include sections on common ways to *govern* many patterns. Architects are responsible for both the choice of architectural capabilities and how to preserve those qualities over time, typically referred to as architectural governance. For relevant patterns, we also show common ways to make sure that an architect can track the values of their chosen capabilities over time.

Ready for AI skills

This is the first software architecture book optimized for "wiring" into the skills of generative AI. Developers can wire skills into a coding agent's context, providing useful information it can leverage to provide advice. In general, architects shouldn't trust the architectural advice of LLMs (which will be a statistically likely answer, not necessarily a correct answer). However, if architects use this book as context, it utilizes the strongest aspect of LLMs: pattern matching between relevant patterns. The actual trade-off analysis was performed by humans (the authors, none of whom are AI), and the LLM uses the trade-off charts and explanations to derive results. This book is structured to allow LLMs to leverage its analysis and reasoning. While we didn't design it with this purpose at the outset, it seems that our goal of optimizing toward decision making for human architects benefits LLMs too.

Most books on architecture patterns catalog each pattern in isolation, discussing each feature independently of other patterns. However the software architect's job involves performing *trade-off analysis*, which often means choosing between two patterns that each offer a solution, but with different trade-offs. This book catalogs the architectural capabilities affected by each pattern, then consolidates those comparisons into tables. This allows for better "apples-to-apples" evaluation and a way to compare patterns for more effective trade-off analysis. To do that, we utilize *qualitative analysis*, described next.

Qualitative Analysis

A popular and distinguishing feature of our *Fundamentals of Software Architecture* book is its comparison charts, which we use to evaluate each architectural style. (An example of our architectural capabilities analysis for the microservices architectural style appears in Figure 1-1.)

	Architectural characteristic	Star rating
	Overall cost	$$$$$
Structural	Partitioning type	Domain
	Number of quanta	1 to many
	Simplicity	★
	Modularity	★★★★★
Engineering	Maintainability	★★★★★
	Testability	★★★★★
	Deployability	★★★★★
	Evolvability	★★★★★
Operational	Responsiveness	★★
	Scalability	★★★★★
	Elasticity	★★★★
	Fault tolerance	★★★★★

Figure 1-1. The architectural characteristics trade-off chart for microservices from Fundamentals of Software Architecture, second edition

In Figure 1-1, starting with structural elements, we evaluate a specific architecture's typical characteristics and capabilities individually on a scale of one to five stars. This is an example of *qualitative analysis*.

How do architects make decisions about nontrivial solutions without building things? Architects regularly encounter problems they've never seen before: given the heterogeneous nature of software architecture, it could be that no one on earth has seen that exact problem. How can they make intelligent decisions if they can't look to what others have done in the past?

For example, let's say an architect has been tasked with implementing a new accounting workflow for a microservices architecture. Which transactional communication pattern should they use? We identified eight possibilities in our book *Software Architecture: The Hard Parts* (O'Reilly, 2021, coauthored with Pramod Sadalage and Zhamak Dehghani).

One option is for the architect to go to their boss and inform them, "Boss, I have good news and bad news. The good news is that we will implement the objectively best possible workflow for the new system…because we're going to build all eight possible solutions, measure the results, keep the best one, and throw the seven deficient solutions away. The bad news? It will take eight times longer and cost eight times more."

Most architects don't live in that world, which is where *qualitative* analysis comes in.

Building Qualitative Analysis

We developed qualitative analysis while teaching workshops for new architects, to compare architectural styles by what really distinguished them. From the split perspective of behavior and capabilities, the generic architectural styles are generic for a reason — developers can implement virtually any behavior in any generic style. What really differentiates the styles is how they handle different architectural capabilities.

Originally, we had a much cruder scale, using thumbs-up and thumbs-down icons for whether the style supported the architectural capabilities or not, as

illustrated in Figure 1-2. However, we quickly realized that a binary scale lacked nuance — some architectural styles support capabilities better than others. We briefly added "sideways thumb" to act as a neutral indicator, but still felt that we didn't have a broad enough scale for useful comparison.



Figure 1-2. Our initial pass at qualitative analysis used a binary scale

That's when we settled on the five-star scale we've used since. A 20% difference between each value is fine-grained enough for useful distinction but not so fine-grained as to indicate artificial precision. For example, a 100-point scale wouldn't be useful here, because that level of precision isn't really possible when making a judgment call. Finding the correct granularity for the scale is key.

So, how did we come up with the values for the star ratings?

The first step in building a qualitative analysis chart is deciding what the comparison criteria should be. For architectural styles, we chose to compare the architectural capabilities that are most impacted when changing from one style to another. In *Software Architecture: The Hard Parts*, we built our qualitative analysis by comparing different trade-offs given different combinations of forces.

In Figure 1-3, we compare each of the named sagas on the left based on its level of coupling (*decoupling*), how simple it is for developers to understand and implement (*simplicity*), its perceived level of performance (*responsiveness*), and how well it handles a large number of concurrent users (*scalability*).



Figure 1-3. Comparing distributed architecture transactional workflows qualitatively

In each case, we built the star ratings qualitatively based on inherent knowledge of what it supports well or not. For example, in Figure 1-3, we scored scalability for the fairy tale saga as four stars, whereas the same score for the epic saga is one star. We didn't build all possible solutions to determine this rating. However, we assessed what characteristics enable scalability in distributed architectures. The fairy tale saga, which uses

asynchronous communication, eventual consistency, and orchestration for coordination, features two of the three settings that allow maximum scalability (asynchronous communication, eventual consistency, and choreography). Alternatively, the epic saga utilizes synchronous communication, atomic consistency, and orchestration, all of which decrease scalability: synchronous communication ties up requests with wait states, atomic consistency requires intra-service coordination, and orchestration creates bottlenecks.

Thus, even though we haven't implemented each possible solution, understanding desirable (and undesirable) characteristics allows us to perform an approximate analysis.

One of the unique features of this book is its use of qualitative analysis for each architecture pattern. Most books treat patterns in isolation, but architects must often choose between competing solutions. By evaluating and analyzing comparison criteria, architects can more readily make intelligent, informed decisions. We encourage this kind of analysis and will illustrate its effectiveness throughout the book.

Building Quantitative Analysis

We almost always prefer quantitative analysis, based on real measurements. For implementations based on qualitative analysis, it's a good idea to follow up with quantitative analysis, typically in the form of architecture fitness functions.

Originally defined in the book *Building Evolutionary Architectures* (O'Reilly, 2022), by Neal Ford, Rebecca Parsons, Patrick Kua, and Pramod Sadalage, an *architecture fitness function* acts as a form of unit test, but for architectural capabilities rather than the problem domain. In other words, fitness functions test capabilities, while traditional evaluations (like unit, functional, and user acceptance tests) test behavior. Thus, some fitness functions are implemented like unit tests and run as part of automated builds, but fitness functions for capabilities such as responsiveness and scalability with monitors must be implemented by architects. The

implementation scopes of fitness functions are broader than those of other types of tests. As illustrated in Figure 1-4, fitness functions overlap with traditional testing but also encompass operational measures, such as monitors.

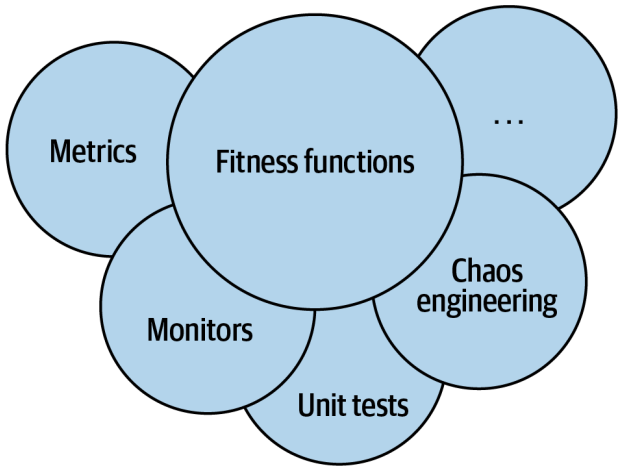


Figure 1-4. Fitness functions have broader scopes than traditional testing

Architects have many tools to assist in building fitness functions. For each applicable pattern, we'll show how to build governance for important characteristics using fitness functions.

How to Use Chapter 2, "Capability and Context Cross-Reference"

As we mentioned before, one of the unique aspects of this book is our focus on how each pattern impacts certain capabilities. To that end, each pattern includes consistent metadata that lists the contexts where the pattern is relevant and what capabilities the pattern impacts.

We want this book to be useful as a reference and a quick aid for architects trying to make decisions. To that end, we created a *reverse index* (Chapter 2) that turns the metadata charts inside out and lists each capability (along with our definition) along with the patterns that affect it. This provides a quick guide for architects looking for a particular capability and pattern intersection. For example, if an architect designs a system that requires high responsiveness, they can look at the capabilities list in Chapter 2 for all the patterns that improve (or degrade) it. The second part is important — if an architect has a critical need for some capability, they need to know what patterns degrade it as much as the ones that improve it. In this book, readers don't have to go through each chapter — we've built a useful cross-reference.

Similarly, we have a list of each context addressed by our patterns in Chapter 2 in "Patterns by Context", which helps architects narrow the set of patterns to applicable contexts. For example, if an architect is primarily building microservice architectures, they're more interested in that context than others (although many of the communication patterns will also apply).

Common Definitions

Many of our readers have already seen *Fundamentals of Software Architecture*, *Software Architecture: The Hard Parts*, and/or *Head First Software Architecture* (O'Reilly, 2024). For those who have, we're deeply grateful! They're already familiar with much of the terminology we defined and refined in those books.

However, we don't expect everyone to have read all these books. Some readers may feel like they're watching the final installment of a superhero movie without having seen the prequels. Additionally, this being a patterns book, readers may consume it in at least two distinct ways: as a narrative or as a reference to look up particular patterns. Many patterns touch on the same common concerns, and we didn't want to duplicate them in each chapter (which would make it hard to read as a narrative).

Because of both these issues, we've consolidated definitions of terms and concepts that cut across patterns into a single chapter in Chapter 37, and the various patterns include cross-references to relevant definitions. For example, we don't want to write the same definition for *architecture quantum* for each pattern that discusses it. Instead, the definition appears once, in "Architecture Quantum", and is linked from each chapter. Unfamiliar terms can likely be found there.

Chapter 2. Capability and Context Cross-Reference

Patterns exist in part because they have catchy, evocative names. However, a clever name doesn't provide a huge amount of context, in particular when addressing the capabilities that a particular pattern improves or degrades. Architects need patterns that enhance certain capabilities, so they need a way to find applicable patterns quickly.

That's what this chapter represents: an inside-out reverse index keyed on *capabilities* (in "Patterns by Capability") and within particular *contexts* (in "Patterns by Context"). When an architect is looking for patterns that improve certain capabilities, they can see the entire list here. Conversely, an architect may want to avoid a particular pattern because it degrades some desirable capability too much. Each of our patterns includes what capabilities it improves and degrades, along with the context where the pattern applies.

To use this chapter, an architect should complete the sentence: "I need a pattern addressing the capability of…." Similarly, if an architect works within a particular context such as microservices, they can complete the sentence: "I need a pattern addressing the context of *microservices*." This chapter provides a pre-filtered view focused on capabilities that architects need to consider (either to improve or prevent from degrading). Similarly, because our patterns cover a wide range of contexts, we identify each pattern's context in its metadata, allowing architects to filter by context.

One of the common questions we avoid in our *Fundamentals of Software Architecture* (2nd edition) book is the request to provide a comprehensive list of architectural capabilities. This is for several reasons. First, it's a never-ending list: architects constantly build new capabilities as our ecosystem evolves. Second, different sources have different definitions of architectural capabilities: they aren't physics terms or even industry

standards. Third, architects are empowered to create brand new capabilities just by adding "-ility" to an existing word, so there can never be a truly comprehensive list.

For this book, we didn't start with a list of capabilities we wanted to track. Rather, we analyzed each pattern in isolation, determining what capabilities were improved or degraded. At the end of writing, we created this inside-out list of each capability, along with all the patterns affected by it. Thus, we have accidentally created a standard list of architectural capabilities that commonly affect our set of architecture patterns, useful as a baseline of capabilities that architects should consider when choosing patterns.

Patterns by Capability

Architectural capabilities — also called architectural characteristics, nonfunctional requirements, cross-cutting requirements, and other names — represent one half of structural design in software architecture (the other part concerns the behavior, also known as the domain). When architects search for patterns, they need to understand what impact using that pattern will have on common capabilities, and often the capabilities are the driving criteria. For example, when building highly scalable architectures, architects must choose workflow patterns that support (rather than degrade) scalability.

This section deals with *primary* capabilities for each pattern, while trying to avoid secondary and tertiary effects. For example, virtually every architecture pattern involves adding structure of some kind, which in turn will likely increase *cost*. However, for most patterns, that is a secondary concern — it's not the reason an architect will choose a particular pattern. We limited our list of capabilities just to keep the list from exploding: software architecture deals with trade-offs, so any comprehensive analysis of a pattern would be enormous.

We didn't start with a list of capabilities for this book; a useful generic list is difficult to create. Rather, we analyzed each pattern (and antipattern) directly, focusing solely on its effects on capabilities. We tracked this

pattern analysis in a tool that automatically built the list of common capabilities, which we reflect here along with common definitions. We call these *common definitions* because one of the perpetual issues with these terms is that they aren't physics terms with concrete, agreed upon definitions. Within organizations, an architect may encounter different perspectives on some of these capabilities.

Adaptability

Adaptability describes how well the architecture adapts to new situations, such as incorporating new platforms or technologies into an organization. It's also the basis for several architecture patterns, where the architect tries to build a system that can adapt to new domain and technology changes. For example, the microkernel architecture style (covered in our book, *Fundamentals of Software Architecture*, 2nd edition), which consists of an unchanging core component and plug-ins where architects implement changes, is ideally suited for adaptability: keeping existing plug-in behavior while incrementally adding plug-ins to adapt to new situations.

Patterns that improve adaptability are generally good candidates for long-term strategic solutions or in fast-moving, highly volatile projects. These patterns appear in Table 2-1.

Table 2-1. Patterns that improve adaptability

Name	Chapter
Backend for Frontend pattern	Chapter 7
Choreography pattern	Chapter 9
Cross-Cutting State pattern	Chapter 13
Data-Based Event Contract pattern	Chapter 14
Extensible Events pattern	Chapter 18
Implicit Workflow State pattern	Chapter 22
Nexus Architecture pattern	Chapter 26
Stamp Coupling antipattern	Chapter 31

If an architect is looking to build a solution where they can continuously adapt to situations, they may want to avoid patterns that degrade adaptability, listed in Table 2-2.

Table 2-2. Patterns that degrade adaptability

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Explicit Workflow State pattern	Chapter 17
Front Controller pattern	Chapter 19
Frozen Caveman pitfall	Chapter 20
Grains of Sand pitfall	Chapter 21
Key-Based Event Contract pattern	Chapter 23
Orchestration pattern	Chapter 27
Service Sharding pattern	Chapter 30
Stovepipe Architecture antipattern	Chapter 32

Agility

Defining *agility* is slippery for two reasons, both related. First, a huge number of definitions exist, mostly skewed toward a particular perspective. But secondly, and the reason it's so hard to pin down, is that agility is a *composite* architectural capability.

COMPOSITE ARCHITECTURAL CAPABILITIES

In our book, *Fundamentals of Software Architecture*, we define *composite* architectural capabilities as those that don't have an objectively measurable value. That's generally because this term is actually composed of a number of other capabilities, many of which *do* have objective measurements. For example, an architect can measure responsiveness or modularity with suitable metrics tools. However, no single value exists that indicates that an architecture is officially "agile."

What architects should do in this situation is decompose the composite capability. Agility is composed of *deployability* (frequency and success of deployments), *testability* (quality and coverage of tests and fitness functions), *modularity* (good modularity allows the system to make changes quickly without breaking anything), and so on.

Given that agility consists of several related capabilities, our assessment from a capabilities standpoint looked at the preponderance of the capabilities — did most of them yield better results (improved agility) or worse (degraded agility)? The patterns that generally improve important aspects of agility appear in Table 2-3.

Table 2-3. Patterns that improve agility

Name	Chapter
Backend for Frontend pattern	Chapter 7
Extensible Events pattern	Chapter 18

Degraded *agility* generally indicates that an architect has chosen a pattern or set of patterns that have a negative impact on the many features that make up agility for offsetting reasons. However, if *agility* is an important criterion

on a project, this chart prevents using one or more of the patterns appearing in Table 2-4.

Table 2-4. Patterns that degrade agility

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Frozen Caveman pitfall	Chapter 20
Overcentralization antipattern	Chapter 28
Stovepipe Architecture antipattern	Chapter 32

Atomicity

Atomicity isn't tracked as an individual capability in this book, but rather as a composite (see "Composite Architectural Capabilities") consisting of *data consistency* (see "Data Consistency") and *data integrity* (see "Data Integrity").

These two terms are often used interchangeably (we define them separately in "Data Consistency" and "Data Integrity"), but collectively are known as *atomicity*. The easiest way to think about the difference is illustrated in the diagram shown in Figure 2-1.

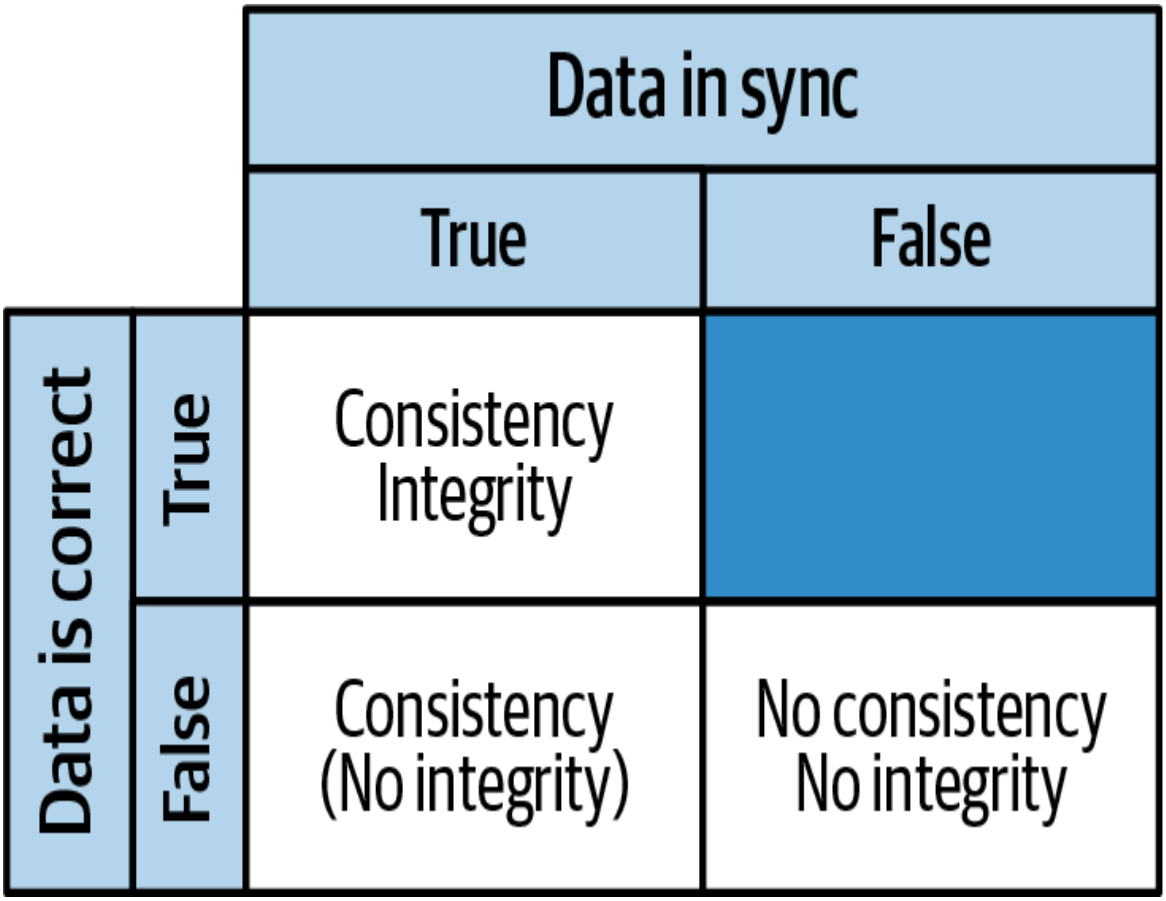


Figure 2-1. The differences and overlap between data consistency and data integrity

In Figure 2-1, the rows represent the boolean condition where *data is correct*, and the columns represent the condition where *data is in sync*. When data is both correct and in sync, we have both data consistency and integrity. When data *isn't* correct but in sync, we have consistency but no integrity. And when data is neither correct nor in sync, we have neither property. Note that one impossible case, where data is correct but not in sync, can't make sense: data cannot be correct if it differs across nodes.

While we may refer to atomicity in text, in our trade-off comparisons we'll use the individual capabilities. Most of our patterns and antipatterns show the same impact on both data consistency and integrity, but a few patterns diverge slightly for pattern-specific reasons (which we'll call out).

Auditability

Auditability is the capability to be audited, providing architects with the ability (sometimes required by law) to track all interactions of interest, such as monetary transfers.

Some distributed architectures make observability and therefore auditability easier, and one pattern also facilitates this operational capability, shown in Table 2-5.

Table 2-5. Pattern that improves auditability

Name	Chapter
Event Sourcing pattern	Chapter 16

To degrade auditability, a pattern would have to impede the ability to log changes. Fortunately, none of our patterns degrade auditability.

Availability

Availability is a capability that overlaps between architecture and operations, which ranks how available an endpoint is in a distributed architecture. This capability is the first of the seven Fallacies of Distributed Computing, which is "The network is reliable." This is often a composite architectural capability with reliability, as they overlap in scope.

Patterns that improve availability are key for architects who need to build systems with consistent uptime, as shown in Table 2-6.

Table 2-6. Patterns that improve availability

Name	Chapter
Backend for Frontend pattern	Chapter 7
Choreography pattern	Chapter 9
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Domain-Broker pattern	Chapter 15

For systems where availability is key, architects should be on the lookout for patterns that make availability worse and either avoid them or mitigate the issue by introducing redundancy or some other technique, as shown in Table 2-7.

Table 2-7. Patterns that degrade availability

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Cross-Cutting State pattern	Chapter 13
Front Controller pattern	Chapter 19
Frozen Caveman pitfall	Chapter 20
Orchestration pattern	Chapter 27
Overcentralization antipattern	Chapter 28
Stovepipe Architecture antipattern	Chapter 32

Bandwidth Usage

Bandwidth usage refers to another of the aforementioned Fallacies of Distributed Computing, number three: "Bandwidth is infinite." As architects painfully discover, bandwidth refers to the rate at which a system may transmit data over a network connection. Bandwidth *usage*, then, refers to patterns that improve rates of transmission.

Improving bandwidth usage refers to patterns that use available communication resources as efficiently as possible. For example, for bulky data, compressing it over a long-distance communication link likely optimizes the available bandwidth. See Table 2-8 for the pattern that improves bandwidth usage.

Table 2-8. Pattern that improves bandwidth usage

Name	Chapter
Key-Based Event Contract pattern	Chapter 23

As in all decisions in software architecture, architects may trade bandwidth usage for other priorities. However, they must be wary of accidentally degrading bandwidth when it's critical, and therefore should stay away from the patterns shown in Table 2-9.

Table 2-9. Patterns that degrade bandwidth usage

Name	Chapter
Data-Based Event Contract pattern	Chapter 14
Extensible Events pattern	Chapter 18
Stamp Coupling antipattern	Chapter 31

Cost

Cost is an obvious, nearly pervasive capability (or constraint) on software projects. Some architecture patterns cost more to implement for one reason or another, affecting cost.

Improving cost means finding a less costly solution to a problem by applying a pattern or avoiding an antipattern that's more costly; patterns that improve cost are shown in Table 2-10.

Table 2-10. Patterns that improve cost

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Stovepipe Architecture antipattern	Chapter 32

If cost is an overriding concern, architects should check to make sure that solutions they've chosen don't negatively impact the cost profile of the solution.

Patterns and antipatterns that degrade cost appear in Table 2-11.

Table 2-11. Patterns that degrade cost

Name	Chapter
Backend for Frontend pattern	Chapter 7
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Domain-Broker pattern	Chapter 15
Event Sourcing pattern	Chapter 16
Frozen Caveman pitfall	Chapter 20
Grains of Sand pitfall	Chapter 21
Microservice All the Things pitfall	Chapter 24
Multi-Broker pattern	Chapter 25
Nexus Architecture pattern	Chapter 26
Thread Delegate pattern	Chapter 34

Data Consistency

Data consistency ensures that data across the system is in sync and consistent across databases and tables. Generally, in architectures that use a single database, the database handles this capability automatically. However, in distributed architectures, teams often must implement this feature via a number of architecture patterns.

Improving data consistency implies that the pattern makes it easier to keep data relationships consistent. In the easiest, most obvious case, allowing a relational database to handle it automatically is convenient and foolproof. Often, in distributed architectures, a single database incurs too many negative trade-offs.

The patterns that improve data consistency appear in Table 2-12.

Table 2-12. Patterns that improve data consistency

Name	Chapter
Async Chain of Persistence pattern	Chapter 6
Compensating Updates pattern	Chapter 11
Cross-Cutting State pattern	Chapter 13
Front Controller pattern	Chapter 19
Key-Based Event Contract pattern	Chapter 23
Orchestration pattern	Chapter 27
Overcentralization antipattern	Chapter 28
Read-Through/Write-Through Cache pattern	Chapter 29
Service Sharding pattern	Chapter 30

Table 2-13 shows patterns that degrade data consistency.

Table 2-13. Patterns that degrade data consistency

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Cache Aside pattern	Chapter 8
Choreography pattern	Chapter 9
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Data-Based Event Contract pattern	Chapter 14
Event Sourcing pitfall	Chapter 16
Grains of Sand pitfall	Chapter 21
Microservice All the Things pitfall	Chapter 24

Data Integrity

Data integrity is a property of a single logical data source — does each datum correctly reflect what it's supposed to? It covers accuracy, completeness, validity, and satisfaction of invariants/constraints within one architecture quantum. Integrity is essentially a point-in-time, single-owner question: "Is this value correct and well-formed where it lives?" It's what relational constraints, foreign keys, and validation rules protect in a relational database.

Table 2-14 shows patterns that improve data integrity.

Table 2-14. Patterns that improve data integrity

Name	Chapter
Async Chain of Persistence pattern	Chapter 6
Compensating Updates pattern	Chapter 11
Cross-Cutting State pattern	Chapter 13
Front Controller pattern	Chapter 19
Key-Based Event Contract pattern	Chapter 23
Orchestration pattern	Chapter 27
Overcentralization antipattern	Chapter 28
Service Sharding pattern	Chapter 30

Table 2-15 shows patterns that degrade data integrity.

Table 2-15. Patterns that degrade data integrity

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Cache Aside pattern	Chapter 8
Choreography pattern	Chapter 9
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Data-Based Event Contract pattern	Chapter 14
Event Sourcing pattern	Chapter 16
Grains of Sand pattern	Chapter 21
Microservice All the Things pitfall	Chapter 24

Debuggability

Debuggability refers to the ease of debugging a particular pattern. Debugging is notoriously difficult in distributed architectures because it isn't a single debugging context but several. Often, architects must rely on forensic debugging using log files, especially in architectural patterns that use asynchronous communication.

Table 2-16 shows patterns that improve debuggability.

Table 2-16. Patterns that improve debuggability

Name	Chapter
Architecture Sinkhole antipattern	Chapter 5
Event Sourcing pattern	Chapter 16
Front Controller pattern	Chapter 19
Orchestration pattern	Chapter 27

Table 2-17 shows patterns that degrade debuggability.

Table 2-17. Patterns that degrade debuggability

Name	Chapter
Choreography pattern	Chapter 9
Frozen Caveman pitfall	Chapter 20

Deployability

Deployability refers to the amount of ceremony involved with releasing the software, the frequency at which releases occur, and the overall risk of deployment.

Table 2-18 shows patterns that improve deployability.

Table 2-18. Patterns that improve deployability

Name	Chapter
Backend for Frontend pattern	Chapter 7
Extensible Events pattern	Chapter 18

Table 2-19 shows patterns that degrade deployability.

Table 2-19. Patterns that degrade deployability

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Frozen Caveman pitfall	Chapter 20
Overcentralization antipattern	Chapter 28
Stovepipe Architecture antipattern	Chapter 32

Domain Alignment

Domain abstraction is an idea derived from Domain-Driven Design (DDD). As architectures like microservices rely on DDD philosophies, it's often important to architects how much a particular pattern aligns with domain (versus technical) solutions. For example, the traditional layered monolithic architecture is technically partitioned, separating layers by technical responsibility. Conversely, the modular monolith style separates components based on DDD, yielding better *domain alignment*.

Table 2-20 shows patterns that improve domain alignment.

Table 2-20. Patterns that improve domain alignment

Name	Chapter
Choreography pattern	Chapter 9
Domain-Broker pattern	Chapter 15
Service Sharding pattern	Chapter 30

Table 2-21 shows patterns that degrade domain alignment.

Table 2-21. Patterns that degrade domain alignment

Name	Chapter
Cross-Cutting State pattern	Chapter 13
Explicit Workflow State pattern	Chapter 17
Front Controller pattern	Chapter 19
Implicit Workflow State pattern	Chapter 22
Orchestration pattern	Chapter 27
Overcentralization antipattern	Chapter 28
Stovepipe Architecture antipattern	Chapter 32

Elasticity

Elasticity describes a system that's able to expand and respond quickly to unexpected or anticipated extreme loads (such as going from 20 to 250,000 users instantly). One of the key metrics architects look at when evaluating elasticity is Mean Time To Start (MTTS). When designing an elastic system, architects pay attention to how quickly they can start up new services.

Table 2-22 shows patterns that improve elasticity.

Table 2-22. Patterns that improve elasticity

Name	Chapter
Cache Aside pattern	Chapter 8
Choreography pattern	Chapter 9
Contextual Events pattern	Chapter 12
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Data-Based Event Contract pattern	Chapter 14
Domain-Broker pattern	Chapter 15
Multi-Broker pattern	Chapter 25
Service Sharding pattern	Chapter 30
Supervisor Consumer pattern	Chapter 33
Thread Delegate pattern	Chapter 34
Workflow Event pattern	Chapter 35

Table 2-23 shows patterns that degrade elasticity.

Table 2-23. Patterns that degrade elasticity

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Async Chain of Persistence pattern	Chapter 6
Compensating Updates pattern	Chapter 11
Front Controller pattern	Chapter 19
Frozen Caveman pitfall	Chapter 20
Key-Based Event Contract pattern	Chapter 23
Orchestration pattern	Chapter 27
Overcentralization antipattern	Chapter 28
Stovepipe Architecture antipattern	Chapter 32

Error Handling

Error handling refers to how easy it is for an architect to facilitate error handling within the pattern. For example, workflows that use the Orchestration pattern (see Chapter 27) make error handling easier because of the presence of the orchestrator component.

Table 2-24 shows patterns that improve error handling.

Table 2-24. Patterns that improve error handling

Name	Chapter
Backend for Frontend pattern	Chapter 7
Front Controller pattern	Chapter 19
Orchestration pattern	Chapter 27
Workflow Event pattern	Chapter 35

Table 2-25 shows patterns that degrade error handling.

Table 2-25. Patterns that degrade error handling

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Choreography pattern	Chapter 9
Frozen Caveman pitfall	Chapter 20
Thread Delegate pattern	Chapter 34

Evolvability

Evolvability, as defined in Neal's book, *Building Evolutionary Architectures*, describes the ability for architects to fundamentally change an architecture over time. This capability is often contrasted with adaptability, which describes adding new features while preserving old

ones. Evolvability, on the other hand, describes evolving a system by replacing parts of the system over time.

Table 2-26 shows patterns that improve evolvability.

Table 2-26. Patterns that improve evolvability

Name	Chapter
Architecture Sinkhole antipattern	Chapter 5
Backend for Frontend pattern	Chapter 7
Choreography pattern	Chapter 9
Cross-Cutting State pattern	Chapter 13
Data-Based Event Contract pattern	Chapter 14
Extensible Events pattern	Chapter 18
Implicit Workflow State pattern	Chapter 22
Nexus Architecture pattern	Chapter 26
Stamp Coupling antipattern	Chapter 31

Table 2-27 shows patterns that degrade evolvability.

Table 2-27. Patterns that degrade evolvability

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Explicit Workflow State pattern	Chapter 17
Front Controller pattern	Chapter 19
Frozen Caveman pitfall	Chapter 20
Key-Based Event Contract pattern	Chapter 23
Orchestration pattern	Chapter 27
Service Sharding pattern	Chapter 30
Stovepipe Architecture antipattern	Chapter 32

Fault Tolerance

Fault tolerance ensures that, when fatal errors occur, other parts of the system continue to function.

Table 2-28 shows patterns that improve fault tolerance.

Table 2-28. Patterns that improve fault tolerance

Name	Chapter
Async Chain of Persistence pattern	Chapter 6
Backend for Frontend pattern	Chapter 7
Choreography pattern	Chapter 9
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Domain-Broker pattern	Chapter 15

Table 2-29 shows patterns that degrade fault tolerance.

Table 2-29. Patterns that degrade fault tolerance

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Front Controller pattern	Chapter 19
Frozen Caveman pitfall	Chapter 20
Orchestration pattern	Chapter 27
Overcentralization antipattern	Chapter 28
Stovepipe Architecture antipattern	Chapter 32

Flexibility

Flexibility describes patterns that give architects choice in implementation decisions, such as selecting different technologies, runtimes, or datastores for different parts of the system. For example, the Command Query Responsibility Segregation (CQRS) pattern (see Chapter 10) provides architects with the flexibility to independently choose the most fitting runtime and datastore for the command side and query side, allowing each to be optimized for its specific workload.

Table 2-30 shows patterns that improve flexibility.

Table 2-30. Patterns that improve flexibility

Name	Chapter
Backend for Frontend pattern	Chapter 7
Cache Aside pattern	Chapter 8
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Domain-Broker pattern	Chapter 15

Table 2-31 shows patterns that degrade flexibility.

Table 2-31. Patterns that degrade flexibility

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Frozen Caveman pitfall	Chapter 20
Read-Through/Write-Through Cache pattern	Chapter 29

Initial Time to Market

Initial time to market describes patterns that facilitate faster development, virtually always at the expense of some other capabilities. For example, an architect may choose a profoundly simple architectural solution like the layered monolithic architecture to achieve fast time to market because

funding will expire before the team has the ability to build something more sophisticated.

Table 2-32 shows patterns that improve initial time to market.

Table 2-32. Patterns that improve initial time to market

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Stovepipe Architecture antipattern	Chapter 32

Table 2-33 shows patterns that degrade initial time to market.

Table 2-33. Patterns that degrade initial time to market

Name	Chapter
Architecture Sinkhole antipattern	Chapter 5
Grains of Sand pitfall	Chapter 21

Maintainability

Maintainability is the level of effort required to locate and apply changes to the system.

Table 2-34 shows patterns that improve maintainability.

Table 2-34. Patterns that improve maintainability

Name	Chapter
Architecture Sinkhole antipattern	Chapter 5
Backend for Frontend pattern	Chapter 7
Extensible Events pattern	Chapter 18
Nexus Architecture pattern	Chapter 26

Table 2-35 shows patterns that degrade maintainability.

Table 2-35. Patterns that degrade maintainability

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Frozen Caveman pitfall	Chapter 20
Overcentralization antipattern	Chapter 28
Stovepipe Architecture antipattern	Chapter 32

Message Processing Order

Message processing order refers to the order in which messages are processed as opposed to how they're organized in the queue (such as FIFO, or first in, first out), and is often a desirable property of message-based systems. Although it's different from FIFO processing, FIFO queues are required for preserving the message processing order. Often, to achieve higher scale, systems run in parallel and consequently shuffle message processing order, which sometimes leads to domain problems. Thus, architects often choose communication patterns based on whether they can preserve message processing order or not.

Table 2-36 shows the pattern that improves message processing order.

Table 2-36. Pattern that improves message processing order

Name	Chapter
Thread Delegate pattern	Chapter 34

Table 2-37 shows patterns that degrade message processing order.

Table 2-37. Patterns that degrade message processing order

Name	Chapter
Ambulance pattern	Chapter 3
Multi-Broker pattern	Chapter 25
Supervisor Consumer pattern	Chapter 33
Workflow Event pattern	Chapter 35

Modularity

Modularity refers to patterns that impose a particular style of modularity on the architecture pattern. For example, the Orchestrator pattern (see Chapter 27) defines a particular orchestrator component as part of the pattern. We mark this capability as improved when a pattern improves the defined modularity within the system. For example, in microservices, architects consider a pattern that makes it easier to preserve service boundaries to improve modularity because it aligns with the underlying philosophy.

Table 2-38 shows patterns that improve modularity.

Table 2-38. Patterns that improve modularity

Name	Chapter
Architecture Sinkhole antipattern	Chapter 5
Domain-Broker pattern	Chapter 15

Table 2-39 shows patterns that degrade modularity.

Table 2-39. Patterns that degrade modularity

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Frozen Caveman pitfall	Chapter 20
Overcentralization antipattern	Chapter 28

Performance

Performance as a software architecture measurement evaluates how efficiently a system executes work under specific conditions. In layman's terms, performance is how fast the system appears to users.

Actually improving performance is difficult: first, architects must figure out what's causing the poor performance, and then they have to figure out a way to make it better, both of which may be difficult problems. Those problems, however, generally have more to do with system-level operational concerns, not architecture.

Instead, throughout this book, we track *responsiveness* for two reasons. First, many of the patterns we include improve responsiveness without having to improve performance; often, using a pattern is much easier than addressing the underlying root cause. Second, users can't tell the difference between performance and responsiveness, making it a sort of architecture sleight of hand: we can make it appear that we've solved a difficult problem (performance) when all we've really done is utilize a pattern that improves responsiveness.

For patterns that affect responsiveness, see "Responsiveness".

Responsiveness

Responsiveness describes the amount of time it takes to get a response to the user. This capability is often an analog for *performance* because end users generally can't tell the difference between good performance and good responsiveness. Many architecture patterns impact responsiveness, reflected in the following charts.

Table 2-40 shows patterns that improve responsiveness.

Table 2-40. Patterns that improve responsiveness

Name	Chapter
Ambulance pattern	Chapter 3
Cache Aside pattern	Chapter 8
Choreography pattern	Chapter 9
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Data-Based Event Contract pattern	Chapter 14
Multi-Broker pattern	Chapter 25
Read-Through/Write-Through Cache pattern	Chapter 29
Supervisor Consumer pattern	Chapter 33
Thread Delegate pattern	Chapter 34
Workflow Event pattern	Chapter 35

Table 2-41 shows patterns that degrade responsiveness.

Table 2-41. Patterns that degrade responsiveness

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Architecture Sinkhole antipattern	Chapter 5
Async Chain of Persistence pattern	Chapter 6
Backend for Frontend pattern	Chapter 7
Compensating Updates pattern	Chapter 11
Front Controller pattern	Chapter 19
Frozen Caveman pitfall	Chapter 20
Key-Based Event Contract pattern	Chapter 23
Orchestration pattern	Chapter 27
Stovepipe Architecture antipattern	Chapter 32

Reusability

Reusability refers to patterns that facilitate reuse within an architecture, particularly code-level reuse. From a development standpoint, code reusability is often viewed as a universal good (less effort, redundancy, and so on). However, architects must view it with more nuance, looking at volatility as well. Code that's highly reused and volatile causes churn across

systems, as every time the reuse target changes, all the reusers must coordinate around that change.

Patterns that improve code reusability are typically focused on reuse; it's rare that extra reuse is a side effect of another pattern. However, some patterns do allow architects to partition code in ways that make it easier to focus reusable assets. For example, separating technical from domain concerns often helps reuse because technical artifacts tend to be easier to reuse (because they tend to be less volatile than domain concerns). The patterns listed in this book that improve code reusability appear in Table 2-42.

Table 2-42. Patterns that improve reusability

Name	Chapter
Choreography pattern	Chapter 9
Nexus Architecture pattern	Chapter 26

Patterns that degrade code reusability make it more difficult to reuse code because some other style or pattern forces decoupling. For example, a common mantra in the microservices world is "duplication is better than coupling," observing that, while both are bad, coupling is considered the worse offense in microservices because of the underlying philosophy. Our patterns that degrade code reusability appear in Table 2-43.

Table 2-43. Patterns that degrade reusability

Name	Chapter
Backend for Frontend pattern	Chapter 7
Front Controller pattern	Chapter 19
Orchestration pattern	Chapter 27
Overcentralization antipattern	Chapter 28

Scalability

Scalability is a function of system capacity and growth over time; as the number of users or requests increase in the system, responsiveness, performance, and error rates remain constant.

Table 2-44 shows patterns that improve scalability.

Table 2-44. Patterns that improve scalability

Name	Chapter
Cache Aside pattern	Chapter 8
Choreography pattern	Chapter 9
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Contextual Events pattern	Chapter 12
Data-Based Event Contract pattern	Chapter 14
Domain-Broker pattern	Chapter 15
Multi-Broker pattern	Chapter 25
Service Sharding pattern	Chapter 30
Supervisor Consumer pattern	Chapter 33
Thread Delegate pattern	Chapter 34
Workflow Event pattern	Chapter 35

Table 2-45 shows patterns that degrade scalability.

Table 2-45. Patterns that degrade scalability

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Async Chain of Persistence pattern	Chapter 6
Compensating Updates pattern	Chapter 11
Front Controller pattern	Chapter 19
Frozen Caveman pitfall	Chapter 20
Key-Based Event Contract pattern	Chapter 23
Orchestration pattern	Chapter 27
Overcentralization antipattern	Chapter 28
Stovepipe Architecture antipattern	Chapter 32

Simplicity

A common, persistent bit of advice every architect receives is to "keep designs simple" (often without telling the architect exactly how to achieve that). We track this as a capability because a key selling point for many patterns is lack of added complexity.

Improving *simplicity* implies that solutions that use this pattern tend to be simpler than the absence of this pattern or other comparable patterns.

Table 2-46 shows patterns that improve simplicity.

Table 2-46. Patterns that improve simplicity

Name	Chapter
Architecture Sinkhole antipattern	Chapter 5
Backend for Frontend pattern	Chapter 7
Read-Through/Write-Through Cache pattern	Chapter 29
Service Sharding pattern	Chapter 30
Stamp Coupling antipattern	Chapter 31

Simplicity becoming degraded implies that the pattern (or antipattern) adds complexity to a solution. Sometimes complexity is warranted (if it's *essential complexity*, a complex problem), but it often grows for bad reasons (*accidental complexity*). Table 2-47 shows patterns that degrade simplicity.

Table 2-47. Patterns that degrade simplicity

Name	Chapter
Ambulance pattern	Chapter 3
Architecture by Implication antipattern	Chapter 4
Cache Aside pattern	Chapter 8
Compensating Updates pattern	Chapter 11
Contextual Events pattern	Chapter 12
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Cross-Cutting State pattern	Chapter 13
Domain-Broker pattern	Chapter 15
Event Sourcing pattern	Chapter 16
Explicit Workflow State pattern	Chapter 17
Extensible Events pattern	Chapter 18
Frozen Caveman pitfall	Chapter 20
Grains of Sand pitfall	Chapter 21

Name	Chapter
Implicit Workflow State pattern	Chapter 22
Microservice All the Things pitfall	Chapter 24
Multi-Broker pattern	Chapter 25
Nexus Architecture pattern	Chapter 26
Overcentralization antipattern	Chapter 28
Supervisor Consumer pattern	Chapter 33
Thread Delegate pattern	Chapter 34
Workflow Event pattern	Chapter 35

Testability

Testability refers to the ease and completeness of testability within a system. For example, the Cross-Cutting State pattern (see Chapter 13) provides a structural component to manage state, making it an ideal place to test the possible states. Without a state component, architects would have to test the system by deriving the state across the services, adding complexity.

Table 2-48 shows patterns that improve testability.

Table 2-48. Patterns that improve testability

Name	Chapter
Architecture Sinkhole antipattern	Chapter 5
Backend for Frontend pattern	Chapter 7
Contextual Events pattern	Chapter 12
Cross-Cutting State pattern	Chapter 13
Event Sourcing pattern	Chapter 16
Explicit Workflow State pattern	Chapter 17
Extensible Events pattern	Chapter 18
Implicit Workflow State pattern	Chapter 22
Nexus Architecture pattern	Chapter 26
Service Sharding pattern	Chapter 30

Table 2-49 shows patterns that degrade testability.

Table 2-49. Patterns that degrade testability

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Frozen Caveman pitfall	Chapter 20
Overcentralization antipattern	Chapter 28
Stovepipe Architecture antipattern	Chapter 32

Throughput

Throughput refers to the actual amount of data successfully transferred or processed over a network or system in a specific amount of time. Along with responsiveness, architects often choose patterns (or avoid antipatterns) based on effects of throughput.

Table 2-50 shows patterns that improve throughput.

Table 2-50. Patterns that improve throughput

Name	Chapter
Ambulance pattern	Chapter 3
Choreography pattern	Chapter 9
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Contextual Events pattern	Chapter 12
Data-Based Event Contract pattern	Chapter 14
Multi-Broker pattern	Chapter 25
Supervisor Consumer pattern	Chapter 33
Thread Delegate pattern	Chapter 34
Workflow Event pattern	Chapter 35

Table 2-51 shows patterns that degrade throughput.

Table 2-51. Patterns that degrade throughput

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Async Chain of Persistence pattern	Chapter 6
Compensating Updates pattern	Chapter 11
Front Controller pattern	Chapter 19
Frozen Caveman pitfall	Chapter 20
Key-Based Event Contract pattern	Chapter 23
Orchestration pattern	Chapter 27

Workflow

Workflow describes the system's ability to manage complex workflows that require multiples of its parts (services) to complete a business request.

Table 2-52 shows patterns that improve workflow.

Table 2-52. Patterns that improve workflow

Name	Chapter
Contextual Events pattern	Chapter 12
Explicit Workflow State pattern	Chapter 17
Front Controller pattern	Chapter 19
Implicit Workflow State pattern	Chapter 22
Overcentralization antipattern	Chapter 28

Table 2-53 shows patterns that degrade workflow.

Table 2-53. Patterns that degrade workflow

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Frozen Caveman pitfall	Chapter 20

Patterns by Context

Part of the definition of *pattern* in the design sense is the presence of an appropriate *context*, which always provides a set of circumstances that makes the pattern a good option (or makes the antipattern more acute).

In each definition throughout the book, we provide metadata about the pattern, and some of the key data are the contexts where this pattern applies.

In this section, we cross-reference each of the contexts that appear in the book and show which patterns, antipatterns, and pitfalls apply to which context. For example, for an architect working in a microservices architecture, they care about that context and not the patterns that deal purely in a layered architecture context.

Agentic Code Generation

Agentic code generation as a practice describes teams leveraging generative AI agents to generate some — or all — of the source code. Unfortunately, unless specifically instructed, these AI agents generally don't care about the quality (behavioral or capabilities) of the resulting code. Table 2-54 shows a pattern in the agentic code generation context.

Table 2-54. Pattern that exists within the agentic code generation context

Name	Chapter
Architecture by Implication antipattern	Chapter 4

Architectural Styles

Architectural styles refers to a distinction we made in the second edition of our book, *Fundamentals of Software Architecture*, which distinguishes styles from patterns (covered in "Styles Versus Patterns"). These patterns affect one or more of the properties that make up a style (component topology, dependencies, physical architecture, deployment, communication style, data topology, and reuse philosophy). Table 2-55 shows patterns in the architectural styles context.

Table 2-55. Patterns that exist within the architectural styles context

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Stovepipe Architecture antipattern	Chapter 32

Caching

The *caching* context arises anytime an architect decides to use caching to distribute the visibility of data. While once rarer, the popularity of microservices has driven many architects to use caching to handle the tasks that were traditionally handled by views in monolithic systems. Table 2-56 shows a pattern in the caching context.

Table 2-56. Pattern that exists within the caching context

Name	Chapter
Cache Aside pattern	Chapter 8
Read-Through/Write-Through Cache pattern	Chapter 29

Contracts

The *contract* context deals with anytime two entities in a software system exchange information: method calls, remote service calls, message or event queues, and so on. The contract defines the amount and metadata about the information the architect is passing and has a surprisingly deep influence on many architectural trade-offs. Table 2-57 shows patterns in the contracts context.

Table 2-57. Patterns that exist within the contracts context

Name	Chapter
Data-Based Event Contract pattern	Chapter 14
Key-Based Event Contract pattern	Chapter 23
Stamp Coupling antipattern	Chapter 31

Data

The *data* context arises anytime an architectural pattern must also deal with data, generally in a database (relational or otherwise). Because of the prevalence of data, this context arises often. Table 2-58 shows patterns in the data context.

Table 2-58. Patterns that exist within the data context

Name	Chapter
Cache Aside pattern	Chapter 8
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Event Sourcing pattern	Chapter 16
Overcentralization antipattern	Chapter 28
Read-Through/Write-Through Cache pattern	Chapter 29

Decision Making

This *decision-making* context is really a process pattern, but it relates directly to architectural decision making and its consequences. Table 2-59 shows a pattern in the decision-making context.

Table 2-59. Pattern that exists within the decision-making context

Name	Chapter
Frozen Caveman pitfall	Chapter 20

Event-Driven Architecture

Event-driven architectures are one of the common distributed architecture styles, primarily involving architectures that use events for communication. Table 2-60 shows patterns in the event-driven architecture context.

Table 2-60. Patterns that exist within the event-driven architecture context

Name	Chapter
Ambulance pattern	Chapter 3
Async Chain of Persistence pattern	Chapter 6
Cache Aside pattern	Chapter 8
Choreography pattern	Chapter 9
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Compensating Updates pattern	Chapter 11
Contextual Events pattern	Chapter 12
Cross-Cutting State pattern	Chapter 13
Data-Based Event Contract pattern	Chapter 14
Domain-Broker pattern	Chapter 15
Explicit Workflow State pattern	Chapter 17
Extensible Events pattern	Chapter 18
Front Controller pattern	Chapter 19

Name	Chapter
Implicit Workflow State pattern	Chapter 22
Key-Based Event Contract pattern	Chapter 23
Multi-Broker pattern	Chapter 25
Orchestration pattern	Chapter 27
Service Sharding pattern	Chapter 30
Stamp Coupling antipattern	Chapter 31
Supervisor Consumer pattern	Chapter 33
Thread Delegate pattern	Chapter 34
Workflow Event pattern	Chapter 35

Event Granularity

The *event granularity* context relates to how fine or coarse-grained events are within an event-driven architecture. This is semantically related to contracts, which also relate to amount and metadata, but event granularity is specific to events within distributed architectures. Table 2-61 shows a pattern in the event granularity context.

Table 2-61. Pattern that exists within the event granularity context

Name	Chapter
Contextual Events pattern	Chapter 12

High-Volume Processing

The *high-volume processing* context appears in architectures that require large amounts of data and/or high message throughput. Table 2-62 shows patterns in the high volume processing context.

Table 2-62. Patterns that exist within the high-volume processing context

Name	Chapter
Multi-Broker pattern	Chapter 25
Supervisor Consumer pattern	Chapter 33
Thread Delegate pattern	Chapter 34

Integration Architecture

Integration architecture refers to any architecture that must combine two existing systems regardless of size. In fact, architects may have many microservice architectures in an integration architecture, but this context generally refers to system-to-system integrations. Table 2-63 shows a pattern in the integration architecture context.

Table 2-63. Pattern that exists within the integration architecture context

Name	Chapter
Overcentralization antipattern	Chapter 28

Iterative Architecture

The *iterative architecture* context appears in architectures that support iterative addition and maintainability. In other words, these architectures support the gradual addition of new features without disrupting what's already there. Table 2-64 shows a pattern in the iterative architecture context.

Table 2-64. Pattern that exists within the iterative architecture context

Name	Chapter
Stovepipe Architecture antipattern	Chapter 32

Layered Architecture

The *layered architecture* context describes an architectural style that separates components in the architecture into distinct layers, each with a *technical* responsibility. For example, the traditional layered monolithic architecture includes presentation, business, and persistence layers. Typically, these architectures are *not* split based on domain considerations but rather use technical partitioning, contrasting these contexts with more domain-centric ones. Table 2-65 shows a pattern in the layered architecture context.

Table 2-65. Pattern that exists within the layered architecture context

Name	Chapter
Architecture Sinkhole antipattern	Chapter 5

Message-Based Architectures

Similar to event-driven architectures, *message-based architectures* are distributed architectures that use messages (rather than events) for communication. Table 2-66 shows patterns in the message-based architectures context.

Table 2-66. Patterns that exist within the message-based architectures context

Name	Chapter
Ambulance pattern	Chapter 3
Async Chain of Persistence pattern	Chapter 6
Cache Aside pattern	Chapter 8
Choreography pattern	Chapter 9
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Compensating Updates pattern	Chapter 11
Cross-Cutting State pattern	Chapter 13
Data-Based Event Contract pattern	Chapter 14
Domain-Broker pattern	Chapter 15
Explicit Workflow State pattern	Chapter 17
Front Controller pattern	Chapter 19
Implicit Workflow State pattern	Chapter 22
Key-Based Event Contract pattern	Chapter 23

Name	Chapter
Multi-Broker pattern	Chapter 25
Service Sharding pattern	Chapter 30
Stamp Coupling antipattern	Chapter 31
Supervisor Consumer pattern	Chapter 33
Thread Delegate pattern	Chapter 34
Workflow Event pattern	Chapter 35

Microservices

The *microservices* architecture style is one of the most popular recent styles, combining the philosophy of Domain-Driven Design (DDD), including data fidelity, with modern engineering practices. Table 2-67 shows patterns in the microservices context.

Table 2-67. Patterns that exist within the microservices context

Name	Chapter
Cache Aside pattern	Chapter 8
Choreography pattern	Chapter 9
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Compensating Updates pattern	Chapter 11
Cross-Cutting State pattern	Chapter 13
Data-Based Event Contract pattern	Chapter 14
Domain-Broker pattern	Chapter 15
Event Sourcing pattern	Chapter 16
Explicit Workflow State pattern	Chapter 17
Front Controller pattern	Chapter 19
Grains of Sand pitfall	Chapter 21
Implicit Workflow State pattern	Chapter 22
Key-Based Event Contract pattern	Chapter 23

Name	Chapter
Microservice All the Things pitfall	Chapter 24
Multi-Broker pattern	Chapter 25
Nexus Architecture pattern	Chapter 26
Orchestration pattern	Chapter 27
Read-Through/Write-Through Cache pattern	Chapter 29
Service Sharding pattern	Chapter 30
Stamp Coupling antipattern	Chapter 31
Supervisor Consumer pattern	Chapter 33
Workflow Event pattern	Chapter 35

Modularity

Modularity refers to the distribution of components, services, entities, or other software artifacts. Many patterns affect modularity, even improving it or disrupting it. Table 2-68 shows patterns in the modularity context.

Table 2-68. Patterns that exist within the modularity context

Name	Chapter
Grains of Sand pitfall	Chapter 21
Microservice All the Things pitfall	Chapter 24
Service Sharding pattern	Chapter 30

Monolithic Architectures

Monolithic architectural styles are the opposite of distribution architectures. A monolithic architecture is typically a single physical deployment (such as an EAR file in Java or an assembly in C#). Included in this category are the layered architecture, modular monolith, microkernel, and pipes and filters architectural styles. Table 2-69 shows patterns in the monolithic architectures context.

Table 2-69. Patterns that exist within the monolithic architectures context

Name	Chapter
Architecture by Implication antipattern	Chapter 4
Architecture Sinkhole antipattern	Chapter 5
Backend for Frontend pattern	Chapter 7
Command Query Responsibility Segregation (CQRS) pattern	Chapter 10
Event Sourcing pattern	Chapter 16
Read-Through/Write-Through Cache pattern	Chapter 29
Stovepipe Architecture antipattern	Chapter 32

Multiple User Interfaces

Some user experience patterns make it easier to support *multiple user interfaces*, a common requirement. For example, architects often need to support a web and several smartphone user interfaces, along with other data formats like JSON. Table 2-70 shows a pattern in the multiple user interfaces context.

Table 2-70. Pattern that exists within the multiple user interfaces context

Name	Chapter
Backend for Frontend pattern	Chapter 7

Service Granularity

Service granularity refers to the level of modularity and data separation for individual services. It also encapsulates types of coupling at the service level, including data sources or shared libraries. Granularity defines the size and scope of services in a microservices architecture. Table 2-71 shows patterns in the multiple user interfaces context.

Table 2-71. Patterns that exist within the service granularity context

Name	Chapter
Grains of Sand pitfall	Chapter 21
Microservice All the Things pitfall	Chapter 24

Workflow

The *workflow* context describes multiple entities (components, services, systems, and so on) collaborating toward some behavioral goal. For example, to build a book-purchasing system, `Customer`, `Catalog`, and `Order` entities of some kind must collaborate toward a goal — we refer to that collaboration as *workflow*.

This is one of the most common utilitarian practices of any distributed architecture, as architects build services or event processors with varying

degrees of granularity that need to collaborate toward a systemic goal. Table 2-72 shows patterns in the workflow context.

Table 2-72. Patterns that exist within the workflow context

Name	Chapter
Choreography pattern	Chapter 9
Compensating Updates pattern	Chapter 11
Cross-Cutting State pattern	Chapter 13
Explicit Workflow State pattern	Chapter 17
Front Controller pattern	Chapter 19
Implicit Workflow State pattern	Chapter 22
Orchestration pattern	Chapter 27
Service Sharding pattern	Chapter 30
Workflow Event pattern	Chapter 35

Chapter 3. Ambulance Pattern

Imagine driving down a road when suddenly the blaring sound of an ambulance siren from behind fills the air. Naturally, drivers pull over to let the ambulance pass. Why? Because the ambulance has *priority* over other vehicles on the road. By the same token, certain requests in a system might have a higher priority over others — such as a service shutdown request, an order cancellation request, or an emergency notification in a medical monitoring system. These requests need to get ahead of other normal requests in the system so they can be processed *without delay*.

The Ambulance pattern prioritizes messages in a way that allows high-priority messages to be processed right away, while allowing normal requests to flow through the system without delay. Table 3-1 summarizes the metadata for the Ambulance pattern.

Table 3-1. Metadata for the Ambulance pattern

CATEGORY	METADATA
ALSO KNOWN AS:	Carpool pattern
CONTEXT:	Event-Driven Architecture, Message-Based Architectures
IMPROVED:	Responsiveness, Throughput
DEGRADED:	Message Processing Order, Simplicity

How Do I Prioritize My Messages?

Since this pattern applies to both events and messages, for brevity we'll refer to both of them as *messages* throughout the chapter.

Most developers will typically prioritize messages by specifying the priority in the `message priority` header property of the message. For example, in Java Message Service (JMS), the priority range is from 0 to 9, with 4 as the default. Developers can send a high-priority message (such as a priority level of 8) in the `send` method of the `Producer` object:

```
producer.send(message, DeliveryMode.PERSISTENT, 8);
```

Similarly, .NET developers using the Advanced Message Queueing Protocol (AMQP) can first specify a priority range using the `x-max-priority` property (in this case 10), and then set the `Priority` property in the message header:

```
IDictionary <String , Object> args = new
Dictionary<String,Object>() ;
args.Add("x-max-priority", 10); ❶
channel.QueueDeclare("order_queue", true, false, true, args);

var properties = channel.CreateBasicProperties();
properties.Priority = Convert.ToByte(8); ❷
channel.BasicPublish("", "order_queue", properties, body);
```

❶ This property allows the developer to set the granularity of the priority level. The range for priority messages is 1 to 255 (with 0 being no priority), although most developers will specify a maximum priority level of 10 (as illustrated in our example) to simplify message priority choice.

❷ This priority header property is for a specific message and is set to a value within the range specified by the `x-max-priority` property (1-10), which in this case is 8.

While this simple technique works when sending only an occasional high-priority message, it can quickly cause problems when people are sending lots of high-priority messages — and here's why.

Most queues support "first in, first out" (FIFO). This guarantees the message order in the queue. However, a high-priority message must be placed at the head of the queue and delivered *before* normal-priority messages, as shown in Figure 3-1. This means that normal-priority messages have to wait until all of the high-priority messages are processed — much like drivers pulling over to let an ambulance go by.

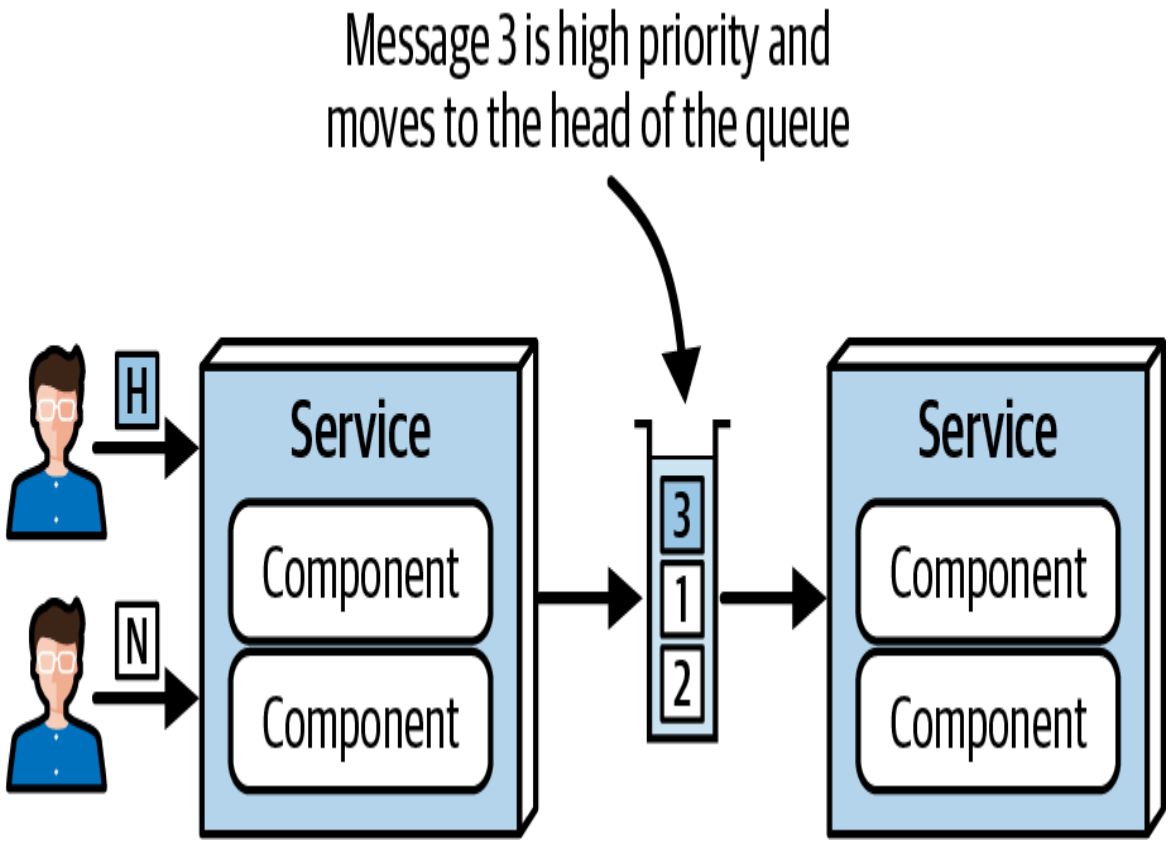


Figure 3-1. High-priority messages move to the head of the queue

Now, imagine what would happen if a long series of ambulances, fire trucks, and police cars forced all the drivers off the road. Those drivers would be waiting a very long time for all the emergency vehicles to pass, which might make them arrive late to their destinations. Similarly, a message producer sending a large number of high-priority messages will cause the normal-priority messages to wait in the queue, just like vehicles

on the side of the road. This scenario, illustrated in Figure 3-2, degrades the responsiveness of the normal-priority message processing. In some cases, those requests might even time out if producers are waiting for a response.

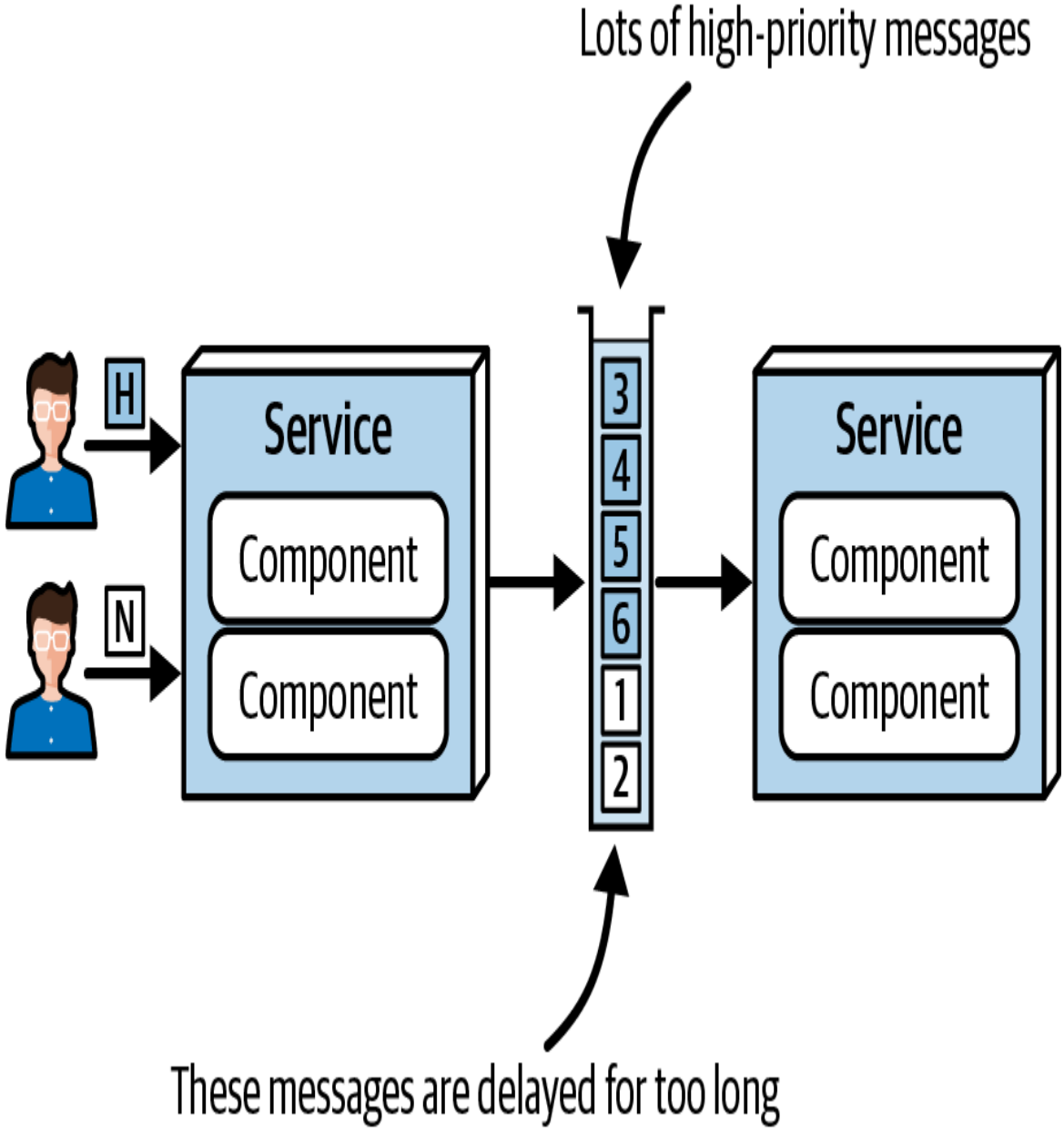


Figure 3-2. Too many high-priority messages will delay normal messages

The Ambulance pattern avoids this issue by allowing some events a higher priority, yet maintaining responsiveness for all other events.

Applying the Pattern

Rather than utilizing the message-priority property in the header of the message, the Ambulance pattern uses two separate queues — one for high-priority messages and one for normal-priority messages — as shown in Figure 3-3.

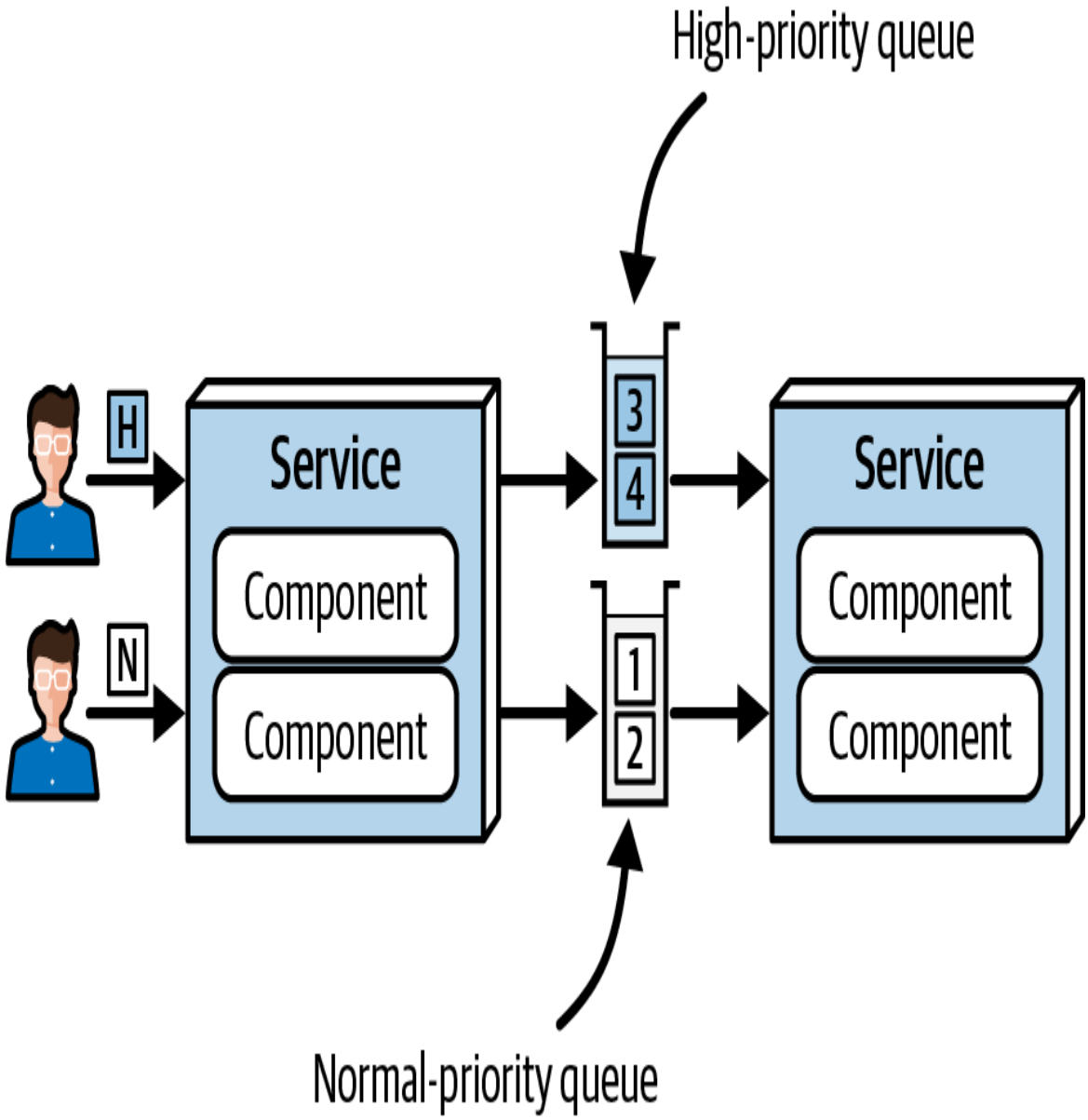


Figure 3-3. High-priority messages are sent to a different queue than the normal-priority ones

This pattern requires switching logic in the message producer to send a message to one of two queues: a priority queue and a normal queue. The

following pseudocode illustrates how this works:

```
var message = create_message("this is a high-priority message",
true)
if (message.high_priority)
 send_to_priority_queue(message)
else
 send_to_normal_queue(message)
end if
```

On the message consumer side, the consuming service needs to have two threads: one to consume high-priority messages from the priority queue, and one to process normal-priority messages from the normal queue, just like having a dedicated lane for emergency vehicles when they come through. This allows the consuming service to continue processing normal-priority messages, while at the same time processing high-priority messages when they come in. This avoids delays for normal-priority messages:

```
var processor = create_message_processor(broker_connection)
create_normal_thread(processor).start();
create_priority_thread(processor).start();
```

Now that priority messages are processed concurrently with normal messages, it's possible for the sending service to provide a further level of granularity by giving some high-priority messages a higher priority than others. Here, developers can use the message priority header on high-priority messages, as shown in the beginning of the chapter. Of course, if too many higher-priority messages flood the priority queue, other lower-priority messages will have to wait in the priority queue. However, using this pattern, normal messages will still be processed, regardless of the message priority granularity.

With this pattern, no high-priority messages are waiting; *both* threads process normal-priority messages, like all drivers sharing the same road until an ambulance comes through. As illustrated in Figure 3-4, the consuming service reuses the priority message thread when the priority

queue is empty, allowing for more normal messages to be processed concurrently.

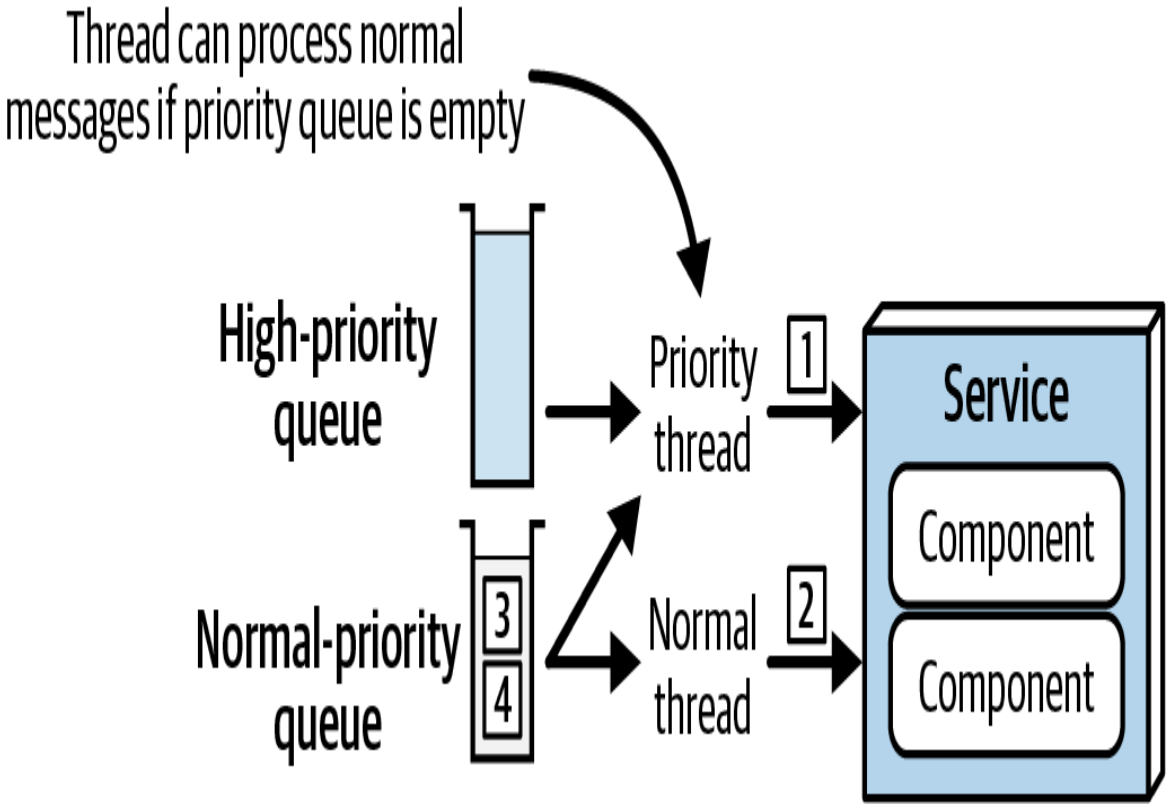


Figure 3-4. The priority thread processes normal messages when it's idle

The priority thread always first checks the priority queue for messages. If it's empty, it reads the next message from the normal-priority queue, as shown in the following pseudocode for priority thread processing:

```
var message = null
if (priority_queue_empty)
 message = receive_message(normal_queue)
else
 message = receive_message(priority_queue)
end if
```

One disadvantage of the Ambulance pattern is that if a high-priority message arrives while the priority thread is processing a normal-priority message, the high-priority message might have to wait. For example, as illustrated in Figure 3-4, if the priority thread reads message 1 from the normal queue and immediately afterward a high-priority message appears in

the priority queue, that priority message will have to wait until the priority thread is done processing the normal message. Depending on the urgency of the system's high-priority messages, architects may consider this a downside of the pattern.

The other disadvantage is that while the Ambulance pattern preserves FIFO, it doesn't preserve the *processing order* of messages. Look at the scenario illustrated in Figure 3-4. The priority thread sees that the priority queue is empty, so it pulls message 1 from the normal queue and starts processing it. At the same time, the normal thread pulls message 2 from the normal queue and starts processing it. Then, the normal thread finishes processing message 2 *before* the priority thread finishes processing message 1, thus mixing up the message *processing* order.

This change in processing order is generally not an issue for read requests. However, for update requests, it could be. Let's say the messages in this scenario come from a customer who places an order (message 1), but then immediately cancels it (message 2). What happens if the cancellation request in message 2 is completed before message 1, which places the order, has been processed? We discuss how to address this problem and guarantee message *processing* order in the related Thread Delegate pattern, described in Chapter 34.

If it's crucial to maintain message processing order and process priority messages *immediately* when they enter the priority queue, architects can choose to use the Carpool variant of this pattern.

Many cities around the world have highways with a dedicated "carpool" or high-occupancy vehicle (HOV) lane. This strategy encourages commuters to ride together, allowing them to get into the city faster while at the same time reducing overall carbon emissions. Similarly, in the Carpool variant of this pattern, shown in Figure 3-5, high-priority messages have a *dedicated* priority thread and are processed immediately after they're sent to the queue.

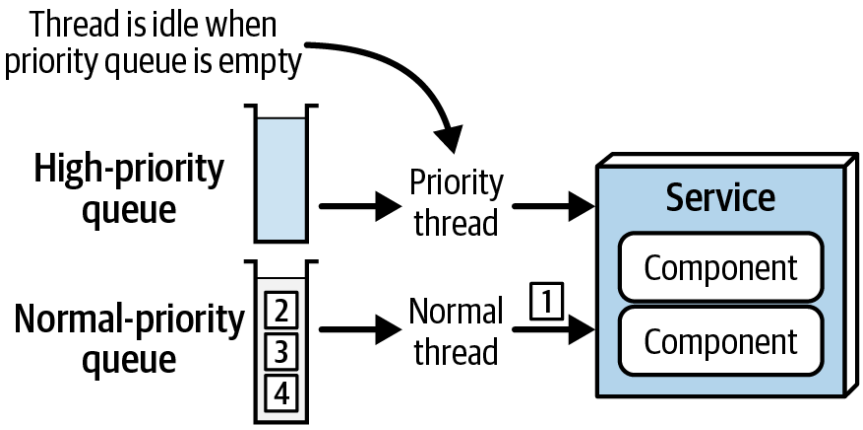


Figure 3-5. With the Carpool variant, the priority thread only processes high-priority messages

When using the Carpool variant, high-priority messages are not delayed when they come in. The dedicated priority thread, standing idle, receives the high-priority message and processes it without delay, while the normal thread is processing normal messages. Also, this variant preserves the processing order of normal-priority messages, since only a single thread processes them.

One disadvantage of this variant is that it slows the processing of normal messages by keeping an idle thread standing by that *could* be processing additional messages. This disadvantage can be addressed by the related Supervisor-Consumer pattern, described in Chapter 33, where a group of concurrent consumer threads within a single service can process multiple requests.

Trade-Offs

While we've demonstrated some of the advantages of this pattern in terms of throughput and responsiveness, architects should be aware of the trade-offs involved in using this pattern, as outlined in the following sections.

Pros

Overall message throughput and responsiveness are the primary advantages of this pattern. The Carpool variant, while preserving message processing order, is a bit less responsive than its Ambulance counterpart because the

priority thread doesn't process normal messages — only priority ones. The same is true with overall throughput: the Carpool variant will have less than its Ambulance counterpart.

Analyze the frequency and number of high-priority messages before choosing to adopt the Ambulance pattern, and weigh the disadvantages outlined in the next section to ensure that this is the most appropriate pattern to use for handling message priority.

Cons

This pattern does add complexity due to using multithreading and multiple queues. Because of this, we recommend avoiding this pattern for situations involving infrequent and nonbursty high-priority messages. In these cases, the priority message header property technique works and is much easier to implement and maintain.

If teams require message processing order for update requests to be preserved, we recommend adopting the Carpool variant of this pattern, or skipping the Ambulance pattern altogether and using the standard message header properties to set priority. Note that preserving message processing order is usually not a concern for read requests, so this pattern works well in those situations.

Comparison Chart

Table 3-2 summarizes the star ratings associated with applying this pattern. It compares them using priority through a standard message header (without the pattern), applying the Ambulance pattern, and applying the Carpool variant.

Table 3-2. Comparison chart for the Ambulance pattern

Capability	Without Pattern	Ambulance Pattern	Carpool Variant
IMPROVED			
Responsiveness	★★	★★★★★	★★★★
Throughput	★★	★★★★★	★★★★
DEGRADED			
Message Processing Order	★★★★★	★	★★★★★
Simplicity	★★★★★	★★★	★★★

The following sections describe each of the capabilities found in Table 3-2.

Improved

The capabilities that are improved when applying this pattern include:

Responsiveness

> The main advantage (and the purpose) of this pattern is to provide consistent responsiveness for normal-priority messages while allowing high-priority messages to be processed as soon as possible, so that a single service can process up to twice as many normal-priority messages (without using the Carpool variant). Using the Carpool variant still improves responsiveness because the system can process high-priority messages at the same time as normal-priority ones, but it only

improves by two stars (rather than three) because each message type (priority and normal) is single-threaded.

Throughput

By utilizing multiple threads, this pattern allows consuming services to process both normal and priority messages concurrently, improving overall message throughput. Using this pattern without the Carpool variant further improves overall throughput because normal-priority messages can be processed concurrently by both threads.

Degraded

The capabilities that are degraded when applying this pattern include:

Message Processing Order

Using this pattern can disrupt message processing order because normal-priority messages are processed concurrently by both the high-priority thread and the normal-priority one. While this generally isn't an issue for query requests, it does present issues for update requests. Using the Carpool variant of this pattern guarantees the processing order of normal-priority messages, because the service only uses the normal-priority thread. However, as noted, this variant doesn't improve responsiveness and throughput as much as the Ambulance pattern does.

Simplicity

To implement this pattern, architects must create a separate priority queue, modify message producers to add switching logic to send to multiple queues, and add logic in the message consumer to create two separate message-processing threads (one for normal priority and one for high priority). Because this pattern adds complexity, using it might be overkill if message producers only send a few occasional high-priority messages. In these cases, architects might want to consider using the simpler message priority header property technique described at the start of this chapter.

Influences

No other outside factors influence this pattern or change the star ratings.

How to Govern This Pattern

By measuring and continuously monitoring the frequency, number, and "bursts" of high-priority messages, architects can determine whether this pattern would be useful in their system. If the metrics show a low number of high-priority messages with few or no "bursts," implementing this pattern would likely be overkill. However, if metrics show a high number of priority messages that arrive in bursts, this might be a good pattern to use.

Another way to govern this pattern and determine whether the Ambulance or Carpool variant would be most appropriate is to measure and track delays when processing high-priority messages. Because the high-priority thread in the Ambulance pattern is used for normal-priority messages when idle, a high-priority message may have to wait until the normal-priority message has been processed. Poor response times for high-priority messages might indicate that architects should consider the Carpool variant, particularly for normal-priority messages that take a while to process. To improve overall responsiveness and throughput, consider also implementing the related Supervisor-Consumer pattern, described in Chapter 33.

Related Patterns, Antipatterns, and Pitfalls

The following patterns are related to the Ambulance pattern:

Supervisor-Consumer pattern (see Chapter 33)

> This pattern is about leveraging multiple expandable threads within a single service so that it can process multiple requests at once. The Ambulance pattern leverages this concept by creating two threads: a priority thread and a normal thread. The Supervisor-Consumer pattern can be combined with the Ambulance pattern to provide parallel processing for normal and priority messages.

Thread Delegate pattern (see Chapter 34)

This complex pattern allows update-related messages to be *processed* in order. It can be used in conjunction with the Ambulance pattern to preserve message-processing order when the priority thread is processing normal-priority messages. If message-processing order is important for update requests but additional throughput isn't needed, we recommend first trying the Carpool variant of this pattern.

Chapter 4. Architecture by Implication Antipattern

When development teams receive requirements for a new system or product, they often simply start developing source code without creating (or even thinking about) an underlying software architecture for it. The Architecture by Implication antipattern describes situations where software is written with an implied architecture (or none at all), often resulting in a failed system. This chapter describes when it's OK to proceed without an underlying architecture, and when doing so can lead to complete failure.

Table 4-1 describes the metadata associated with this antipattern. Because this chapter is dealing with an antipattern and not a pattern, the Improved metadata category refers to the capabilities that are improved when adopting this antipattern, and the Degraded metadata category refers to the capabilities that are degraded when adopting this antipattern.

Table 4-1. Metadata for the Architecture by Implication antipattern

CATEGORY	METADATA
CONTEXT:	Agentic Code Generation, Architectural Styles, Monolithic Architectures
IMPROVED:	Cost, Initial Time to Market
DEGRADED:	Adaptability, Agility, Availability, Data Consistency, Data Integrity, Deployability, Elasticity, Error Handling, Evolvability, Fault Tolerance, Flexibility, Maintainability, Modularity, Responsiveness, Scalability, Simplicity, Testability, Throughput, Workflow

When Do I Need an Architecture?

Software architecture defines the *structure* of a system, much in the same way as an architecture defines an office building. However, the concept extends far beyond just structure. Software architecture is what provides the *capabilities* of a system or product as well: operational capabilities, such as scalability, responsiveness, availability, fault tolerance; and even process-related capabilities such as maintainability, testability, and deployability. The *functionality* of a system, however, describes its *behavior*.

Certain capabilities, such as scalability and elasticity, can only be solved through architecture, meaning that the architecture of a system is critical to its success. However, suppose the system has no definable architecture and the team just starts developing source code to implement the desired *behavior*. Chances are, the system will fail to meet the business's needs and goals.

The term *architecture by implication* means that the system *has* some sort of underlying architecture, but it's implied rather than well thought out and defined as an architecture. Typically, most teams starting out without a well-planned architecture will develop a layered or modular monolith architecture, or (in the case of distributed architectures) something that resembles a cross between a service-based architecture and microservices.[1] This usually leads to an unstructured spaghetti mess or Big Ball of Mud architecture that fails to satisfy any of the needed architectural capabilities.

Why Is This an Antipattern?

Not every system needs a well-defined architecture. If a pet owner wants to build a simple doghouse in their backyard, they don't need an architecture: all they need is some wood, a bunch of nails, and a cute dog. However, if a company wants to build a new 10-story office building, it needs an architecture. By the same token, if a children's soccer coach wants to put together a quick registration website for the upcoming season, those few web pages don't need an architecture. However, if a financial services company needs a new system to support its backend settlement and reconciliation operations, it absolutely needs a well-defined and well-structured architecture.

As we mentioned in the introduction to this book, an *antipattern* is something that sometimes works at the start, but is likely to lead to failure later. The Architecture by Implication antipattern fits this definition perfectly. The soccer registration page we just mentioned is a great example. Putting up the initial website doesn't need an architecture.

But suppose other soccer clubs in the area start using the same website. It becomes popular and expands to support soccer registrations across the entire state, adding behavior to manage payments, donations, equipment allocations, and so on. Suddenly this simple website without a well-defined and structured architecture has become a large, complex system that will likely struggle with scalability, throughput, and responsiveness, not to mention the difficulty of applying changes to and testing such an

unstructured system. This kind of scenario is how Architecture by Implication becomes an antipattern.

To illustrate another example of how this antipattern can occur, consider the Sysops Squad architectural kata, used throughout our book *Software Architecture: The Hard Parts*. The Sysops Squad is a trouble-ticket system where customers can report issues with their electronic equipment. Customer-facing experts in the field (that's the Sysops Squad) come to their home or office to fix the problem. Figure 4-1 shows the basic flow for this system.

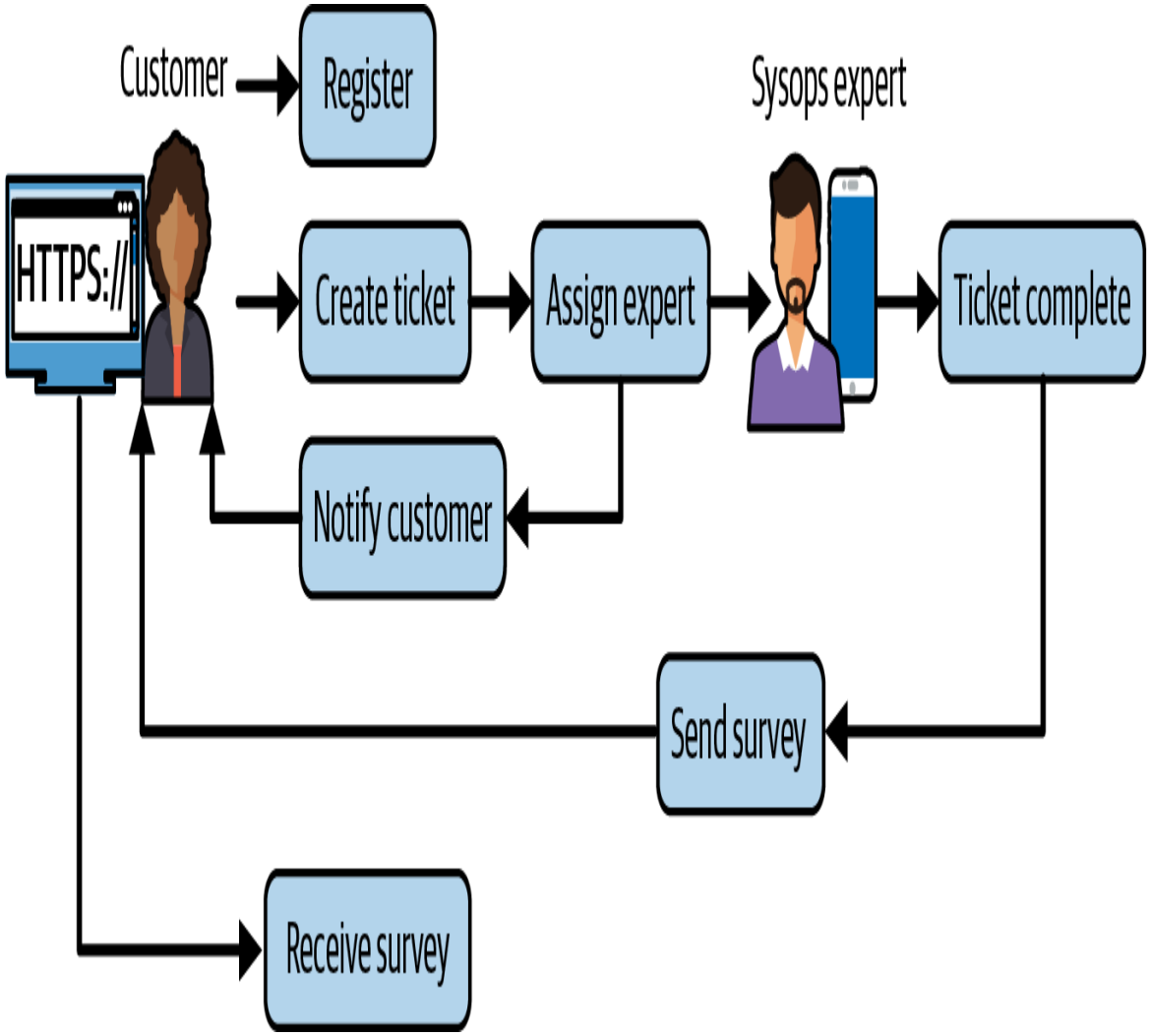


Figure 4-1. The basic workflow for the Sysops Squad system

In looking at this workflow, an architect might envision the logical architecture (internal structure of the system) illustrated in Figure 4-2. It

consists of three domains — ticketing, customer, and survey — each of which has its own architectural components and subdomains.

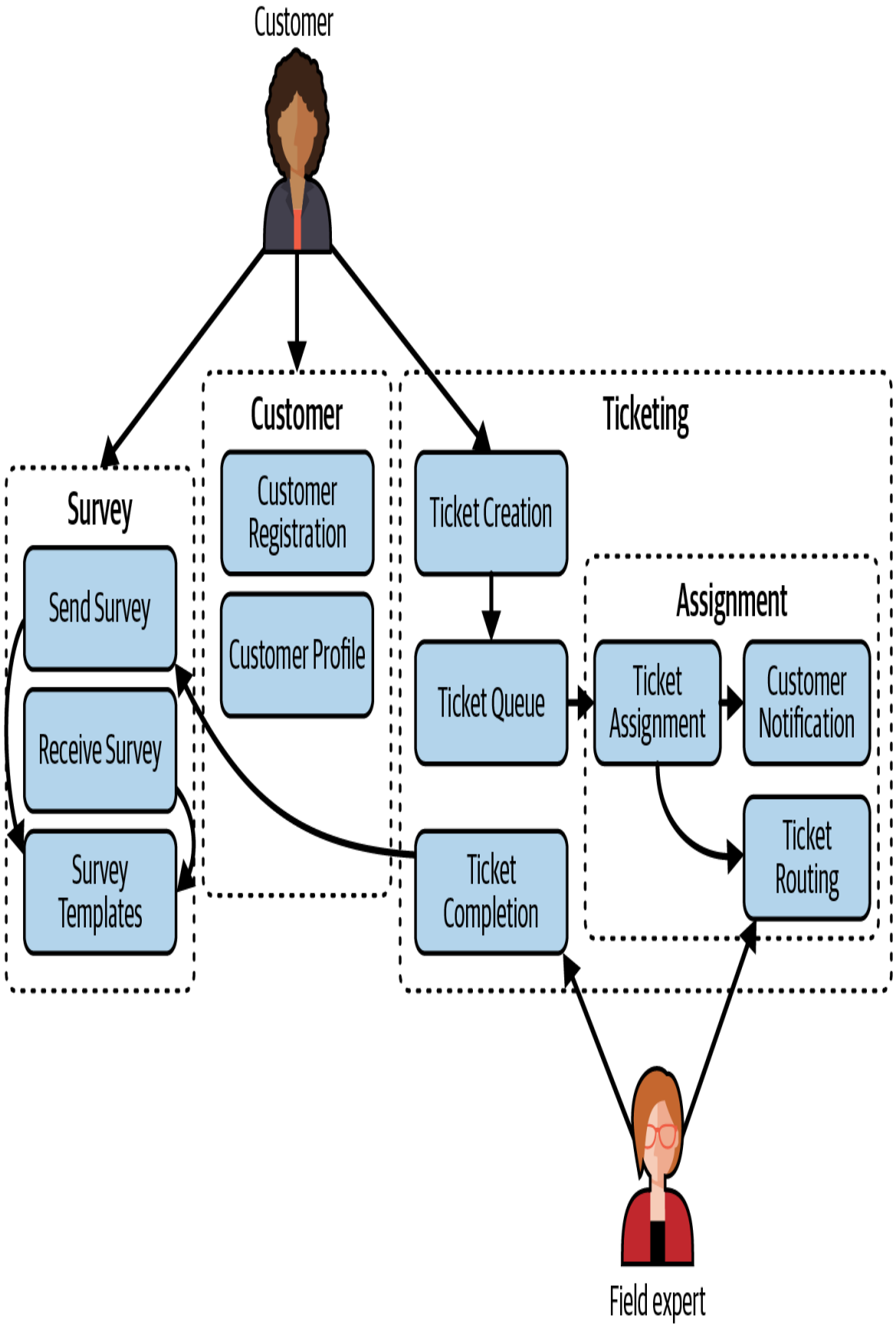

Customer
Customer
Ticketing
Survey
Customer Registration
Ticket Creation
Send Survey
Assignment
Customer Profile
Ticket Queue
Ticket Assignment
Customer Notification
Receive Survey
Ticket Routing
Survey Templates
Ticket Completion
Field expert

Figure 4-2. A well-structured architecture for the Sysops Squad system

The `Ticketing` domain consists of three components (`Ticket Creation`, `Ticket Queue`, and `Ticket Completion`) as well as an `Assignment` subdomain. `Assignment` includes components for assigning an expert to a ticket (`Ticket Assignment`, `Customer Notification`, and `Ticket Routing`).

The `Customer` domain has two components (`Customer Registration` and `Customer Profile`), and the `Survey` domain has three (`Send Survey`, `Receive Survey`, and `Survey Templates`). This architecture is well-defined and structured, it's easy to follow and understand, and it provides good levels of the following architectural capabilities:

Maintainability

The ease with which development teams can locate and apply coding changes or add new features

Testability

The ease and completeness of testing

Deployability

The ceremony involved with deploying software, the frequency of deployment, and the overall risks of deploying software

Adaptability

The ease with which the system can adapt to changes in technologies, frameworks, platforms, languages, and infrastructure while keeping the same behavior

Evolvability

The ease with which the architecture can be extended to add new features and behavior

However, what if the development team ignored this architecture and just started implementing the system *without* an architecture (or with an *implied* unstructured architecture), thus falling into the Architecture by Implication antipattern? If that were the case, the resulting architecture might look like the one shown in Figure 4-3.

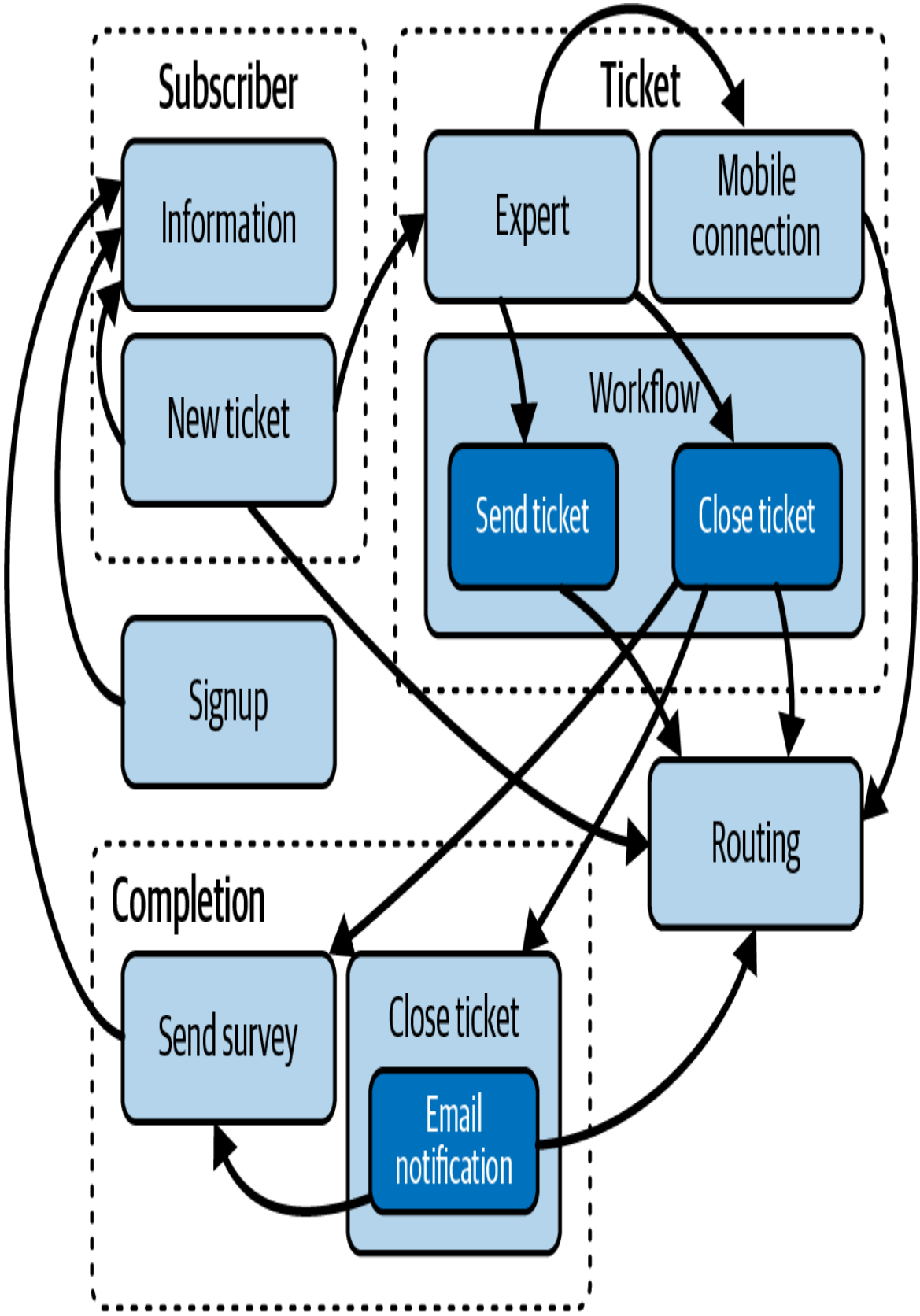


Figure 4-3. An implied architecture for the Sysops Squad system

In comparing these two architectures, it's quite obvious that the well-structured one shown in Figure 4-2 is much better, whereas the implied one would struggle to achieve the architectural capabilities listed above. The system's *behavior* might work, but the system would have poor (or no) *capabilities* such as those listed above.

Architecture by Implication can also become a *pitfall* when a team starts building large, complex systems from scratch without an architecture. While it may be tempting to start writing source code to get the system out the door quickly, such a system will fail to meet needed operational capabilities such as scalability and responsiveness, as well as process-related capabilities like maintainability and testability. As the name "pitfall" suggests, this is never a good idea, even from the start.

Most systems fall, in size and complexity, somewhere between the two extremes of a children's soccer registration website and an enterprise financial settlement system. So how do architects and development teams know when a system needs an architecture?

How to Avoid This Antipattern

Avoiding the Architecture by Implication antipattern involves understanding what architectural capabilities the system needs. If the system is simple and has no identifiable architectural capabilities, then chances are it doesn't need a well-defined and structured architecture. However, if the system *does* require specific architectural capabilities — such as scalability, availability, fault tolerance, responsiveness, and so on — then that system needs definition and structure to achieve those operational capabilities. The same is true with process-related capabilities such as maintainability, testability, and deployability; these capabilities would be hard to achieve without a well-defined architecture.

In many cases, teams start out with a small, simple system that doesn't have any critical architectural capabilities (and hence no concern for defining an architecture), but as that system grows and adds features, it becomes hard to maintain, test, and deploy. In these cases, architects can *refactor* to begin

transforming the unstructured architecture into one that is structured. Specifically, the flattening architectural refactoring we describe in Chapter 5 of the book *Software Architecture: The Hard Parts* is a great place to start for architects who want to get out of this antipattern.

Trade-Offs

Because Architecture by Implication is an antipattern, it could have some positive trade-offs, particularly for small, simple systems. However, as systems get larger and need more capabilities, the negative trade-offs far outweigh the positive ones, which is why this is an antipattern.

Pros

The advantages of adopting this antipattern include:

Lower cost and time

> Developing a well-defined and well-structured architecture takes time, and is therefore more costly than proceeding without an architecture. With our example soccer registration website, the pros (lower costs, less time) far outweigh the need for system capabilities as defined by architectural capabilities. However, for systems that *do* require specific critical capabilities, the cons would far outweigh the pros in this case.

Cons

The disadvantages of adopting this antipattern include:

Lack of architectural capabilities

> A software architecture defines the system's overall logical and physical structure and provides capabilities such as scalability, responsiveness, availability, maintainability, and testability. Systems without a well-defined architecture, even when their behavior works perfectly fine, will find significant challenges in meeting these capabilities and often fail. A

large order-entry system that only scales to five concurrent users is of no use to anyone.

Comparison Chart

Table 4-2 shows the qualitative star ratings for various factors when falling into versus when avoiding this antipattern. Architects should consider the context and capabilities of the system (such as a small, simple website versus a critical, complex business system), as well as their needs for critical architectural capabilities, when deciding whether to fall into this antipattern or not. In this chart, good attributes start at a five-star rating and negative consequences reduce the five stars.

Table 4-2. Comparison chart for the Architecture by Implication antipattern

Capability	Using Antipattern	Avoiding Antipattern
IMPROVED		
Cost	$$	$$$$$
Initial Time to Market	★★★★★	★★
DEGRADED		
Adaptability	★	★★★★★
Agility	★	★★★★★
Availability	★	★★★★★
Data Consistency	★	★★★★★
Data Integrity	★	★★★★★
Deployability	★	★★★★★
Elasticity	★	★★★★★
Error Handling	★	★★★★★
Evolvability	★	★★★★★

Capability	Using Antipattern	Avoiding Antipattern
Fault Tolerance	★	★★★★★
Flexibility	★	★★★★★
Maintainability	★	★★★★★
Modularity	★	★★★★★
Responsiveness	★	★★★★★
Scalability	★	★★★★★
Simplicity	★	★★★★★
Testability	★	★★★★★
Throughput	★	★★★★★
Workflow	★	★★★★★

The following sections describe each of the capabilities found in Table 4-2.

Improved

The capabilities that are improved when adopting this antipattern include:

Cost

One of the reasons teams migrate toward this antipattern is because it's less expensive to implement a system without worrying about architectural capabilities. However, while the *initial* costs of

implementing a system might be lower, over time the system will become harder to maintain, thus increasing costs associated with adding features (see the related Stovepipe Architecture antipattern in Chapter 32). By adopting this antipattern and not including an architecture, teams will lower costs by a magnitude of three dollar signs.

Initial Time to Market

Teams can get a system implemented and out the door much faster by focusing solely on the behavior, ignoring any architectural capabilities that might be needed. Architecture requires planning, and as such takes additional time and effort. Like cost, by adopting this antipattern teams will see a three-star increase in the initial time to market, but over time this advantage quickly degrades as the system becomes more complex.

Degraded

The capabilities that are degraded when adopting this antipattern include:

Adaptability

Systems that undergo frequent change by adding new behavior on top of existing behavior typically need a well-formed structure in place to prevent creating a big-ball-of-mud architecture. By adopting this antipattern, teams will find it increasingly difficult to augment existing behavior in the system.

Agility

While *initial* time to market is improved by ignoring an architecture, over time teams will find it increasingly difficult to maintain, test, and deploy changes. Without an architecture in place, overall agility — the ability to respond quickly to change — will quickly degrade, significantly slowing teams down when applying changes to the system.

Availability

Making a system operational requires planning. Without an architecture in place, it's difficult to ensure a reasonable level of availability.

Data Consistency

This capability is mostly related to distributed systems, but monolithic systems can also have data consistency issues without a well-defined structure, particularly when performing multiple updates to a database. Coordinating a transaction to ensure that all data is either updated or rolled back requires a solid architectural solution. Without it, expect data consistency issues.

Data Integrity

Unlike data consistency, *data integrity* refers to the correctness of the data. It's possible for data to be consistent yet incorrect. Systems without a well-defined architecture often suffer from data integrity and correctness issues.

Deployability

Deploying software is a highly error-prone phase of the software development lifecycle, and typically carries with it high levels of risk that something will break. Side effects, forgotten libraries, missing data migration scripts, and hidden configuration settings can all cause a failed deployment. Having a well-defined architecture in place can significantly reduce the level of risk and ceremony involved with deploying software.

Elasticity

Elastic systems — those that have highly variable user load — require an architecture that's suited for this type of condition, with the focus being on the Mean Time to Start (MTTS) of services within that system. Large, coarse-grained services (and even monolithic systems) typically have an MTTS measured in minutes, whereas small, fine-grained microservices typically have an MTTS measured in milliseconds. A

system responding to an immediate spike in user load requires services to immediately start up in the fastest time possible. Ad hoc systems without an architecture in place rarely achieve this goal.

Error Handling

Proper error handling, particularly in distributed systems, requires a well-defined architecture to coordinate atomic transactions and respond appropriately to error conditions. Ad hoc systems usually result in ad hoc error handling, negatively impacting workflow, data consistency, and data integrity.

Evolvability

Simple, small systems without an architecture can evolve to a certain degree, but larger, more complex systems require a well-defined architecture to evolve. Systems without an architecture typically evolve into the related Stovepipe Architecture antipattern in Chapter 32.

Fault Tolerance

This capability is about parts of the system being able to function while other parts fail, and also the amount of time it takes to recover from a fault (known as Mean Time To Recovery, or MTTR). This capability requires planning and an appropriate architecture. For example, monolithic systems have little or no fault tolerance, because a fatal error such as an out-of-memory condition causes the entire deployment unit to fail. Distributed systems, on the other hand, can have high levels of fault tolerance if an appropriate architecture is in place. Services within ad hoc systems without an architecture are typically interconnected in an ad hoc fashion, so that when one service goes down, all others are impacted.

Flexibility

Flexible systems adapt to changes in infrastructure, language versions, library version, and so on. Flexibility requires a well-formed structure in

place so that teams understand the impacts of such changes.

Maintainability

As systems get larger and more complex, they naturally become harder to maintain. Unstructured systems without an architecture make it increasingly difficult to locate bugs, change existing behavior, and add behavior. By creating an architecture, teams can identify well-defined domains, components, and services with appropriate levels of coupling, allowing them to better understand and maintain the system.

Modularity

Modularity allows teams to partition a system into logical or physical parts. Not only does this help them better understand the system, it also makes maintenance and testing easier. Modularity can only be achieved with well-planned architecture.

Responsiveness

Ad hoc systems without an architecture can sometimes be responsive, but as they change and become more complex, their responsiveness usually degrades. Like most operational capabilities, responsiveness requires planning — and, correspondingly, an appropriate architecture.

Scalability

As illustrated by the children's soccer registration website example at the start of this chapter, systems designed without regard to scalability are rarely able to scale. John Gall observed this in his famous "Gall's Law," described in "Why Is This an Antipattern?" in Chapter 32. Scalability requires upfront planning and an appropriate architecture that provides scalability, such as an event-driven architecture or even microservices. Scalability can't simply be added to a system — it requires planning.

Simplicity

While systems like the children's soccer registration website start out simple, they can quickly become complex as teams continue to add new features. This is a cautionary tale for teams that vibe code as a quick solution to get it out the door quickly.

Testability

Testability is about the ease of and completeness of testing. While ad hoc systems may be *easy* to test, the *completeness* of testing quickly degrades as those systems become less structured. Having modular systems with well-defined domains, components, and services with appropriate levels of coupling help achieve more complete testing.

Throughput

Overall throughput is a measurement of how many requests can flow through the system in the fastest way possible. Ad hoc systems that don't plan for high levels of throughput will rarely achieve this goal.

Workflow

A system with a complex workflow requires an architecture that can manage that. Architects should analyze their capability requirements and choose either Choreography (see Chapter 9) or Orchestration (see Chapter 27) to manage the workflow and its corresponding state. Choosing the most appropriate option requires a keen sense of what architectural capabilities the system needs, and therefore an architecture.

Influences

Although no outside factors influence the star rating found in Table 4-2, this antipattern can influence other areas of the technical and business environment.

Not having an architecture in place means not having a sense of the operational capabilities the system requires. This affects the corresponding

infrastructure, since infrastructure requires planning and must be aligned with the system's capabilities.

This antipattern can also influence the business's needs and goals, in terms of its direction and position in the market. For example, if the business is undergoing significant growth and expanding into other regions and markets, not having an architecture in place for the core business systems means those systems are unlikely to align with the corresponding goals of scalability, evolvability, and other operational capabilities required to grow the business.

Am I in This Antipattern?

Teams can identify whether they're in this antipattern by observing and analyzing both process-related and operational capabilities:

Process-related capabilities

> Teams can measure and track the rate of change of the system and the amount of time and effort involved in applying changes. If it takes too long to apply changes or add features, this might indicate that the system is ad hoc and lacks an appropriately structured architecture.
>
> Teams can also measure and track the number of errors reported (and unreported). If that number increases over time, the system may be unstructured and highly coupled, and would likely benefit from a well-defined architecture.

Operational capabilities

> Teams can track complaints and errors related to things like scalability, responsiveness, fault tolerance, availability, and data integrity. All of these operational capabilities require planning and a well-defined architecture. As companies grow, these capabilities become increasingly important; tracking them early and often allows teams to rewrite code or refactor the system with an appropriate architecture, as needed.

Related Patterns, Antipatterns, and Pitfalls

The following antipattern is related to the Architecture by Implication antipattern:

Stovepipe Architecture antipattern (see Chapter 32)

> This antipattern involves "bolting" additional behavior onto a software architecture so that it loses its original definition, becoming an unstructured mess over time. Many systems start out with a well-defined and well-structured architecture that changes over time into an implied architecture.
>
> The big difference between Stovepipe Architecture and Architecture by Implication is that the Stovepipe Architecture antipattern usually starts as a well-defined and well-structured architecture but turns into a Big Ball of Mud, poorly connected and heterogeneous, looking like an old-fashioned stovepipe. With Architecture by Implication, there's no well-defined architecture, even at the start.

---

[1] Refer to *Fundamentals of Software Architecture*, 2nd edition for a complete description of each of these architectural styles.

Chapter 5. Architecture Sinkhole Antipattern

The Architecture Sinkhole antipattern occurs within layered architectures when certain kinds of requests are simply passed down to the next layer without performing any business logic, validation, or transformation. This antipattern forces adherence to the *philosophy* of the architecture rather than adding any real value.

This chapter describes an *antipattern*, so we've adjusted the metadata to reflect that. In this case, the Improved metadata category lists capabilities that improve when adopting this antipattern, and the Degraded metadata category refers to capabilities that are degraded when adopting this antipattern. Table 5-1 describes the metadata associated with this antipattern.

Table 5-1. Metadata for the Architecture Sinkhole antipattern

CATEGORY	METADATA
CONTEXT:	Layered architecture, Monolithic architectures
IMPROVED:	Debuggability, Evolvability, Maintainability, Modularity, Simplicity, Testability
DEGRADED:	Initial Time to Market, Responsiveness

When Should I Open Layers in a Layered Architecture?

The Architecture Sinkhole antipattern applies to layered architectures. Within a layered architecture, components are organized by technical concern. Each layer has a very specific role and responsibility. These layers are "stacked" on top of one another, as shown in Figure 5-1.

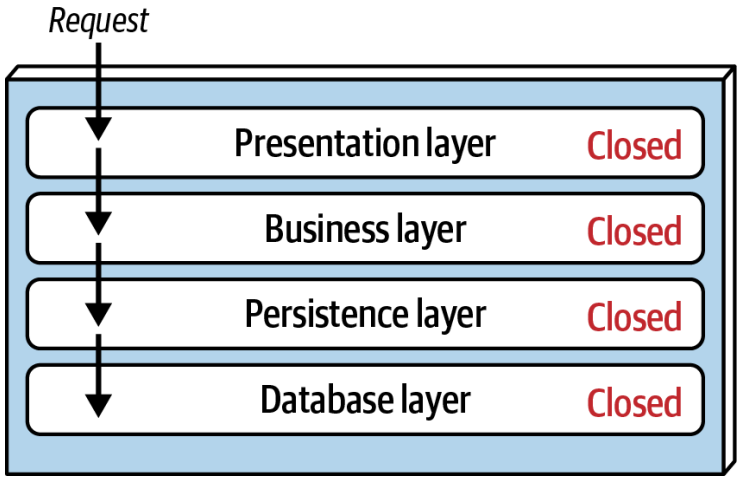


Figure 5-1. Components partitioned by technical concern and stacked on top of one another

In Figure 5-1, requests enter at the "top" layer (in this case, the Presentation layer), and are processed sequentially by layer. Each layer executes some logic or performs some validations or transformations, then passes the request to the layer underneath it. This culminates in a response, which then passes back *up* through each layer and finally back to the client. It's somewhat similar to a state machine, in which the request is transitioned as it traverses each layer.

Why Is This an Antipattern?

In a conventional layered-architecture topology, every layer is *closed* — that is, each layer is only allowed to forward the request to the layer just beneath it. Consequently, the response must traverse every layer back *up* the stack before it's sent back to the client. In other words, a request is not allowed to skip any layers, even if some layers in the stack don't contribute any meaningful processing.

The benefit of keeping layers closed is that it isolates changes. A closed layer can only interact with the layers directly above and beneath it. These

clear dependencies mean that changing a layer has a very limited ripple effect — changes affect only the layers that depend on the changed layer. This well-defined dependency graph leads to better maintainability and testability.

Since the layered architecture is technically partitioned, all these benefits culminate to make the architecture evolvable. For example, the team could consider swapping handcrafted SQL for a modern ORM framework, knowing exactly which layers would be in the "blast radius" of changes.

On the flip side, let's say the team has an application with a large number of simple read requests that only need to fetch data from the database and return it to the client. These requests need little or no processing in certain layers. Since every layer is closed, the request must still pass through every layer, although most have little logic, validation, or processing to add. This is akin to having a lot of "no-op" transitions in a state machine, which adds a lot of boilerplate code and can even affect responsiveness.

It's referred to as the Architecture Sinkhole antipattern.

How to Avoid This Antipattern

Depending on the use case, the constraints on the architecture, and the kinds of friction the team faces when working with those constraints, there are ways to avoid this antipattern.

Opening Layers

The solution to the sinkhole antipattern is to open up one or more layers. An *open* layer is one that allows the layers above and below it to bypass it and communicate directly with the next closed layer. In Figure 5-2, the Services layer is now open. This allows the Business layer to either forward the request to the Services layer when applicable (as marked by the solid straight line), or bypass it and go directly to the Persistence layer (shown by the dotted straight line).

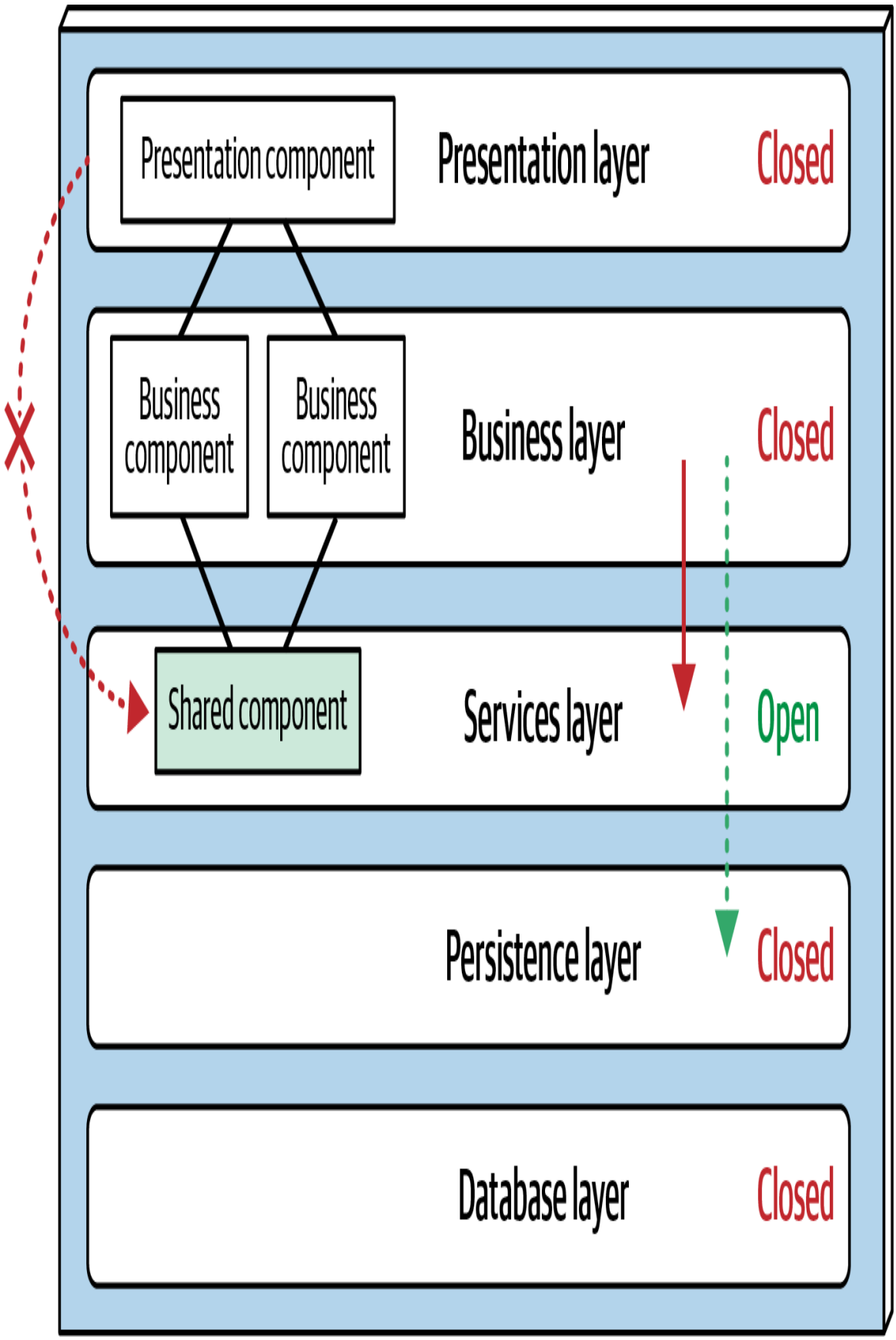

Presentation component
Presentation layer
Closed
Business component
Business component
Business layer
Closed
Shared component
Services layer
Open
Persistence layer
Closed
Database layer
Closed

Figure 5-2. Opening a layer so it can be bypassed

Using open layers avoids the Architecture Sinkhole antipattern because it means that requests can be forwarded only to the layers that add meaningful processing, rather than being forced to pass through every layer just to adhere to the philosophy of the architecture.

Consider a Different Architecture Style

While the layered architecture style has a very clear technical separation of concerns, in a technically partitioned architecture the *domain* is "smeared" all over: no single place in the architecture is responsible for a particular domain.

If the development team is making more domain-aligned changes than technology-aligned ones, they'll find themselves constantly fighting against the "grain" of the layered architecture style. If implementing any change to the system's behavior requires lots of cross-team coordination to ensure that everyone is on the same page, that's a common indicator that many changes are cutting "vertically" through the architecture and are thus misaligned with the nature of a layered architecture.

In cases where large numbers of requests don't actually need to traverse every layer (for example, the analytics domain), it might be worth switching to another architecture style to avoid this antipattern. Consider the modular monolith instead: a monolithic architecture style that's partitioned by domain, not by technical concern.

Trade-Offs

Antipatterns, like patterns, come with their own sets of trade-offs. Here are some key trade-offs for the Architecture Sinkhole antipattern.

Pros

The advantages of adopting this antipattern include:

Evolvability

> Closing each layer isolates changes and creates a clear and predictable dependency graph, improving the evolvability of the architecture. Opening layers reduces clarity, and any change has the potential to have a larger blast radius than expected. Think of it this way: if every layer were open, the architecture would essentially be a Big Ball of Mud. Every change would become a code spelunking exercise through the codebase, likely to introduce unintended side effects.

Cons

The disadvantages of adopting this antipattern include:

Lower cost and time

> Developing distinct layers, with a clean separation of concerns, takes time. Considering and designing each layer's responsibilities and role in processing requests and responses carries an upfront cost. However, if the team is allowed to collapse layers, it can help decrease this cost.

Responsiveness

> Every request *must* pass through every layer, which can increase resource consumption and thus latency, affecting the system's overall responsiveness.

Comparison Chart

The comparison chart in Table 5-2 shows the qualitative ratings for various factors when a team finds itself in this antipattern as opposed to avoiding it. In other words, using the antipattern means keeping every layer closed, while avoiding it suggests opening up one or more layers.

Table 5-2. Comparison chart for the Architecture Sinkhole antipattern

Capability	Using Antipattern	Avoiding Antipattern
IMPROVED		
Debuggability	★★★★★	★★★
Evolvability	★★★★★	★★★
Maintainability	★★★★★	★★★
Modularity	★★★★★	★★★★
Simplicity	★★★★★	★★★
Testability	★★★★★	★★★
DEGRADED		
Initial Time to Market	★★★	★★★★★
Responsiveness	★★★★	★★★★★

Improved

The following sections describe each of the capabilities found in Table 5-2:

Debuggability

When a request can bypass layers, this introduces multiple pathways for a request to traverse through the architecture. That adds an additional layer of complexity when debugging issues in production. Closed layers

offer a clear, predictable request path, making production issues easier to trace and diagnose.

Evolvability

This is one of the strengths of the layered architecture: being able to swap out a layer, for example Angular for Vue, with confidence, as long as the contract between the layer and its dependents is unchanged. With open layers, however, this gets much harder, since the contract now extends to more than just the layer above it.

Maintainability

Opening up layers increases the blast radius of changes to any layer, since the dependency graph might contain more than one layer that might be affected. One way to think about this is to ask: What would happen if every layer were open? It'd be a Big Ball of Mud!

Modularity

The layered architecture is *modular*, except the modularity is by technical concern. Opening up layers violates this modularity.

Simplicity

Opening up layers bends the philosophy of the layered architecture, with the trade-offs being feasibility and responsiveness. However, for someone familiar with the layered architecture, opening up layers can be both surprising and counterintuitive. The more an architect attempts to avoid the antipattern, the fuzzier the architecture becomes. Gone are the clean separation of concerns and the clear dependency graph between layers. Without appropriate documentation to explain the choices made and governance to enforce those choices, every change to the codebase becomes an exercise in spelunking, with unintended effects.

Testability

With every layer closed, it's easy to mock out the dependent layer, which significantly helps with testing layered architectures. However, with open layers, a layer may have more than one dependency, which means multiple layers need to be mocked, and it can be hard to know which layers to mock for which tests. It also obscures the intent of the tests, since the developer has to wade through the dependency graph to understand them.

Degraded

The following capabilities are degraded when the team uses this antipattern:

Initial Time to Market

To design layers with appropriate responsibilities and separate concerns, architects need to spend time thinking through the design. If time to market is a pressing concern, simply choosing to open up some layers can reduce the upfront design time. This can help even at implementation time, since bypassing layers might reduce the need to create additional data structures to transfer data across layers that add no value.

Responsiveness

When in this antipattern, requests may unnecessarily traverse through layers with no-op functions, which increases latency and resource consumption.

Influences

No other outside factors influence this antipattern or change the star ratings.

Am I in This Antipattern?

Governance around open and closed layers, particularly in monolithic codebases that use statically typed languages, can be practically trivial with

tools like ArchUnit. For example, if the architecture mandates that all layers remain closed, then the pseudocode shown in Example 5-1 can be used to enforce this.

Example 5-1. Defining a governance rule around closed layers

```
DESCRIPTION Define Layer Dependencies
DEFINE System layered
  DEFINE LAYER presentation
  DEFINE LAYER business
  DEFINE LAYER persistence
  DEFINE LAYER database
ASSERT(presentation has NO DEPENDENCY on persistence or database)
ASSERT(business has NO DEPENDENCY on database)
```

Related Patterns, Antipatterns, and Pitfalls

The following antipattern is related to the Architecture Sinkhole antipattern:

Architecture by Implication antipattern (see Chapter 4)

> The layered architecture's philosophy is simple: technical partitioning by layer allows for a separation of concerns. This brings with it a host of benefits and some trade-offs — in particular, the Architecture Sinkhole antipattern. An overzealous architect, in an attempt to nullify the trade-offs, might choose to start opening up layers, or worse, ignore the philosophy altogether — quickly falling for the Architecture by Implication antipattern.

Chapter 6. Async Chain of Persistence Pattern

When using asynchronous messaging or event handling, it's possible for messages to get lost during processing, or be *dropped*. The Async Chain of Persistence pattern guarantees that messages never get dropped from the system by ensuring that the message is always persisted in every step of the workflow.

Table 6-1 summarizes the metadata for the Async Chain of Persistence pattern.

Table 6-1. Metadata for the Async Chain of Persistence pattern

CATEGORY	METADATA
CONTEXT:	Event-Driven Architecture, Message-Based Architectures
IMPROVED:	Data Consistency, Data Integrity, Fault Tolerance
DEGRADED:	Elasticity, Responsiveness, Scalability, Throughput

How Do I Prevent Message Loss?

Both event-driven and message-based systems can experience message loss. The following sections describe the differences between event-driven and message-based systems, and where messages can get lost in each.

Event-Driven Versus Message-Based Systems

An *event-driven system* is one where services publish an *event* (something that has happened in the past, such as `order_placed`) to multiple services, which respond to the event, process it, and then in turn publish their own derived events to other services. This chain of events continues until no service is interested in the last published event. The related Extensible Events pattern, covered in Chapter 18, describes ending an event chain; Figure 37-17 in Chapter 37 shows the differences between events and messages in more detail. The typical processing flow for an event-driven system is illustrated in Figure 6-1.

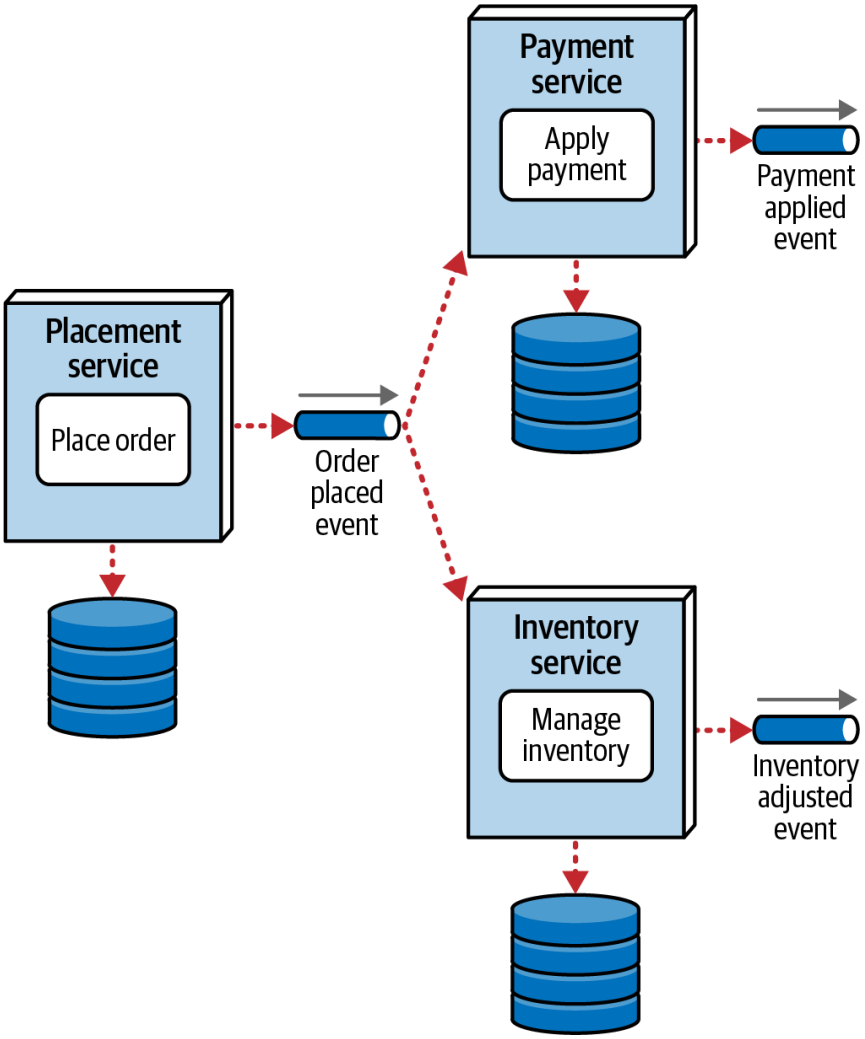


Figure 6-1. The typical processing flow for event-driven systems

A *message-based system* sends a *message* (a command or query that hasn't happened yet, such as `apply_payment`) to a queue to be processed,

typically by one specific service. Unlike an event-driven system, a message-based system typically does not send out a further message advertising what it did. Instead, in most cases, the receiving service processes the message and then the message-processing flow ends with a database commit. The typical processing flow for a message-based system is illustrated in Figure 6-2. Some systems use both event-driven and message-based approaches, depending on their workflows and needs.

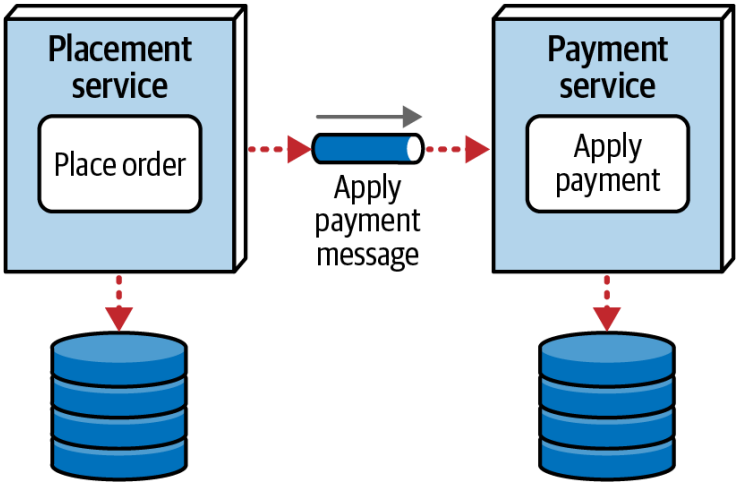


Figure 6-2. The typical processing flow for message-based systems

As demonstrated in the prior figures, event-driven systems using events and topics differ from message-based systems using messages and queues. However, they do share some common traits, particularly with regard to this pattern. For brevity, we'll refer to both messages and events as *messages* and the message channel as a *queue*, unless we're calling out specific cases for event-driven systems (as is the case with "Event chaining", "Synchronous Send", and "Last Participant Support").

Where Messages Can Get Lost

The following sections describe areas in the processing flow where messages can get lost.

Receiving a message

The most common time messages can get lost in a system is when the system is responding to them. In most message brokers and cloud-based

services, the default setting when responding to a message is to acknowledge it automatically, thus *dequeueing* it (removing it from the queue or topic). If a fatal or unrecoverable error occurs or the service instance crashes while processing the message, the original message is lost because it no longer exists on the queue, as shown in Figure 6-3.

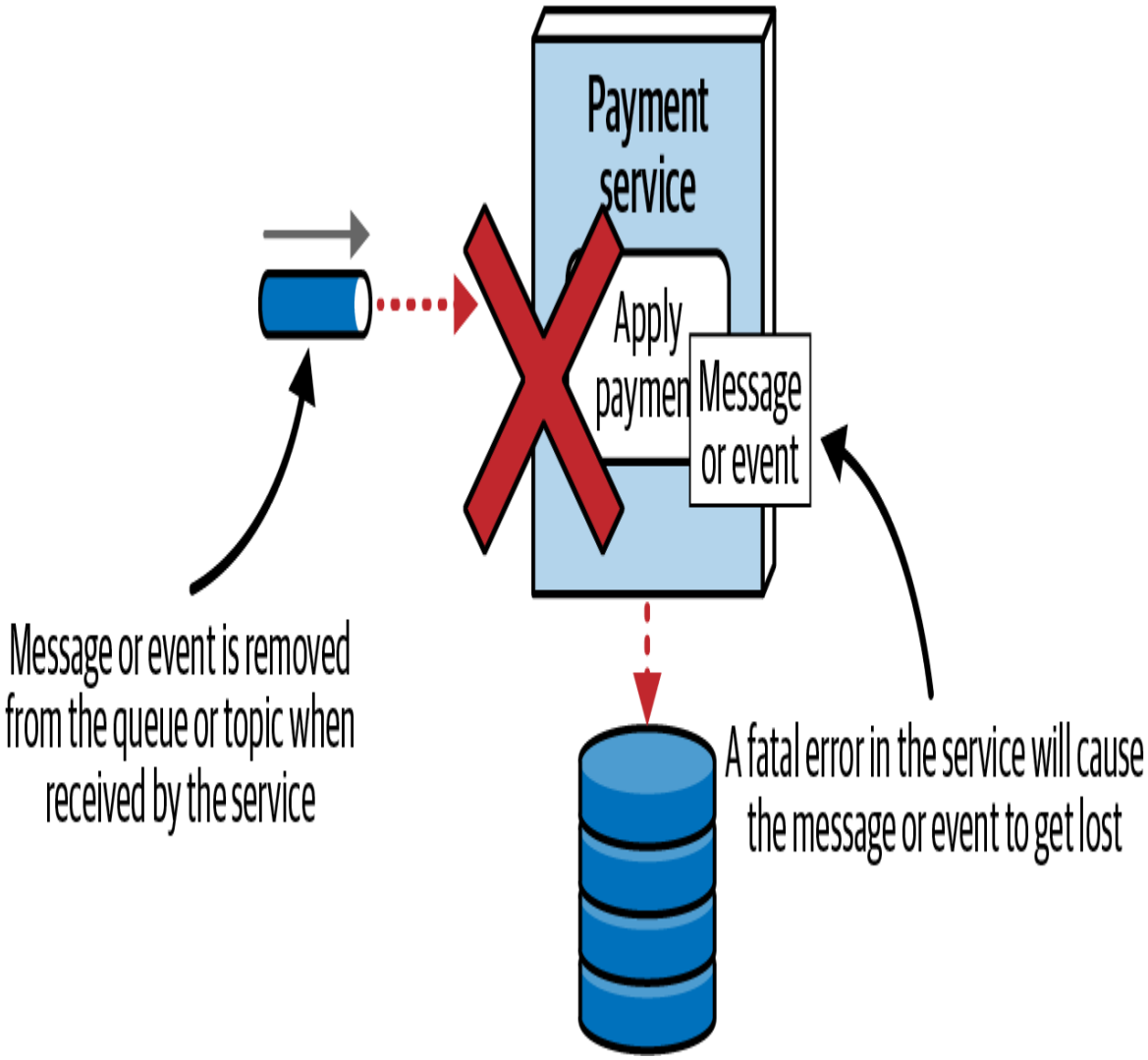


Figure 6-3. A message can get lost if an error occurs while it's being processed

The message broker crashes

Another way messages can get lost is when a message broker or messaging service crashes. The default setting in most message brokers (or cloud-based messaging services) is *not* to persist pending messages, meaning

they're stored only in memory on the message broker. This nonpersistent default is used to improve responsiveness, throughput, and scalability.

However, if the message-broker instance crashes, nonpersistent messages will be lost forever. While this is much less common than the prior case of responding to messages, it can occur. Figure 6-4 illustrates this scenario.

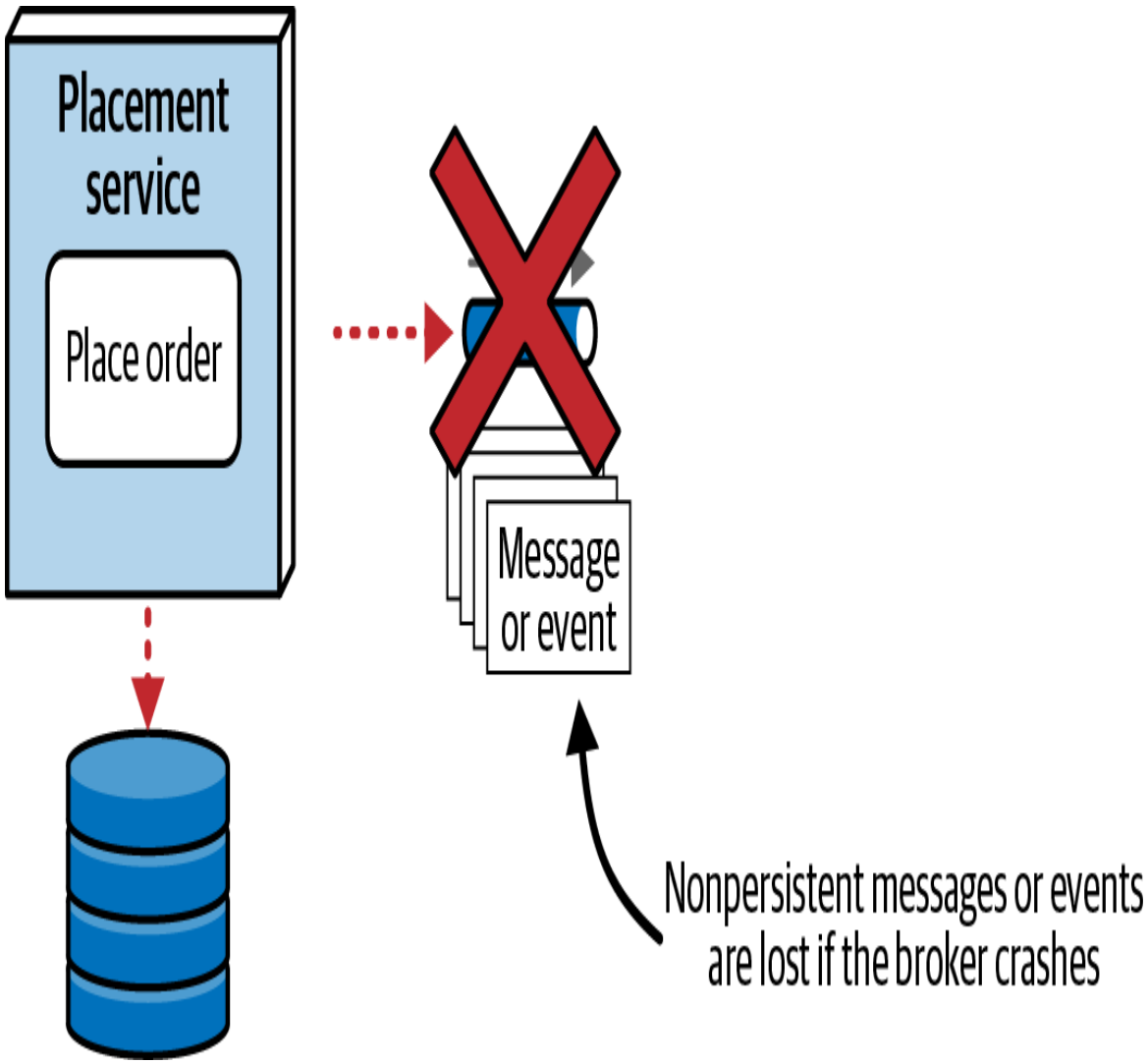


Figure 6-4. A nonpersistent message can get lost if the message broker crashes

Event chaining

With event-driven systems, once a service has completed its processing, it publishes what's called a *derived event*, advertising what it's just done to the rest of the system (see Figure 6-1). This continuous event flow is sometimes referred to as *event chaining*. Events can be dropped in two ways when a

service publishes a derived event. The first has to do with how the event is published, and the second has to do with transaction coordination and timing.

Events are often published through what's called *asynchronous send* mode. When using this feature, the event publisher doesn't have to wait for the message broker to acknowledge that it successfully received the event. Rather, the publisher continues processing after publishing the event, and at some point it receives an acknowledgement indicating whether the message broker successfully received the event or not, in the form of an asynchronous callback. This mode, which can also be used in message-based systems, is an attractive option for developers because it uses a nonblocking call, increasing overall responsiveness in the service publishing the event.

With asynchronous send, however, if a fatal or unrecoverable error occurs *before* the publisher receives the broker's acknowledgement, the system can't guarantee that the event ever made it to the message broker. Even worse, there's no way of knowing whether it's safe to retry the operation, since that might result in a duplicate event, as illustrated in Figure 6-5.

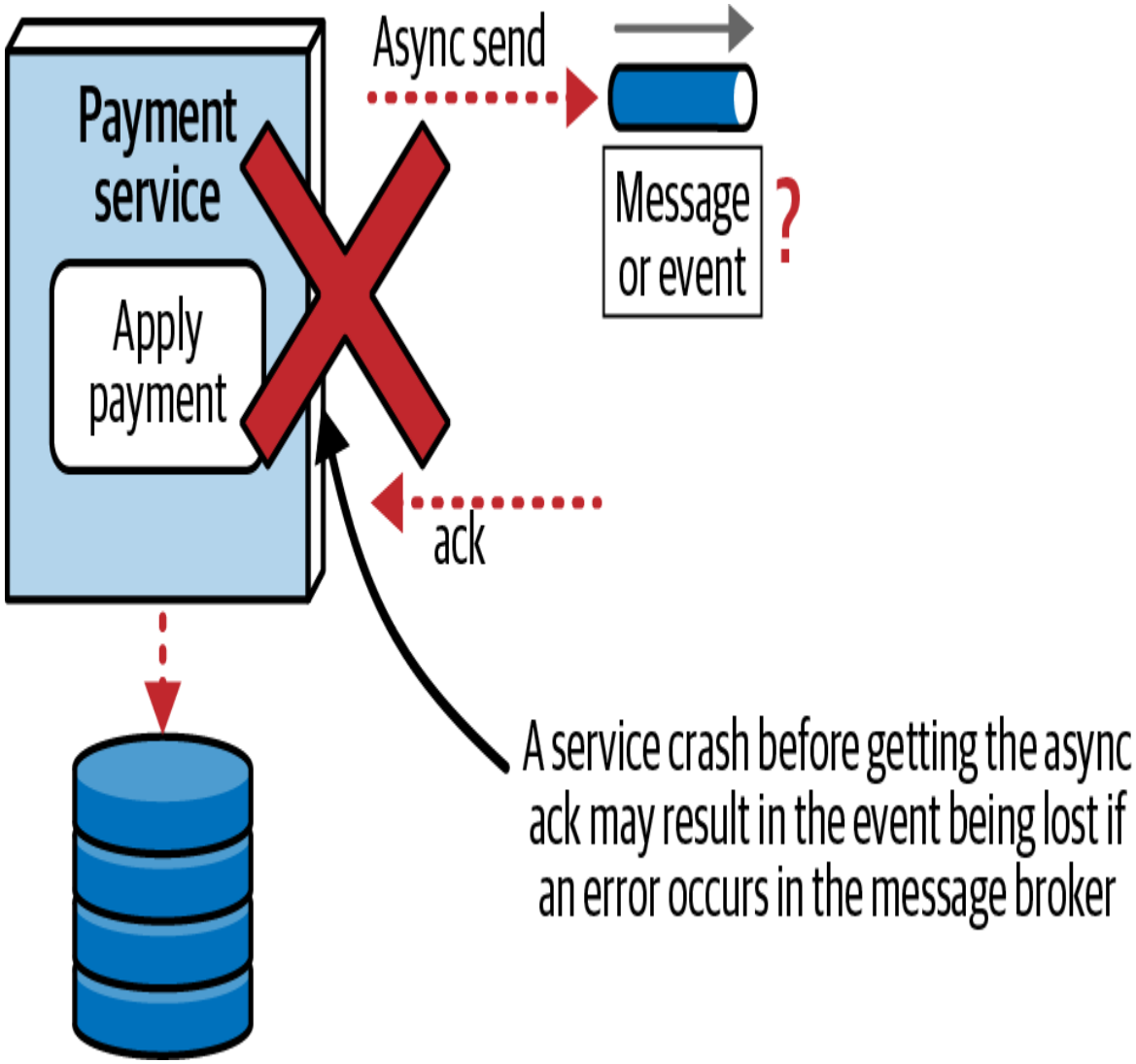


Figure 6-5. A derived event can get lost if an error occurs while sending it asynchronously

Another time messages can get lost in event-driven systems is when a service completes its processing and tries to publish a derived event in the event chain, but crashes before it sends the derived event. In this scenario, the prior event has already been processed and the data committed to the database, but the service can't advertise what it did, thus breaking the event chain (see Figure 6-6).

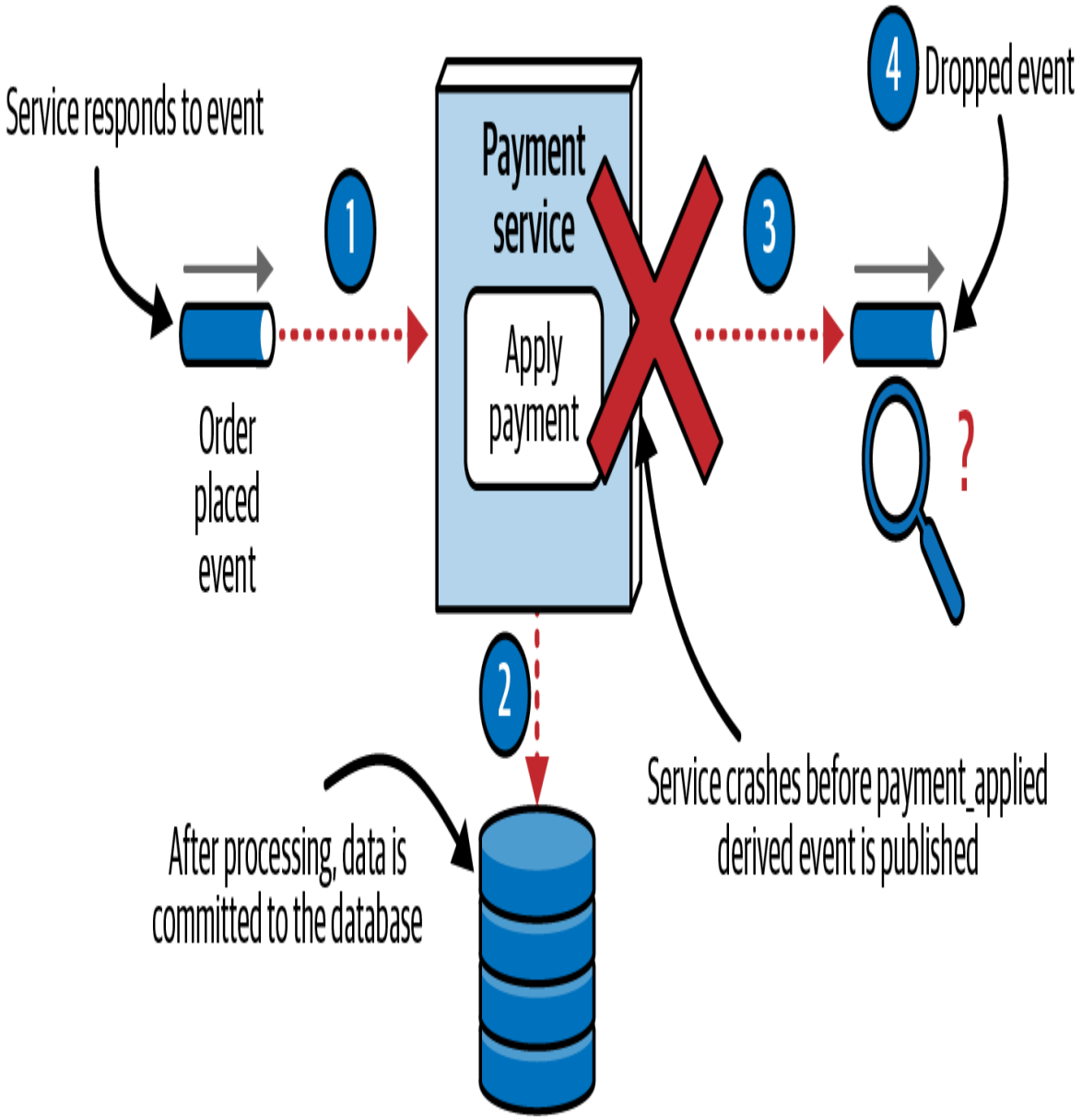


Figure 6-6. A derived event can get lost if an error occurs after a database commit

To summarize, messages can get dropped from the system when:

- A service crashes or encounters a fatal error while processing a message
- The message broker crashes with pending messages on the queue
- A service crashes or encounters a fatal error when trying to publish a derived event after completing processing

Applying the Pattern

The Async Chain of Persistence pattern works by guaranteeing that the message is stored on disk (in other words, *persisted*) during message processing and is not removed from the queue until the event is guaranteed to be persisted further down the chain. If an error occurs, the message can't get lost in the processing flow because it will still be persisted in its prior state. In other words, the message is *always* persisted somewhere in the chain.

Applying this pattern involves four main steps:

- Making messages persistent in the message broker
- Using client acknowledgement mode when responding to a message
- Using synchronous send when publishing a derived event or callback message
- Using last-participant support to acknowledge the message being processed

All four of these steps are necessary to ensure that the system never drops any messages. The following sections describe them and show how to apply the pattern.

Since this pattern applies to all message brokers, platforms, and programming languages, we'll use pseudocode to describe each of the steps in applying this pattern. Architects can use this pseudocode to prompt generative AI tools to generate corresponding code using their desired language and message-broker product or platform.

Message Persistence

To guarantee that a message isn't lost, the first step is to mark it as `PERSISTENT`, meaning that the message broker or cloud-based messaging service will store the message somewhere on disk when receiving it. This is

typically done by setting the delivery mode to `PERSISTENT`, either on the `Message Producer` or when publishing the event.

When the delivery mode is set to `PERSISTENT` on the `Message Producer` object, all messages sent with that message producer will automatically be marked as being persisted, so there's no need to specify the delivery mode when sending messages:

```
var delivery_mode = PERSISTENT
var producer = create_producer(delivery_mode)

// all messages sent will be persisted in the message broker
producer.send_message(APPLY_PAYMENT)
```

To specify the delivery mode when sending the message:

```
var delivery_mode = PERSISTENT
var producer = create_producer()
producer.send_message(APPLY_PAYMENT, delivery_mode)
```

Figure 6-7 illustrates the use of message persistence in the event the broker crashes. This guarantees that all messages will still be there when the message broker comes back up.

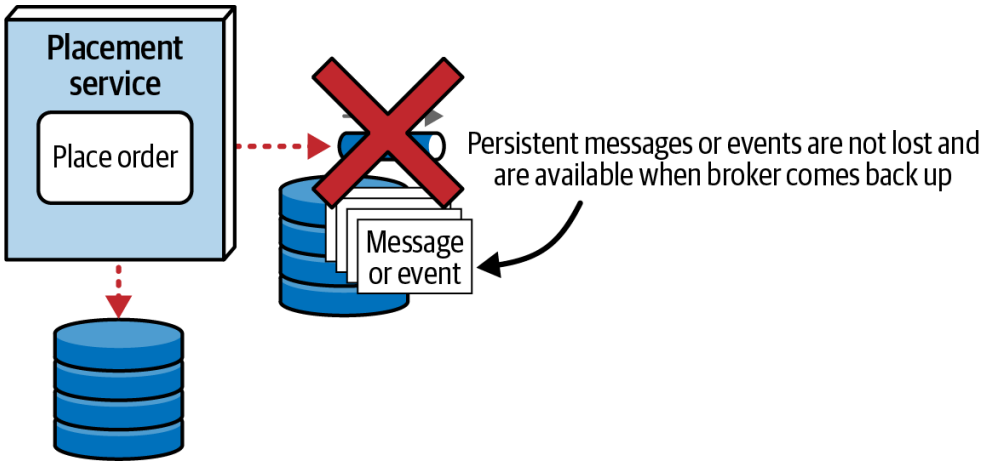


Figure 6-7. Message persistence guarantees there will be no message loss if the message broker crashes

Client Acknowledgement Mode

The second step in applying this pattern is to use *client-acknowledgement* mode when receiving a message rather than the default mode, auto acknowledge. In client-acknowledgement mode, the message being received will remain on the queue until it's acknowledged by the service processing it. The message broker assigns a client ID to the message to indicate that the message is currently being processed, so that it isn't received by any other service within the same consumer group. Once the receiving service acknowledges the message, it's removed from the queue.

Using client-acknowledgement mode guarantees that the message will not be lost when a service is processing the message because it's still persisted in the queue during processing. If the service receiving the message happens to encounter a fatal error during processing, it terminates without acknowledging the message, resulting in the message being redelivered to the same service or consumer group. This redelivery process will continue for a certain number of times (as configured in the message broker). After that, the message broker will send the message to a *dead letter queue*, a queue that stores messages that can't be processed and which is configured in the message broker. The Workflow Event pattern (described in Chapter 35) describes further techniques for handling messages that can't be processed. Figure 6-8 illustrates the use of client-acknowledgement mode.

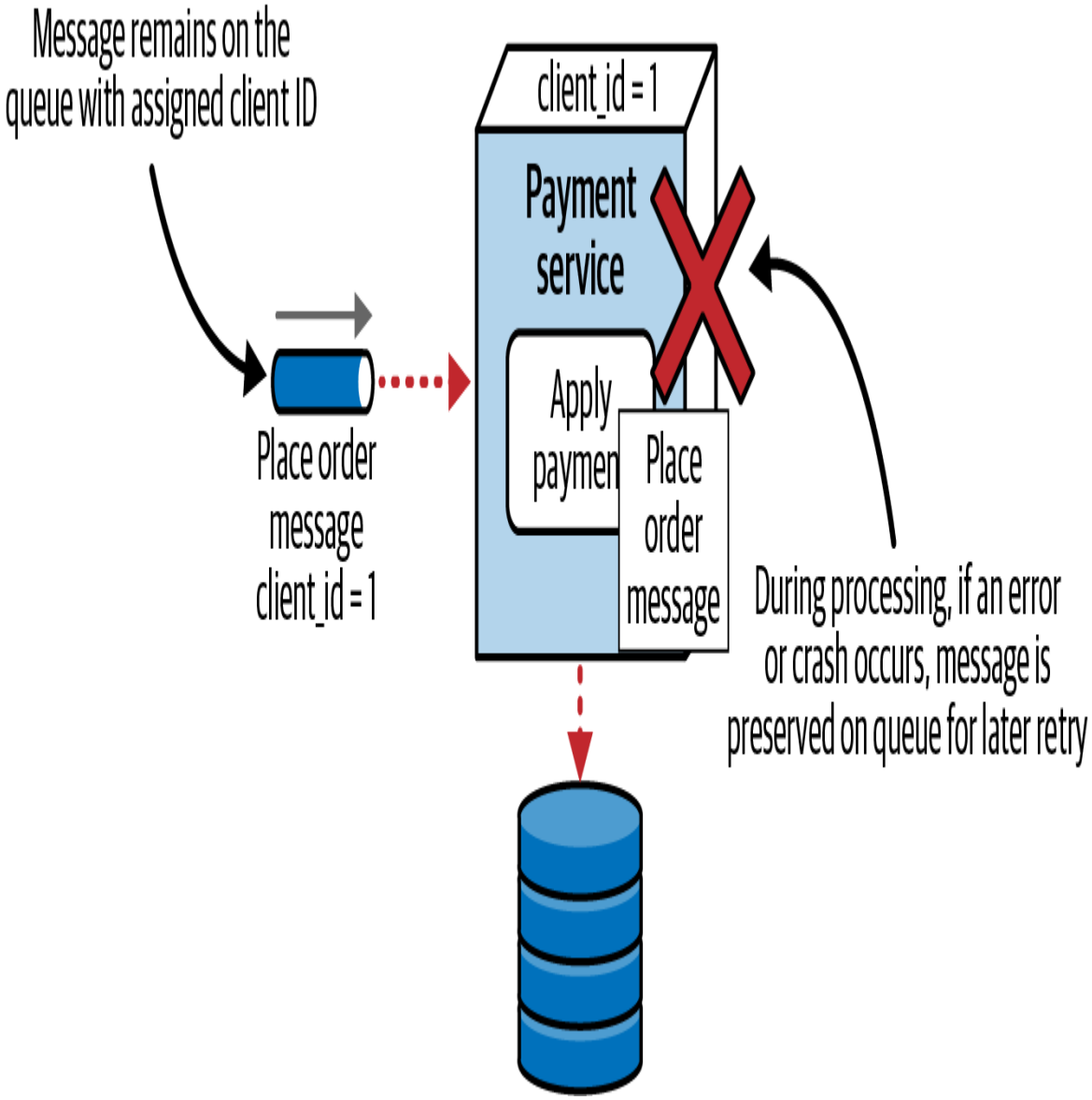


Figure 6-8. Client-acknowledgment mode keeps the message in the queue during processing

Most programming languages and message brokers allow developers to specify the acknowledgement mode when creating a message channel or session, or when receiving the next message from the queue. If the developer chooses to use client-acknowledgement mode when creating the session or channel, all messages published using that session will automatically be marked as using that mode:

```
var ack_mode = CLIENT_ACK
var session = create_session(ack_mode)
var message = session.receive_message()
```

```
process_message(message)
message.acknowledge()
```

Alternatively, the developer can set the client-acknowledgment mode when receiving the message:

```
var ack_mode = CLIENT_ACK
var session = create_session()
var message = session.receive_message(ack_mode)
process_message(message)
message.acknowledge()
```

Either way, using client-acknowledgement mode rather than the default auto-acknowledgement mode keeps the message in the queue until the receiving service has completed its processing.

> **DON'T FORGET TO ACKNOWLEDGE THE MESSAGE**
>
> When using client-acknowledgement mode, it's important to remember to acknowledge the message when processing is complete. Otherwise, depending on the message broker or messaging service, it will be redelivered to the service when the session ends, when the session is rolled back, or if the message broker restarts. This may result in duplicate message processing.

Synchronous Send

The next step in applying this pattern is primarily for event-driven systems, but is also used for message-based systems that return an asynchronous callback or notification message. This step involves always using *synchronous send* when sending a message rather than the common *asynchronous send* option. Using synchronous send, while slower, guarantees that the message broker has received the message and has persisted it on disk during the send operation because it's a blocking call. If an error occurs in the message broker when sending the message, the service knows about the error synchronously through a blocking call. It can

therefore take appropriate action, such as retrying the send or persisting the message to disk for later sending.

The following pseudocode illustrates this technique, using a synchronous blocking call to publish a derived event to the rest of the system:

```
// blocking call to publish the corresponding derived event
var ack = publish_event(PAYMENT_APPLIED)
if ack not_successful
    retry_or_persist(PAYMENT_APPLIED)
end_if
```

Last Participant Support

We've saved the most complex step for last: it's about when to acknowledge that a message has been processed. Message-based systems generally send an acknowledgement to the message broker that processing is complete after the processed data has been committed to the database. In this manner, the message broker becomes the last participant in the transactional flow, with the database being the first (and primary) participant. Acknowledging the message *before* the database commit action opens the door to possible message loss, should the database action fail or the service instance go down after that message is sent but before the database commit is complete. This is because when the message is acknowledged, it's removed from the queue, so it's gone when the error occurs — prior to the database commit action.

The following pseudocode illustrates the order in which we recommend a message-based system should acknowledge a message:

```
var message = receive_message()
process_message(message)
database.commit()
message.acknowledge()
```

Last-participant support is more difficult in event-driven systems (and message-based systems that return an asynchronous callback or notification). This is because *two* participants operate after the database

commit action. One sends the acknowledgement that the event has been processed, and one publishes the derived event after the acknowledgement.

We recommend treating the participant that publishes the derived event as the very last participant. At that point, the original event has been processed and data committed to the database, so event processing has completed and should therefore be acknowledged as the first unit of work completed in the transactional flow. If the derived event is published *before* the original event is acknowledged and a failure or crash occurs, the derived event is already on its way to other services without the service being able to acknowledge the original event, meaning the original event will be processed again.

The following pseudocode illustrates our recommended processing order for last-participant support in event-driven systems:

```
var event = receive_event()
process_event(event)
database.commit()
message.acknowledge()
var ack = publish_event(PAYMENT_APPLIED)
if ack not_successful
    retry_or_persist(PAYMENT_APPLIED)
end_if
```

In the preceding pseudocode, if publishing the `PAYMENT_APPLIED` derived event fails, the service can either retry or persist the derived event on disk for later publishing. In this case, however, the original event is marked as complete (since it is), and the derived event can be persisted in the event of a publishing error for later retry.

Conflicting Patterns

Use care if applying this pattern in conjunction with the Extensible Events pattern (described in Chapter 18). In that pattern, services always advertise what they've done through a derived event, even when other services don't care about the event and no other services are responding to it. It supports good levels of evolvability, at the cost of increasing event noise.

These two patterns conflict because the Async Chain of Persistence pattern calls for all messages to be marked as persistent in the broker. However, if no other services are responding to those extensible events, they'll remain in the queue forever, causing possible issues with storage and memory in the broker.

Therefore, we urge architects and developers not to apply this pattern for extended events. If that becomes necessary, use a time-to-live (TTL) on extended events so they're purged after a period of time. Once extensible events are being processed, the TTL can be removed or the Async Chain of Persistence pattern applied to that event.

Trade-Offs

While the Async Chain of Persistence pattern is powerful, it does come with trade-offs that architects should be aware of and analyze prior to applying this pattern. The following sections describe the pros and cons of this pattern.

Pros

The advantages of applying this pattern include:

Lower risk of data loss

> The single biggest advantage of this pattern is that it prevents loss during message processing. Since this is a concern in most asynchronous systems, most architects leverage the Async Chain of Persistence pattern when the trade-offs indicated below aren't significant enough to outweigh possible data loss.

Cons

The disadvantages of applying this pattern include:

Potential duplicate processing

Enabling client-acknowledgement mode could result in the same message being processed multiple times, resulting in duplicate processing and consequently duplicate records in the database. This can occur if the service instance crashes *after* the database commit action and *before* the message acknowledgement. Although this error scenario is uncommon, it's worth considering adding logic that checks incoming messages to see if they've already been processed. One way to do this is by recording and tracking message IDs once a message's data has been committed in the database, and comparing incoming message IDs to that list. The trade-off here is responsiveness, since the system has to retrieve and check the list of already processed message IDs.

Degraded operational characteristics

The other downside of this pattern is that it does degrade operational capabilities like scalability, elasticity, responsiveness, and overall message throughput. Most of this has to do with persisting messages. Figure 6-9 illustrates the difference in throughput between persistent and nonpersistent messages using ActiveMQ with a 2 KB message size.

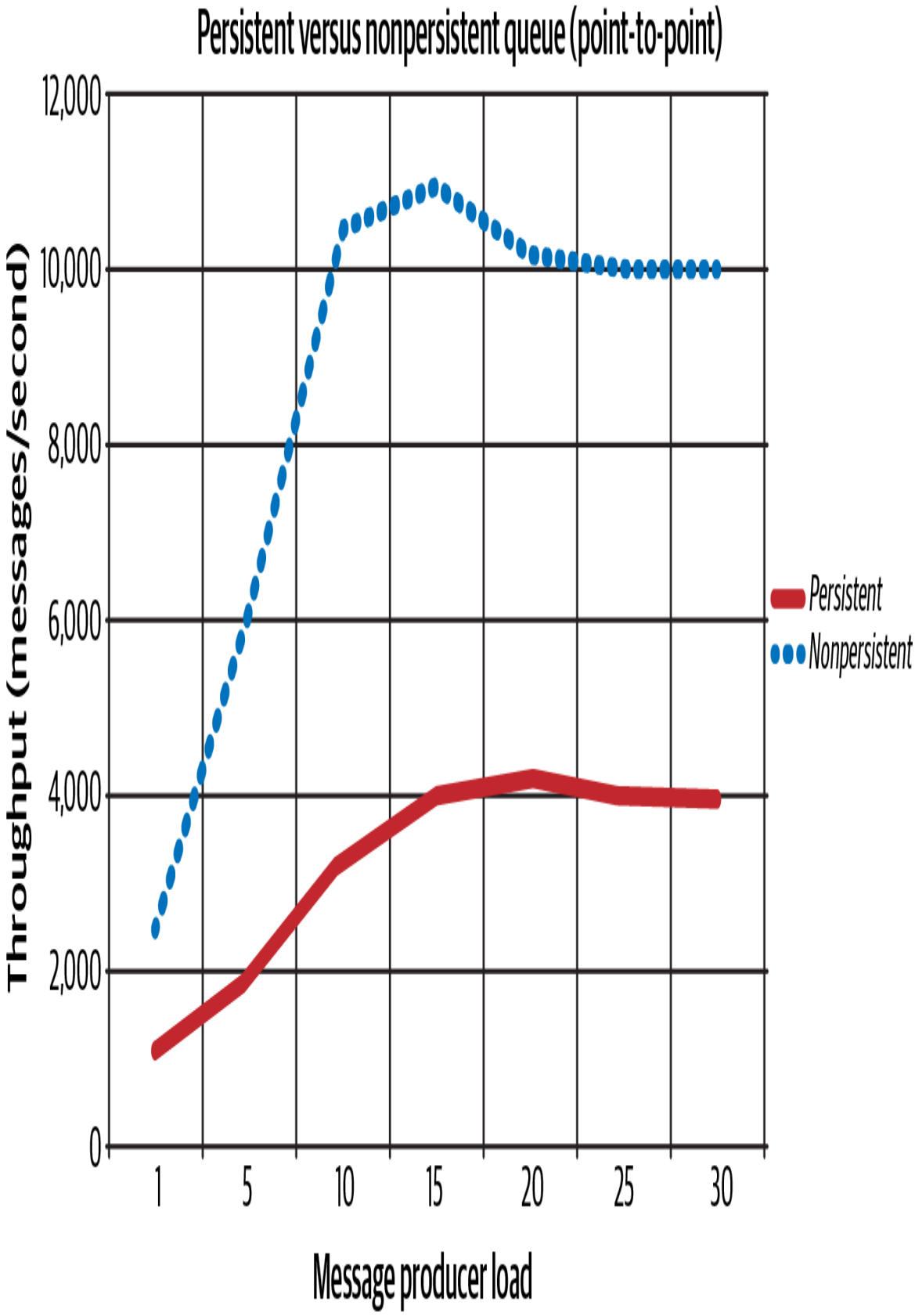


Figure 6-9. Comparing persisted and nonpersisted messages

When sending a message to a queue, the message broker will take longer when persisting the message, negatively impacting responsiveness. This in turn can degrade both scalability and elasticity. We encourage architects to fully quantify their responsiveness and scalability needs before deciding to adopt this pattern.

Comparison Chart

Table 6-2 summarizes the star ratings associated with this pattern. The Without Pattern column refers to using nonpersistent messages along with auto-acknowledgement mode for messages, and optionally using async send when sending a message to a queue.

Table 6-2. Comparison chart for the Async Chain of Persistence pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Data Consistency	★	★★★★★
Data Integrity	★	★★★★★
Fault Tolerance	★★★	★★★★★
DEGRADED		
Elasticity	★★★★★	★★
Responsiveness	★★★★★	★★★
Scalability	★★★★★	★★★
Throughput	★★★★★	★★★

The following sections describe each of the capabilities found in Table 6-2.

Improved

The capabilities that are improved when applying this pattern include:

Data Consistency

> As we've demonstrated, systems can drop messages in a variety of places. Ensuring that messages don't get dropped helps maintain data consistency across services and systems, since a message can always be

picked up by another service for processing. Teams can leverage the Workflow Event pattern (see Chapter 35) for messages that are repeatedly posted with errors, ensuring they'll always eventually be processed.

Data Integrity

When a message is dropped, its necessary processing and updates don't occur, which degrades *data integrity*, or correctness. This pattern guarantees that the system won't lose messages; any and all updates will eventually be applied to the system.

Fault Tolerance

One aspect of fault tolerance is how quickly the system can recover from a fault. Without this pattern, when a fault occurs in a receiving service, the message has usually left the queue and is therefore lost forever. This pattern guarantees that when a fault occurs in a service while processing a message, the message is preserved in the queue and can be picked up quickly by another service for processing.

Degraded

The capabilities that are degraded when applying this pattern include:

Elasticity

Elastic systems generally require high levels of responsiveness to process a burst of sudden user load as fast as possible. The faster a system can process a request, the more requests it can handle. Because persisting messages is on average about three times slower than not persisting them, the service handling the request takes longer to receive the next request. This can make it difficult for that service to keep up with the throughput of requests during a spike in user load. For this reason, we've lowered the rating for elasticity to two stars. We urge architects to fully quantify their elasticity needs before adopting this pattern.

Responsiveness

Because this pattern leverages synchronous sends to a queue and persisted messages, adopting it will generally hurt responsiveness. Since most message brokers and messaging services also store the message in memory for faster reading, we've reduced the responsiveness star rating from five to three. Only once the message is acknowledged is it removed from both memory and disk.

Storing messages in memory in addition to disk is a configurable parameter in most message brokers. However, it depends on a number of conditions, such as memory capacity and message volume, so the system can't guarantee the message will be stored in memory. Don't count on that when considering responsiveness needs for receiving persisted messages.

The final part of this pattern that affects responsiveness is its use of synchronous send. Because synchronous send uses a blocking call to receive the acknowledgement, no other processing can take place between sending a message and receiving acknowledgement from the message broker. This delay becomes more significant when using persisted messages, as in this pattern.

Scalability

Because responsiveness and throughput are degraded, correspondingly scalability is generally degraded as well. Whereas elasticity is about sudden spikes in user load, scalability is about handling gradual increases in the number of concurrent users. Since the request spends more time in each service (particularly the ones forwarding the message through a queue), each service can handle fewer concurrent users. Although message brokers and messaging services differ in overall throughput between persisted and nonpersisted messages, persisted messages will always have less overall throughput, which could degrade scalability for high use loads and message-request volumes.

Throughput

Depending on the message broker or messaging service, persistent messages can take up to four times longer to send than nonpersistent messages. This affects overall message throughput, as Figure 6-9 illustrates.

Influences

This pattern can influence systems integration in that it can increase coupling between systems and services. Because this pattern is spread across multiple deployment units, all participants in the event flow, including services or other systems, must align with this pattern's constraints.

For example, assume that System A is sending a message to System B. To apply this pattern across systems, System A must use synchronous send and mark the message as persistent in the broker. System B must also use client-acknowledgement mode when responding to messages from System A. This requires System B to have intimate knowledge of how System A implements and configures its queue. Each system must have governance in place to keep them in sync when using this pattern.

This pattern also influences team topologies. Because it can span multiple systems, implementing it successfully requires team coordination. The key factor here is the *distance* between teams; the further apart they are, the harder this pattern is to leverage. We'll call this *distance coupling*, meaning the distance between the development teams involved.

To illustrate this point, consider using the Async Chain of Persistence with two services, `Order Placement` and `Payment`, each maintained by the same development team (see Figure 6-10).

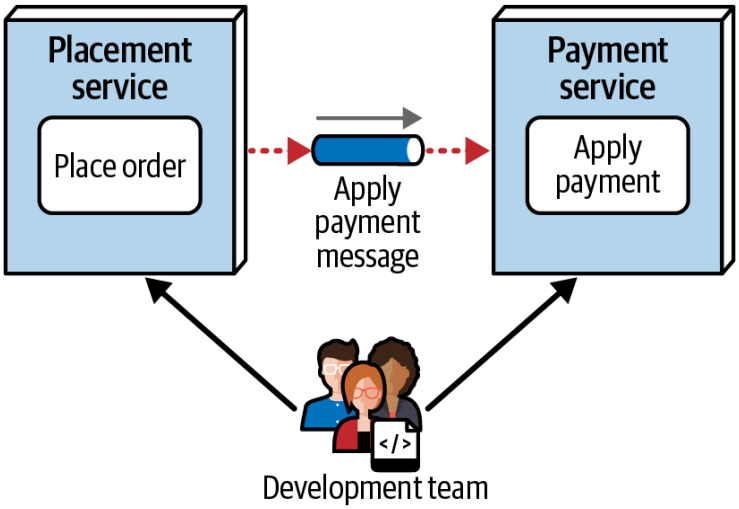


Figure 6-10. A single team maintaining both services has very low distance coupling

Here the distance coupling is very low, so this pattern is easy to implement; team members can coordinate directly to implement and test all of the required elements.

Now consider the same set of services, but with each service maintained by a separate team (Figure 6-11). Increasing the distance between teams makes it more difficult to apply this pattern.

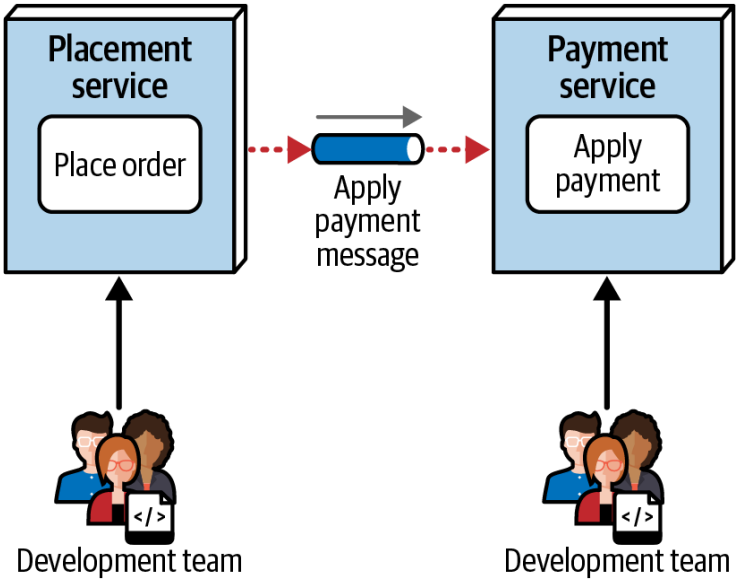


Figure 6-11. Coordinating multiple teams makes implementing this pattern more difficult

To illustrate an extreme (but certainly possible) example, suppose now that two separate *systems* are involved with implementing this pattern. One team handles the `Order Placement` system, and another team (possibly in a

different geographical region) is responsible for the `Warehousing` System (see Figure 6-12).

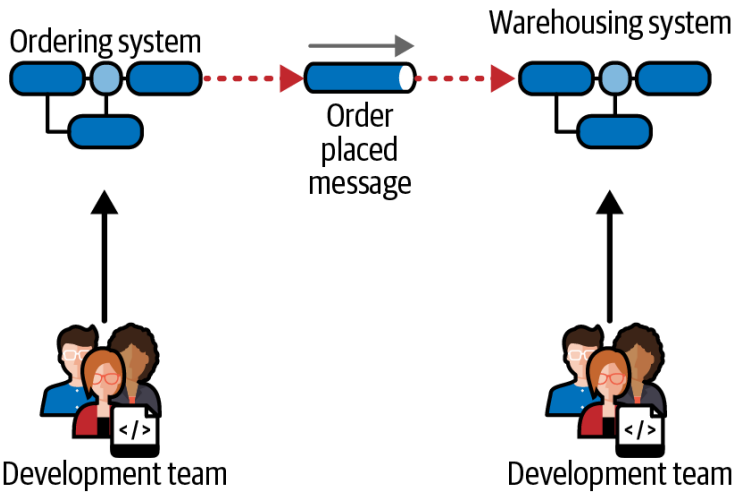


Figure 6-12. Coordinating multiple teams across different systems increases the coupling distance

This scenario, particularly when teams are geographically separated, significantly increases the coupling distance, making the pattern hard to implement. These systems might be written in different languages on platforms; they might even be third-party systems.

Before implementing this pattern, architects should account for these factors in determining its feasibility, particularly for integrating systems and coordinating multiple teams.

How to Govern This Pattern

This pattern can be governed by using fitness functions. Through logging and tracing, a fitness function can compare the messages sent in the system with messages received by other services and with expected results to detect missing messages. Automated unit tests in the form of chaos engineering can also detect if the system is losing messages. Some ways to perform this type of testing are:

- Bringing down a message broker with pending messages
- Forcing an exception or crash when processing a message

- Validating the use of synchronous send
- Forcing an exception or crash when publishing a derived event after a database commit

Fitness functions can also count the number and frequency of errors when sending and processing messages, duplicate messages, and unprocessed messages sent to a dead-letter queue.

From an implementation standpoint, architects should write automated fitness functions to mark all events published by a service as `PERSISTENT`, and ensure that all services responding to the event are using client-acknowledgement mode. Fitness functions can also verify that all messages are sent using synchronous send mode.

To govern and detect violations of these constraints, we recommend using static analysis tools, such as ArchUnit in Java, NetArchTest and ArchUnitNET in C#, PyTestArch in Python, and TSArch in TypeScript and JavaScript.

Generative AI can be used to generate these tests as well. Here's an example prompt:

```
Write a fitness function test in Java using ArchUnit
that ensures all messages sent in the system are sent
using PERSISTENT mode.
```

This prompt will generate executable code in ArchUnit to govern this pattern's message persistence. Similar prompts can generate tests for the exclusive use of client acknowledgement and synchronous send modes.

Related Patterns, Antipatterns, and Pitfalls

Because this pattern is applied within the context of event-driven architecture, it's related to several EDA patterns described in this book:

Contextual Events pattern (see Chapter 12)

The Contextual Events pattern increases the number of events triggered in a system. This could further increase event noise, which is already a negative trade-off associated with the Async Chain of Persistence pattern.

Data-Based Event Contract pattern (see Chapter 14)

The size of the *event payload* (the amount of data passed in an event) could accentuate some of the negative trade-offs associated with this pattern. The Data-Based Event Contract pattern increases the data size of events, potentially affecting scalability, throughput, and responsiveness.

Extensible Events pattern (see Chapter 18)

Like the Contextual Events pattern, Extensible Events increases the number of derived events in the system. This means it too can affect scalability, throughput, and responsiveness by increasing event noise. We recommend avoiding this pattern for message-based architectures; consider at least setting a time-to-live on sent messages, so as not to fill up storage space with persisted messages that are never dequeued and processed.

Key-Based Event Contract pattern (see Chapter 23)

Unlike the Data-Based Event Contract pattern, the Key-Based Event Contract pattern only contains the *keys* of the data entities in the message payloads. Thus, it doesn't exacerbate the negative trade-offs of Async Chain of Persistence.

Workflow Event pattern (see Chapter 35)

Because Async Chain of Persistence leverages client acknowledgement mode when responding to events, it's possible that a message or event may be continually rejected and not processed due to exceptions. In such cases, the Workflow Event pattern is useful.

Chapter 7. Backend for Frontend Pattern

Over time, applications often have to cater to a variety of clients with different data and interaction demands. A single backend attempting to serve *all* these clients can become hard to maintain, potentially creating a development bottleneck. In recent years, more organizations have realized the power of platforms and opened up their APIs for third-party integrations, further aggravating the problem.

The Backend for Frontend (BFF) pattern addresses this concern by introducing a dedicated backend for every client, tailored specifically to its needs. Table 7-1 summarizes the metadata for the BFF pattern.

Table 7-1. Metadata for the Backend for Frontend pattern

CATEGORY	METADATA
CONTEXT:	Monolithic Architectures, Multiple User Interfaces
IMPROVED:	Adaptability, Agility, Availability, Deployability, Error Handling, Evolvability, Fault Tolerance, Flexibility, Maintainability, Simplicity, Testability
DEGRADED:	Cost, Responsiveness, Reusability

How Can I Support Multiple User-Interface Types?

With the rise of sophisticated frontend technologies and frameworks, modern browsers can run extremely sophisticated applications entirely on the client side. This advancement bodes well for the layered architecture style, which allows the Presentation layer (that is, the frontend) to be built and deployed independently of the backend, as shown in Figure 7-1. In this topology, the backend provides a single general-purpose API to serve the needs of the client.

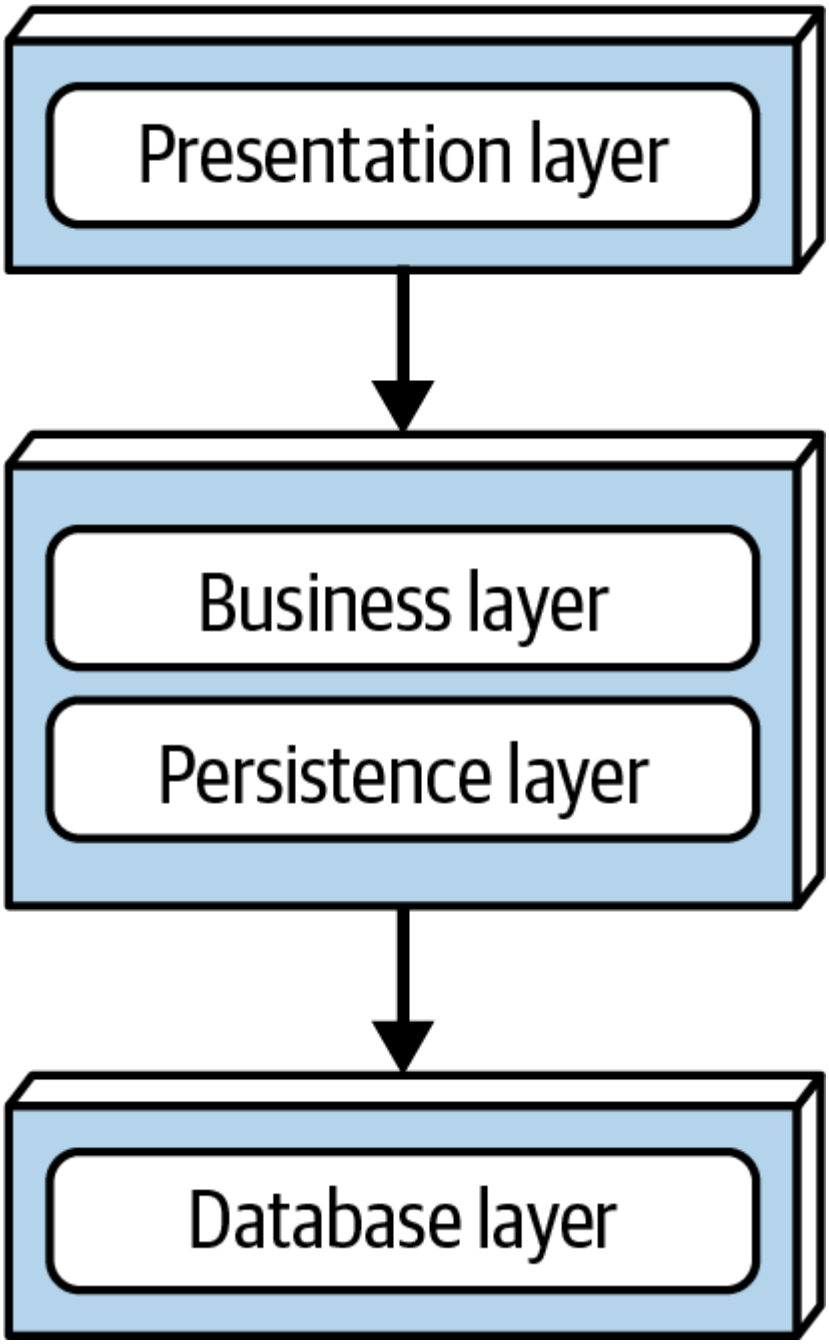


Figure 7-1. Layered architecture with an independently deployed frontend

Eventually, in the modern multidevice ecosystem, developers may need to add new types of interactions — perhaps user interfaces (UIs) for smart TVs, kiosks, and tablets, wearables, or other mobile devices. If these clients' demands are somewhat similar, maintaining a single backend API to serve the needs of multiple clients would be the most straightforward route, as shown in Figure 7-2.

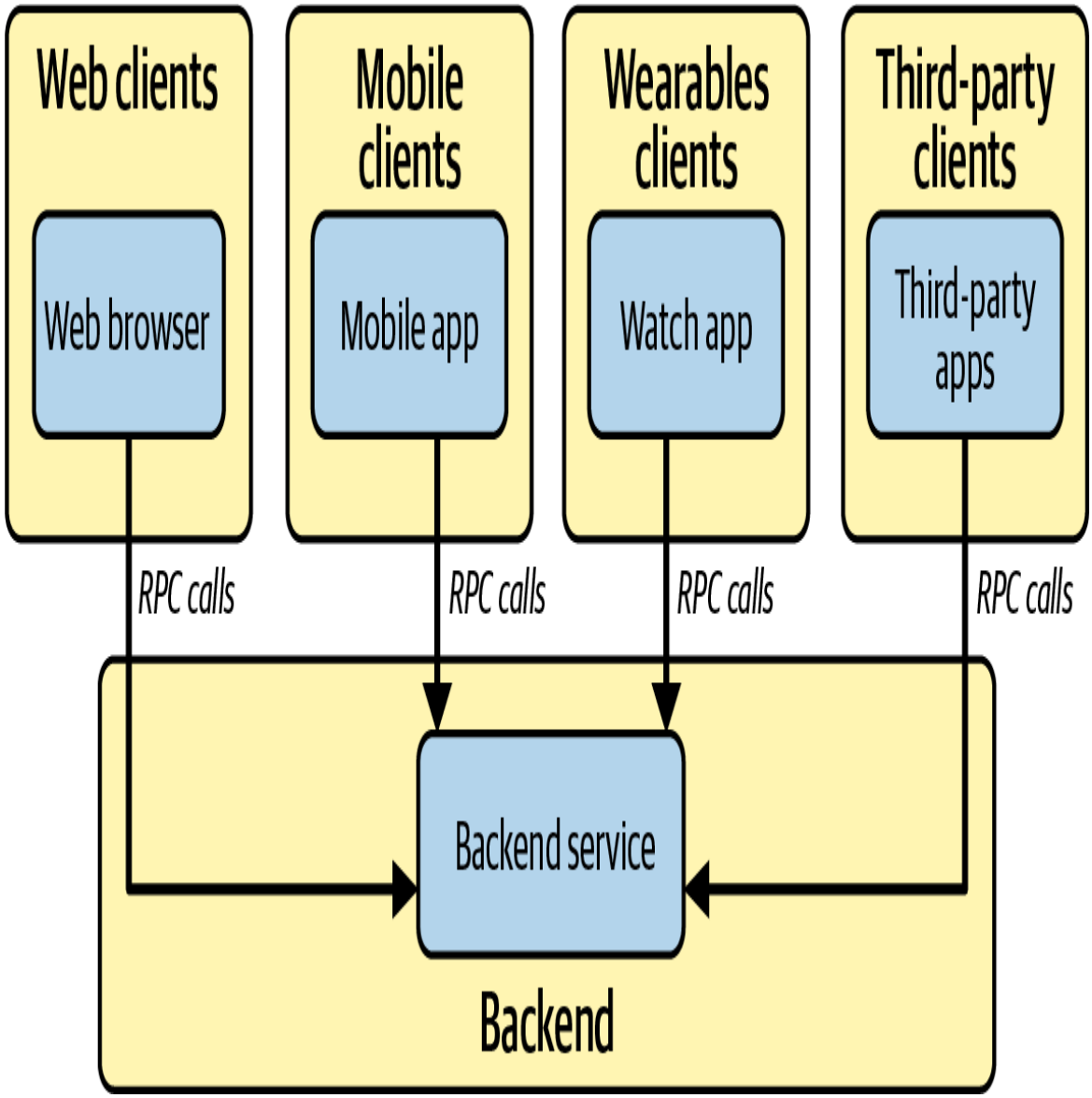


Figure 7-2. Single BFF supporting multiple different clients

However, in most cases, different interfaces have widely different UIs and user experiences (UXs), and consequently place very different demands on the backend. Consider the case of an airline site with a landing page promoting "getaway" package deals on vacations to exotic locations, with a call to action to tempt its visitors into booking flights. The primary purpose of this web page is to make it easy for potential customers to book a flight, whether for a planned trip or an ad hoc vacation. The airline's mobile application, however, should focus on what matters most to harried passengers: opening the app should take them by default to the boarding

pass, and the app should make it easy to access information about boarding times, layovers, and gate assignments. (Take a moment to think about how wearables differ in their interaction patterns for this particular use case.) Additionally, mobile and wearable clients should be able to present real-time updates, even under unreliable network conditions. While customers *can* book flights on the mobile application, that is not its primary purpose.

Different interfaces also provide different capabilities — for example, a bank's mobile application might let customers deposit checks by taking a picture using the camera, or leverage native push notifications to alert users of activity on their accounts. Such a mobile application might support biometric authentication that is native to the mobile platforms — as opposed to the web application, where passkeys work differently and biometrics require WebAuth APIs.

Fundamentally, different devices offer (or should offer) a different experience than a web application does. However, they also have different constraints than, say, a desktop or a laptop in terms of compute, power, and connectivity. For example, having a mobile application open up multiple requests with multiple connections to the backend could drain a smartphone's battery life and degrade performance over slower networks. Mobile devices also need to get their data as quickly as possible, in case they lose connectivity.

To add to all that, many organizations treat their APIs as products, building out "platforms" to further extend their audiences by allowing third parties to integrate with their services. This enables automations and perhaps even a marketplace of plug-ins and extensions. Externally facing APIs usually have a different lifecycle and are typically less volatile than APIs meant to serve internal teams. This typically forces their developers to use versioning, which increases maintainability overhead and requires more governance.

All of these factors, in combination, mean that a single backend just won't do. Having a single backend forces the backend team to take on multiple responsibilities and attempt to cater to drastically different expectations,

often causing their backlog to balloon. Testing is onerous, since the impact of a change can be widespread. What's more, having a single set of service reliability metrics (SLIs, SLOs, and SLAs) often reduces every client to the lowest common denominator. In conclusion, in certain scenarios, having a single backend hinders progress for all involved.

Applying the Pattern

The Backend for Frontend (BFF) pattern works by introducing a dedicated backend service for each frontend client (such as web, mobile, wearables, or third-party integrations). Each BFF is an intermediary sitting between the client it serves and the underlying APIs, as shown in Figure 7-3.

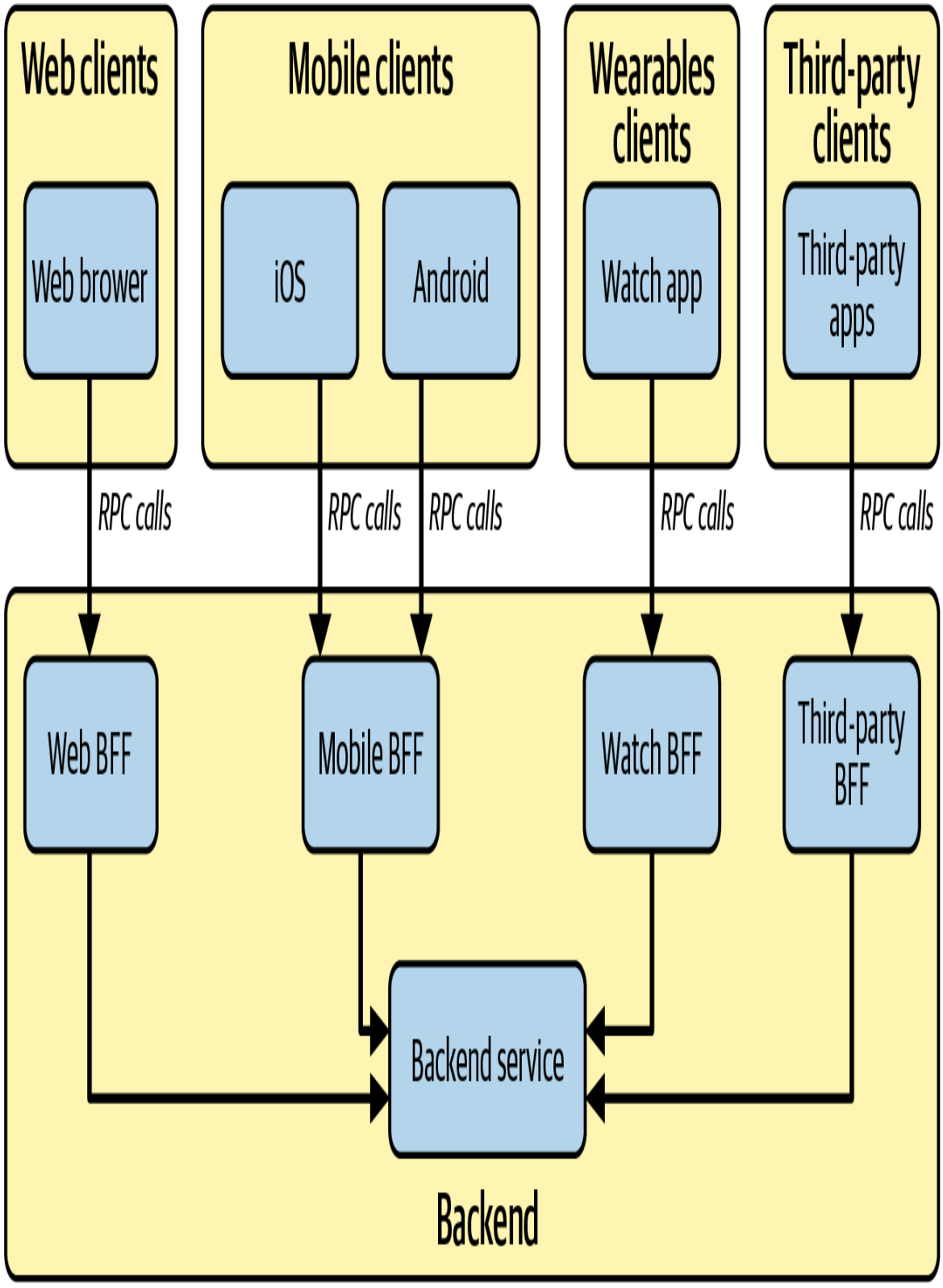


Figure 7-3. The BFF pattern

This allows each BFF to specialize to meet the *specific* needs of the frontend. For example, a BFF can:

- Shape data to optimize payloads for a specific kind of client and reduce "chattiness"
- Orchestrate calls to other downstream services
- Implement client-specific behavior, such as push notifications for mobile clients, live updates on wearables, or using the push API for web browsers
- Implement appropriate authentication and authorization mechanisms
- Allow for "testing in production" logic, such as feature toggles and A/B testing per client
- Enforce client-specific rate limiting and caching strategies
- Guarantee a specific quality of service with regard to SLIs, SLOs, and SLAs
- Improve observability and incident response for each client

However, a BFF is not the place to implement business logic. That belongs in the backing services. The BFF is simply an adapter.

BFF Granularity

People often ask how granular BFFs should be. Should they use one BFF per *UI*? That is, one BFF for the web client, one for mobile, one for tablets, one for third-party integrations, and maybe one for wearables or smart TVs? Or should it be one BFF per *client type*? That is, a *separate BFF* for every different mobile, wearable, and appliance operating system the application supports? Much like with the nebulous "single-purpose" principle in microservices, there's much to consider here.

Let's consider the trade-offs involved with each approach:

Per user interface

In this setup, one BFF supports the web application and another supports all the different mobile (and perhaps even tablet) operating systems. In essence, it has one BFF per client *type*, as shown in Figure 7-2.

This setup can work well for a group of applications developed by a single team (for example, all mobile applications), because it reduces the maintenance overhead of making feature updates.

However, for complex clients, it rarely plays out well.

Consider the case of "mobile-first" applications. Different platforms each have very different UI, UX, interaction patterns, and specialized APIs for notifications and alerts. Thus, teams are usually organized around specializing in a particular platform, with separate teams working on clients targeting different platforms. These teams evolve to have different requirements, priorities, and release cadences. In these cases, a shared BFF essentially becomes the shared kernel. This brings with it all the challenges of a shared kernel, including:

- Prioritizing and coordinating competing intakes can make the BFF the development bottleneck.

- The API cannot be optimized for the unique requirements of any particular client, so it represents the lowest common denominator — in other words, it's a jack-of-all-trades and master of none.

- Sharing the BFF increases the scope of impact for any change, consequently increasing the scope of testing required for every change.

Per client type

The alternative is to build out a BFF *per client type*, as shown in Figure 7-4.

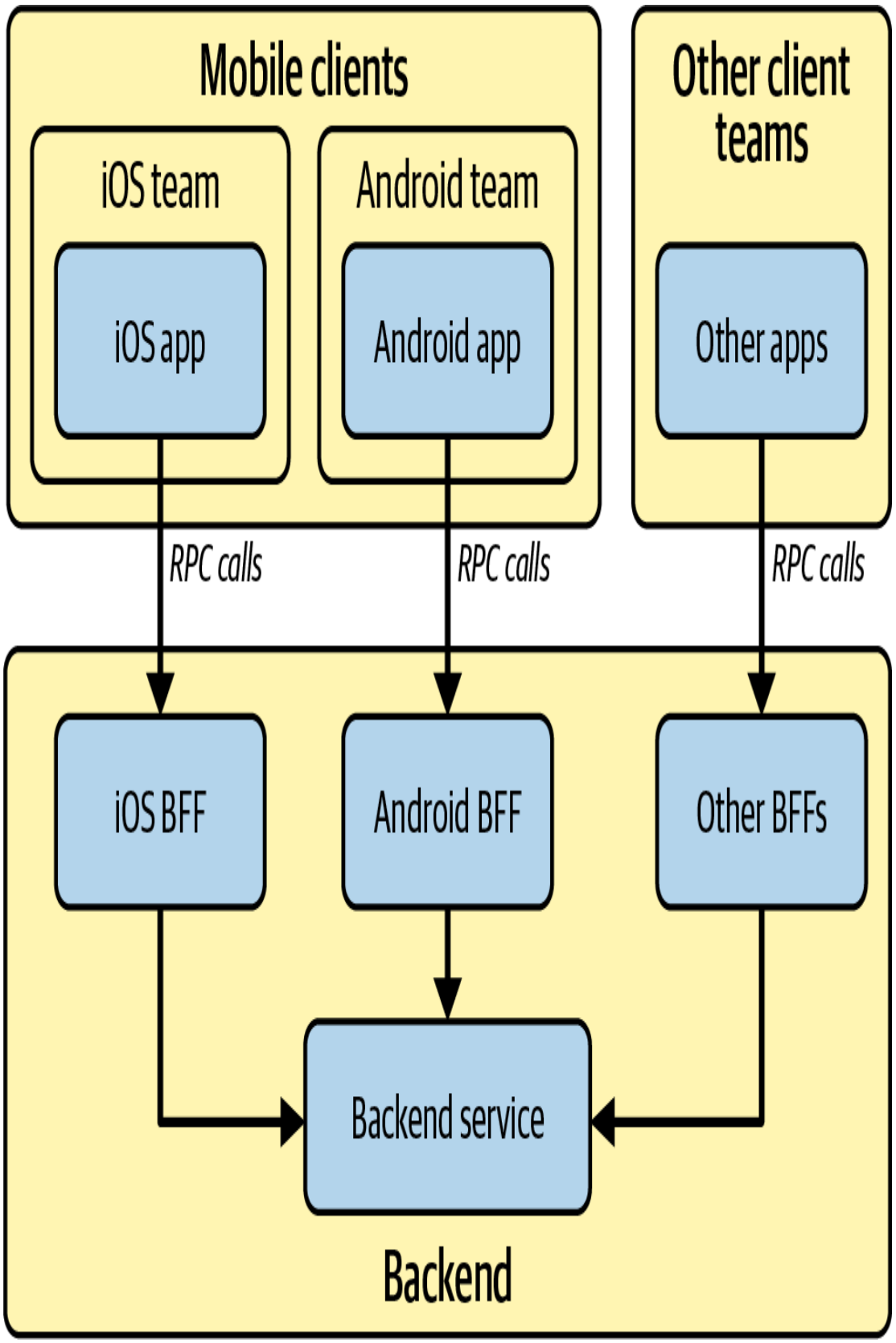


Figure 7-4. BFF per client type

Figure 7-4 only shows mobile clients, but the same pattern can, and most likely would, be applied to other client types as well. This model works well for highly specialized teams building out clients on different platforms. By changing the ownership model, this setup *can* offset some of the concerns brought on by having one BFF per UI. When one BFF supports each client type, the team building out the client application takes ownership of building that BFF. This is shown in Figure 7-5.

This shift in the ownership model means the BFF is no longer the development bottleneck. Furthermore, the owning team can hyperspecialize the BFF, tailoring it to meet the client's every need. The BFFs are far more granular. They also exhibit very specific modularity, testability, and deployability improvements, since the scope of clients that could be impacted by a change is very specific.

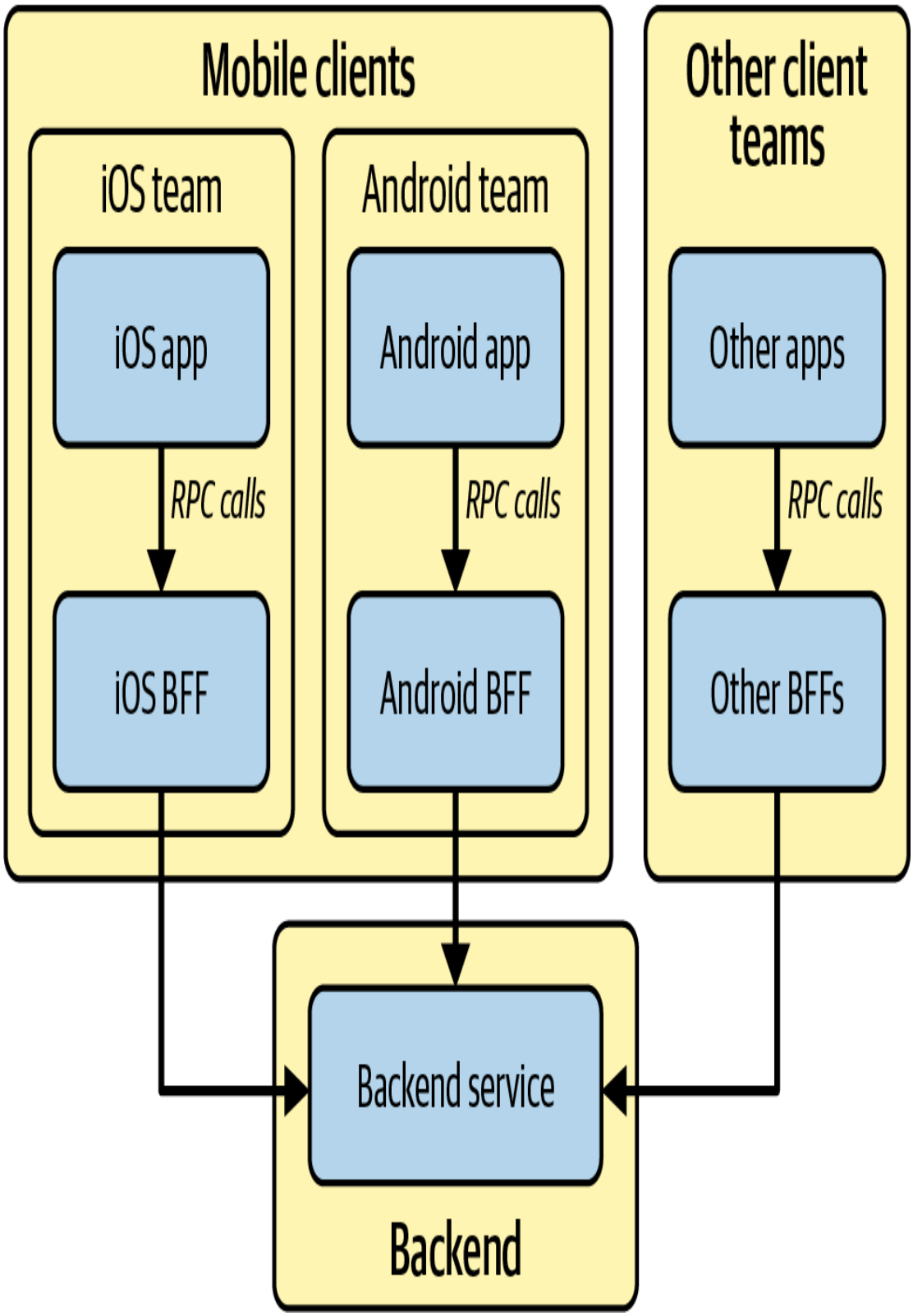


Figure 7-5. Shifting the ownership model

Having multiple BFFs, each supporting one client type, creates the potential for a lot of duplication, particularly with regard to cross-cutting concerns. Should these be shared libraries or shared services? We address the trade-offs between the two under "Reusability" in "Degraded".

The Place for Specialized Data Query Languages

A question that often arises when considering BFFs is the place for data query languages like GraphQL. These languages provide an alternative to RESTful APIs, allowing clients to request exactly the data they need and nothing more. Teams might be tempted: they could implement a single backend, yet be able to expose a unified, efficient mechanism for multiple clients to aggregate all they need in a single request.

However, the BFF pattern itself is protocol agnostic: developers can implement it using REST, GraphQL, or other RPC mechanisms like gRPC. Technologies like GraphQL can help with data shaping and query optimization, but they don't address the organizational, operational, and ownership challenges that the BFF pattern aims to solve.

Trade-Offs

The BFF pattern introduces several trade-offs that teams must consider. Furthermore, the granularity question (whether to have one BFF per user interface or one per client type) introduces its own set of trade-offs.

Pros

The advantages of adopting this pattern include:

Maintainability, simplicity, and adaptability

Each BFF is usually quite focused and can evolve specifically to meet the needs of a specific kind of client, improving both adaptability and maintainability. As a side effect, the backend services can be far more

generic when they aren't burdened by having to cater to client-specific logic, making them easier to maintain.

Reliability

Having a single backend for multiple clients is a single point of failure, so any failure in the backend would impact all clients. Having one BFF per client is better for isolating the consequences of a failure. The BFF can also implement retry and circuit-breaker logic, so upstream failures in services don't cascade down to the client.

Furthermore, a singular backend means only one set of service reliability metrics (SLIs, SLOs, and SLAs), which is usually reduced to the lowest common denominator for all clients. With a BFF per client, each BFF can implement its own set of service reliability metrics, catering to the needs of the specific client it serves.

Security

Each BFF can implement client-specific authorization and authentication mechanisms. For example, a mobile BFF might implement both password and biometric authentication, while a web BFF might only support password authentication.

Cons

The disadvantages of adopting this pattern include:

Responsiveness

A BFF introduces an additional network hop between the client and the backend services, which can add latency to requests. This can be mitigated, at least in part, using caching or optimized query patterns.

Reusability

As more and more BFFs are created, reusability can become a challenge, particularly for cross-cutting concerns such as error handling and logging. No matter the approach, any reuse strategy can introduce

additional coupling points and reduce teams' independence — and that independence is one of the primary benefits of the BFF pattern.

Comparison Chart

The following sections describe each of the capabilities found in Table 7-2. The "Without Pattern" column refers to a singular service acting as the backend for multiple frontend clients.

Table 7-2. Comparison chart for the Backend for Frontend pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Adaptability	★★	★★★★
Agility	★	★★★
Availability	★	★★★★★
Deployability	★	★★★
Error Handling	★	★★★
Evolvability	★	★★★★★
Fault Tolerance	★	★★★★★
Flexibility	★	★★★★★
Maintainability	★	★★★★
Simplicity	★★	★★★
Testability	★	★★★
DEGRADED		

Capability	Without Pattern	With Pattern
Cost	$$	$$$$
Responsiveness	★★★★	★★★
Reusability	★★★★★	★★

Improved

The capabilities that are improved when applying this pattern include:

Adaptability

Having different backends for different clients means that each backend can adapt to the clients' changing needs. Suppose an underlying architecture requires notifying customers when their orders ship. The mobile BFF team might choose to leverage push notifications. At the same time, the web BFF team might decide to use web push notifications for desktop clients. Each backend can be highly optimized for a particular use case.

This same adaptability can be extended to all kinds of other facilities. For example, different backends might use different data payload shapes or leverage caching to reduce network chattiness for bandwidth-constrained clients.

Agility

Agility is a composite capability that embodies various other capabilities, including deployability and testability. A singularly focused backend being released has less risk associated with it, improving its deployability. Isolating changes to individual backends reduces the blast radius of changes, which improves testability.

On the other hand, more granular BFFs can affect agility negatively, since reusability gets harder. Changes that affect all BFFs also require more team coordination, which impacts agility.

Availability

Separate backends for different frontends means avoiding having a single point of failure in the underlying architecture, as would happen with just one backend serving multiple clients. If the mobile backend goes down or is unavailable for some reason, other clients can continue to function normally: their BFFs remain operational and isolated from the mobile BFF's issues.

Deployability

Breaking a single backend for all user interfaces into multiple backends reduces the risk that a change targeting a particular client type might affect other types. This reduces the risk involved in deployment, improving deployability. Furthermore, it means that different BFFs can be developed and released at different cadences, which also helps deployability.

Error Handling

Having a specialized backend for each client means that the team can handle error conditions in a way that serves the client best. Different clients have vastly different capabilities for presenting and recovering from errors. The BFF pattern, with a specialized backend for each client, enables the team to tailor error-handling strategies. For example, if a downstream service becomes unavailable, a mobile BFF might choose to return cached data (where applicable) so that a user can continue working. A web BFF, on the other hand, might decide to return an error message that suggests retrying, or perhaps pop open a chat window for the user to contact a representative, leveraging the additional screen real estate that a desktop offers.

However, error handling is often a cross-cutting concern, and with more granular BFFs, maintaining consistent error handling across BFFs becomes more challenging due to reduced code reusability.

Evolvability

A BFF team can choose to swap out one backend for another, perhaps even switching an existing one out and replacing it with one written in a technology stack that best serves their or their clients' use case.

Fault Tolerance

Separate BFFs allow for isolation between BFFs: a fault in one does not cascade across the architecture to all services. Furthermore, each BFF can leverage different resilience patterns specifically tailored to their clients. A mobile BFF might fail fast rather than having the client retry, to save on power consumption and bandwidth usage. The web BFF, on the other hand, might send out appropriate HTTP response codes signalling that the client should retry, or even redirect them.

Flexibility

Each BFF team chooses its language, platform, integration protocols, and even feature prioritization, which significantly improves the flexibility of the overall architecture.

Maintainability

A single backend serving multiple clients typically becomes a complex codebase riddled with conditional logic ("if mobile client, do X; if web client, do Y") and overlapping concerns. Each change has an impact across all client types, and the need to understand that makes the code harder to reason about and modify.

BFFs eliminate this complexity through focused, single-concern codebases with singular interaction patterns.

Simplicity

Single-concern BFFs are inherently simpler to develop, maintain, and operationalize. They eschew the complexity that a single backend serving multiple client types inevitably accumulates.

Testability

The focused nature of BFFs significantly improves testability in several ways. Since each BFF serves a single client type with well-defined responsibilities, test suites can be more targeted and comprehensive. Feature enhancements or bug fixes are localized to a specific client, reducing the blast radius of changes and the scope of regression testing required.

Furthermore, teams can tailor their testing strategies to match each client's characteristics. For example, while web BFF tests might focus on browser compatibility and API contract testing, mobile BFF tests might require emulators for iOS and Android devices, and wearable BFFs might need specialized testing infrastructure for watches or fitness trackers.

Degraded

The capabilities that are degraded when applying this pattern include:

Cost

BFFs introduce multiple cost considerations that vary based on organizational context.

Using a single backend for multiple frontends minimizes the upfront architectural and design decisions, as well as development and infrastructure costs. The architecture contains one codebase to build, one service to deploy, and one set of resources to provision.

However, BFFs introduce several cost categories:

Design complexity

Choosing the right granularity (per UI versus per client type) and determining where shared behavior resides requires careful upfront analysis.

Development costs

BFFs introduce the need to analyze and assess how and when to share code, as opposed to simply duplicating code (see "Reusability" in this list). Sharing code increases coupling, and consequently coordination, since duplicating code across BFFs increases implementation time.

Coordination costs

Cross-cutting changes that affect multiple clients require tighter cross-team alignment, particularly since BFFs are organized around functional or UX concerns rather than technical domains.

Operational costs

Each BFF requires independent deployment pipelines, monitoring dashboards, incident-response procedures, and ongoing maintenance.

Infrastructure costs

Multiple services mean more compute resources, load balancers, monitoring systems, and associated cloud/hosting expenses.

The cost equation shifts depending on team size and organizational structure. For small teams with simple clients, a single backend may be more cost-effective. For larger organizations with specialized platform teams, the velocity gains and reduced coordination friction from BFFs often offset the infrastructure and development costs.

Responsiveness

Introducing a BFF usually means an additional hop between the client and backing APIs. This introduces latency, which affects responsiveness.

However, some of this can be offset. Perhaps the mobile BFF shapes its payloads differently than the web BFF, optimizing for better bandwidth usage or reduced chattiness.

Reusability

BFFs inevitably introduce code-duplication challenges, particularly for cross-cutting and shared concerns. Observability, error handling, authentication and authorization, rate limiting, and even business logic like orchestrating downstream service calls may all need to be implemented repeatedly, across each BFF. This duplication increases the maintenance burden, requires coordination when making changes, and creates opportunities for implementation drift between BFFs.

Teams have three main options for managing this duplication, each with distinct trade-offs:

Duplication

Accept the duplication and maintain separate implementations in each BFF. This maximizes team autonomy and flexibility but increases maintenance overhead.

Shared libraries

Extract common logic into shared libraries. This reduces duplication, but introduces coupling between BFFs through shared dependencies. It also creates versioning challenges and constrains teams to using only programming languages and runtimes supported by the shared libraries.

Shared services

Extract reusable logic into dedicated services, such as a centralized authentication service or a downstream aggregation service that handles common patterns of downstream calls. While this eliminates duplication in BFFs, it requires building, maintaining, and operating additional services, and introduces questions about service ownership and team boundaries.

Influences

This pattern has a couple of influences across team topologies and service granularity.

Team topologies

This pattern influences team topologies. When frontend teams own their BFFs, it significantly improves agility by giving each team control over feature prioritization, release cadence, testing, and deployment strategies. However, when separate teams support different BFFs but all rely on the same backend services, they need more coordination to implement this pattern successfully.

Consider a change to one or more backends that will most likely affect some or all BFFs. Or consider a new capability that all clients must roll out at the same time, leading to a "shared fate" (as referred to in the "Partnership" strategic integration pattern in Domain-Driven Design). This further increases cross-team coordination costs and may introduce bottlenecks.

Granularity

While this pattern generally improves granularity, it's always a consideration when implementing this pattern. As new clients are introduced (or existing ones are no longer supported), teams *must* reconsider their BFF configuration (refer back to "BFF Granularity") — should it be per user interface, per client type, or somewhere in the middle?

Having BFFs that are too fine-grained exacerbates the negatives, like reusability, while having large BFFs affects things like agility, evolvability, and maintainability, alongside a host of other capabilities.

Not only does this pattern affect granularity, it's also *influenced* by granularity, in that one of the motivating factors for building out BFFs is often the need to increase the granularity offered by monolithic backends.

How to Govern This Pattern

The key aspect of the BFF pattern is that *the client should never communicate directly with the backend services*. All communication *must* go through the BFF. Of course, the client and the BFF can be developed using any combination of languages and runtimes, so capturing all incoming and outgoing requests to and from the BFF sometimes requires getting clever with logging and tracing. Techniques like structured logging can help here. With logging in place, Example 7-1 shows an example of a pattern fitness function to govern client-to-BFF communication.

Example 7-1. Defining a governance rule to monitor communication between client application and BFF

```
var ios_app
var mobile_bff

ASSERT(ios_app only communicates with mobile_bff)
```

Installing such a governance function ensures that clients do not inadvertently bypass their respective BFFs to communicate directly with backend services, which would affect several of the capabilities this pattern brings to the table, such as availability, error handling, flexibility, security, and simplicity.

Related Patterns, Antipatterns, and Pitfalls

The following pattern and pitfall are related to the BFF pattern:

Cache Aside pattern (see Chapter 8)

One of the trade-offs introduced by this pattern is responsiveness, given that the BFF represents an additional hop. Introducing a cache into the mix can improve responsiveness, particularly if the kinds of data and read/write patterns involved lend themselves well to caching. Furthermore, if the client fronting the BFF needs a different data model than the one exposed by the backend service, caching (particularly of static data) could help avoid transforming the data on every ask.

Grains of Sand pitfall (see Chapter 21)

While not directly related to this pattern, the Grains of Sand pitfall is a great reminder to think about the granularity of services or BFFs and not take things too far.

Chapter 8. Cache Aside Pattern

The Cache Aside pattern, commonly referred to as "lazy loading," is one of many caching patterns. It's designed to optimize caching for frequently accessed data and lends itself well to read-heavy workloads.

Table 8-1 summarizes the metadata for the Cache Aside pattern.

Table 8-1. Metadata for the Cache Aside pattern

CATEGORY	METADATA
ALSO KNOWN AS:	Lazy Loading
CONTEXT:	Caching, Data, Event-Driven Architecture, Message-Based Architectures, Microservices, Monolithic Architectures
IMPROVED:	Elasticity, Flexibility, Responsiveness, Scalability
DEGRADED:	Data Consistency, Data Integrity, Simplicity

How Do I Improve Both Performance and Scalability?

The Cache Aside pattern is a familiar caching pattern in which the application code is responsible for reading and writing from the cache, in addition to the primary datastore.

Applying the Pattern

The Cache Aside pattern has the usual components of a caching pattern: a cache, a primary datastore, and an application that needs to read and write data. In this pattern, however, the application code orchestrates reads and writes between the primary datastore and the cache. Architects must carefully consider this when adopting this pattern.

All caching patterns cater to both reads and writes (also called *updates*). Let's look at both in the context of the Cache Aside pattern, one at a time.

Reads

From an implementation perspective, the Cache Aside pattern's commonly used moniker, "lazy loading," is a good hint for how the pattern works and how it should be implemented. The sequence diagram in Figure 8-1 illustrates the "happy path" — that is, where the sought item is already in the cache.

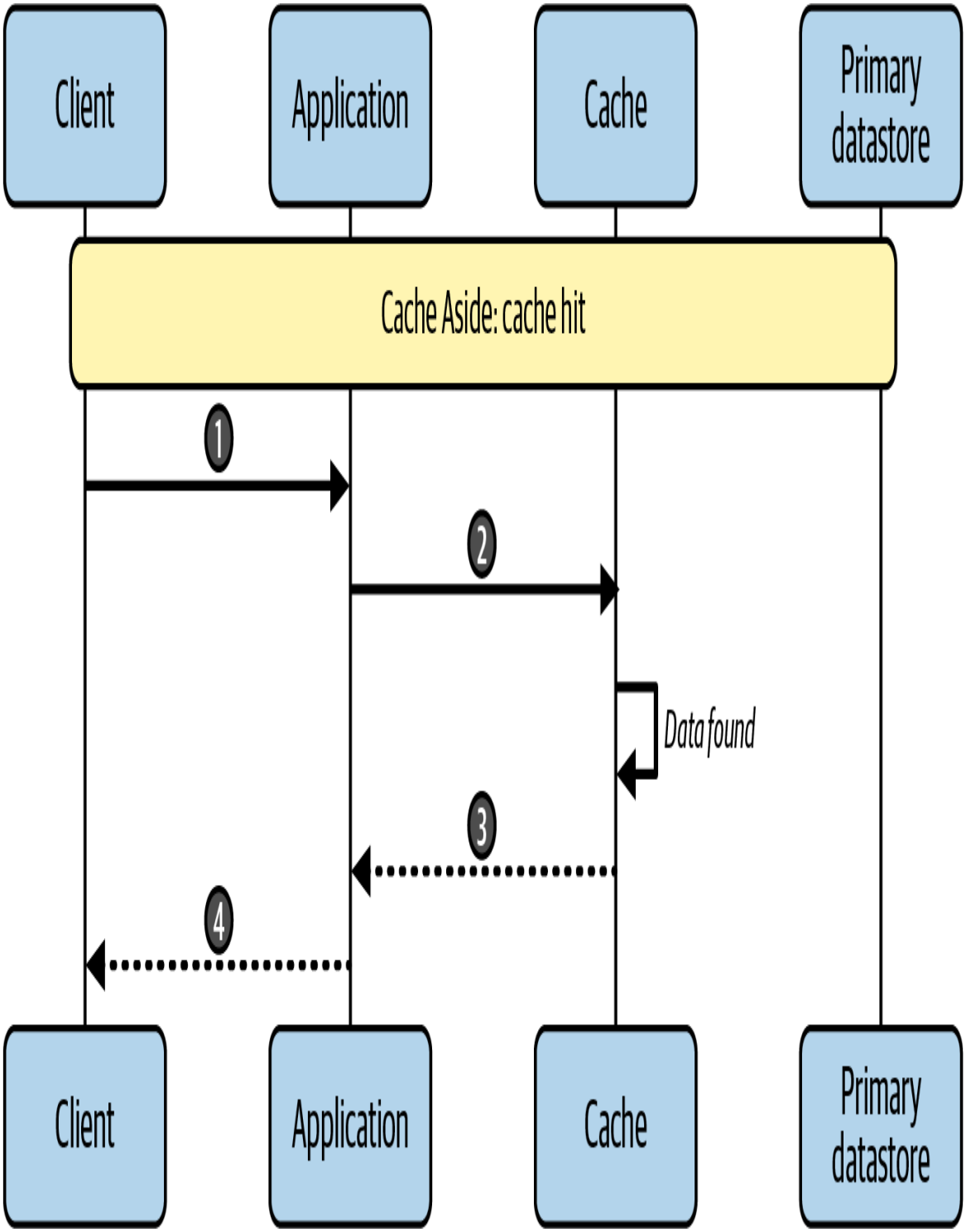


Figure 8-1. Cache Aside pattern reads a sequence diagram for a cache hit

As depicted in Figure 8-1, the sequence of events for reading is as follows:

1. The client requests data from the application.
2. The application checks the cache for the requested data.

❸ The cache looks up the requested data and returns it to the application.
❹ The application returns the data to the client.

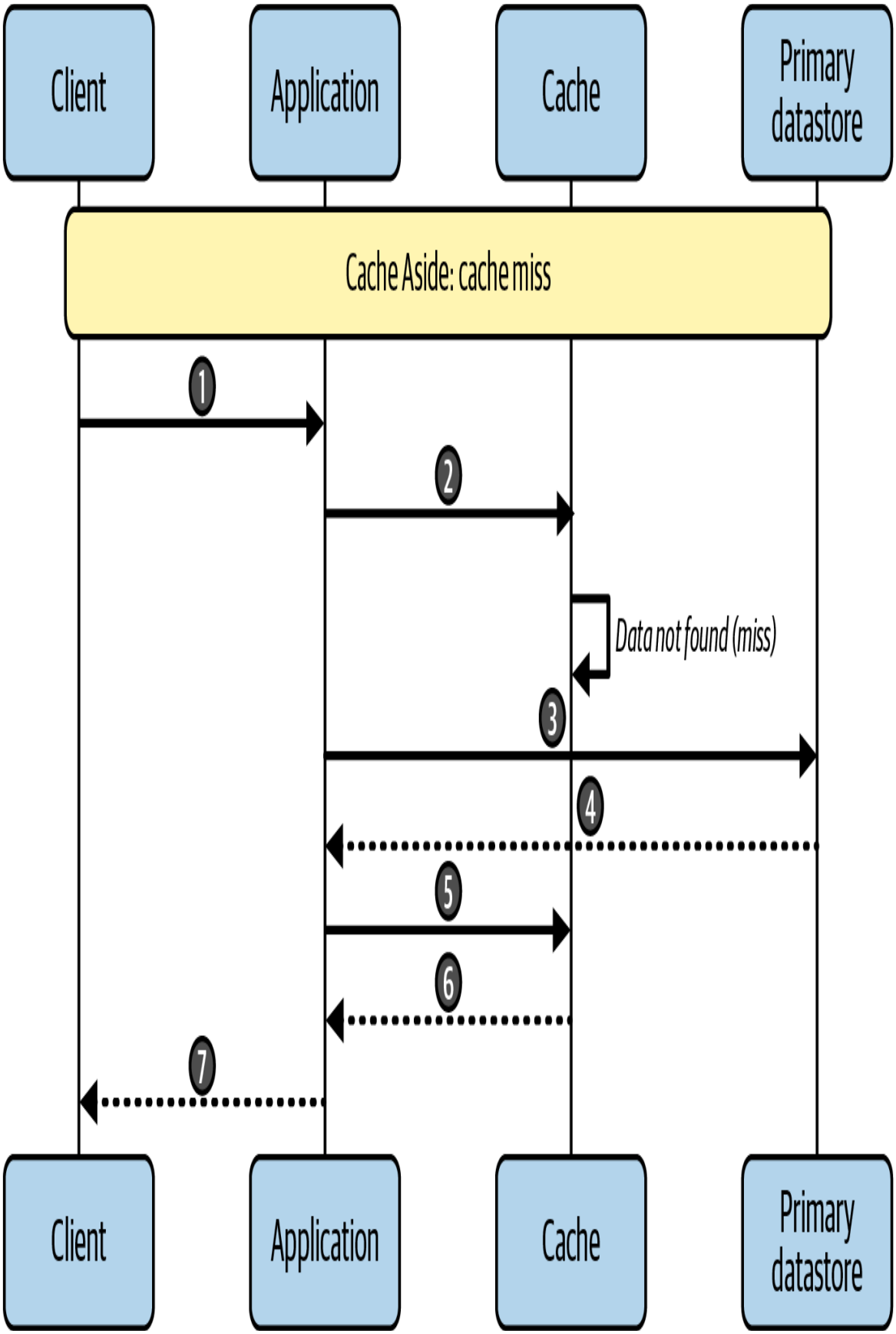

Client
Application
Cache
Primary datastore
Cache Aside: cache miss
1
2
Data not found (miss)
3
4
5
6
7
Client
Application
Cache
Primary datastore

Figure 8-2. Cache Aside pattern reads a sequence diagram for a cache miss

Figure 8-2 illustrates what happens on a cache miss:

1. The client requests data from the application.
2. The application checks the cache for the requested data.
3. If the data is *not found* in the cache, the application attempts to retrieve it from the primary datastore.
4. The database returns the data to the application.
5. The application stores a copy of the retrieved data in the cache for future requests.
6. The cache acknowledges a successful save.
7. The application returns the data to the client.

The Cache Aside pattern starts with an empty cache, which it only populates upon a miss. This is the "lazy loading" aspect of the pattern. The application is also responsible for orchestrating reads between the cache and the primary datastore, and for populating the cache on a miss. All this happens synchronously, in that calls to the cache and the database are blocking calls.

Writes/Updates

On the write side, data may be new or an update to existing data, which may or may not be already in the cache. The sequence diagram in Figure 8-3 illustrates the "happy path" for writes to the primary datastore in the Cache Aside pattern.

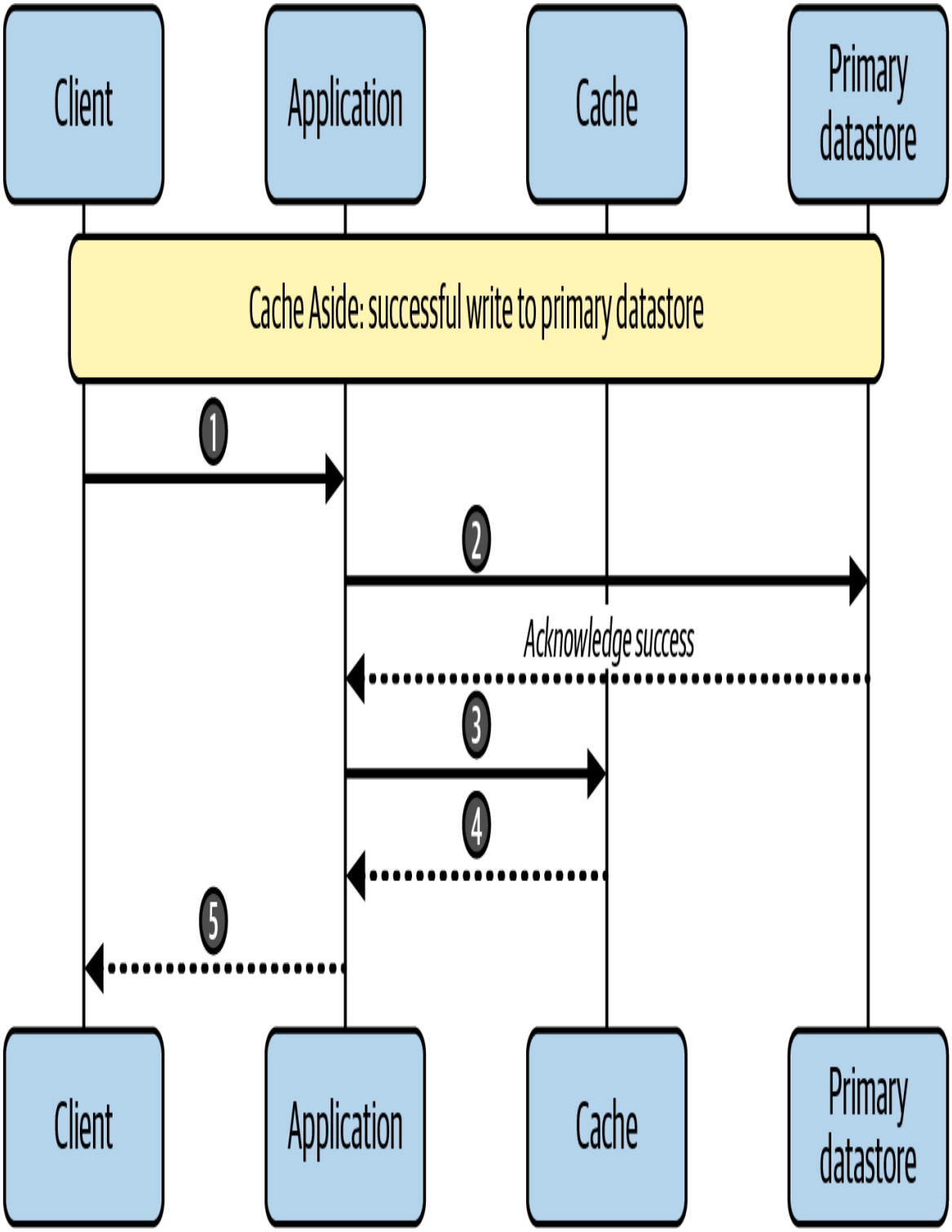


Figure 8-3. Cache Aside pattern write/update success path sequence diagram

The sequence of events is as follows:

❶ The client sends new or updated data to the application.

❷ The application writes the new data or updates existing data in the primary datastore.
❸ If the write is successful, the application *invalidates* the corresponding cache entry on an update.
❹ The cache acknowledges a save.
❺ The application acknowledges to the client that the write was successful (for example, with a `200 OK` HTTP status code).

The less happy path, shown in Figure 8-4, illustrates the sequence of events if the write to the primary datastore fails:

❶ The client sends new or updated data to the application.
❷ The application attempts to write the new data to or update the existing data in the primary datastore, which fails.
❸ The application does not update the cache, but acknowledges a failure to the client.

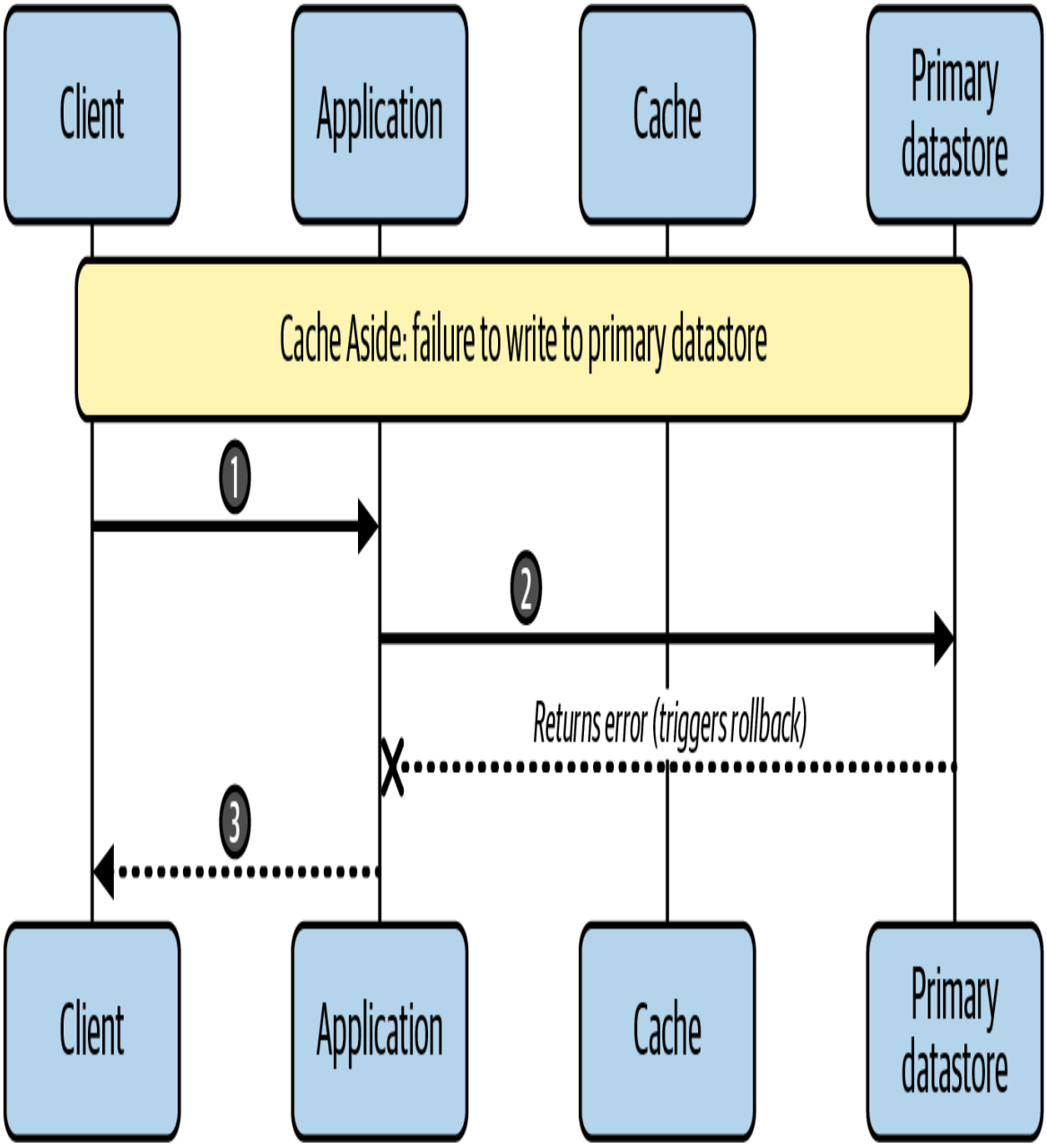


Figure 8-4. Cache Aside pattern write/update failure path sequence diagram

In case of a failure to write to the primary datastore, the cache is left unchanged. By doing this, the pattern guarantees that the next read will not only be a cache hit but should return the value stored in the primary datastore.

A couple of characteristics are worth highlighting here. First, the cache is invalidated only *after* the write to the primary datastore succeeds. In other words, the application should guarantee atomic consistency between the

primary datastore and the cache. This also means that the next read on that particular cache entry will be a miss — and that, after the read sequence, the updated entry will be lazy-loaded into the cache. The application logic can choose to update the cache entry instead of invalidating it, but this decision comes with its own set of trade-offs and must be made judiciously. Whether the cache entry is invalidated or updated, all operations are still synchronous (that is, blocking in nature). Again, it's the application code's responsibility to orchestrate writes and updates between the primary datastore and the cache.

The Cache Is Not in the Critical Path

In the Cache Aside pattern, the cache is *not* and *should never* be the system's source of truth. This means the onus is on the application to ensure that writes and updates happen in the primary datastore *first*, with the cache updated or invalidated only afterward. Conversely, if the write to the primary store fails, then the cache should not be updated either. This implementation detail can be verified easily using an integration test. In this test, both the primary datastore and cache should be mocked out, with the mock datastore programmed to return a failure. When the write method is invoked, the test should assert that, although the write invoked the database mock, since the write failed, the subsequent mock for the cache was not invoked. This is demonstrated in Example 8-1.

Example 8-1. Pseudocode to test that the cache is not written to if primary datastore write/update fails

```
TEST "Ensure cache is not written to if primary datastore
write/update fails"
  CREATE dataStoreMock
  CREATE cacheMock
  CREATE serviceInstance with dataStoreMock and cacheMock injected

  THROW (new DatabaseException()) WHEN dataStoreMock.saveData(any)

  RESULT = serviceInstance.writeData("testKey", "testValue")

  ASSERT RESULT IS FAILURE
```

```
ASSERT dataStoreMock WAS CALLED WITH "testKey" and "testValue"
ASSERT cacheMock WAS NOT CALLED
```

Handling Cache Failures

The idea that the cache is not in the critical path has implications for how the system should handle cache operation errors. Consider reads: if a cache read operation fails due to, say, a timeout, the application should simply treat this as a cache miss. While doing so will increase the latency on the request, it emphasizes the core idea that the cache is not in the critical path.

Similarly, if a failure happens while updating or invalidating cache entries, write operations to the primary datastore should not be affected. However, if a cache entry is incorrectly invalidated, the cache might serve stale data on subsequent reads. Architects should consider appropriate remediation strategies, such as retry logic or simply alerting other downstream systems to allow for manual intervention, and codify them within the application's cache-interaction logic.

Write-Around Caching Pattern Variant

The Write-Around Caching pattern is another caching pattern often mentioned alongside the Cache Aside pattern. It employs the same read workflow as the Cache Aside pattern, but departs significantly from it in the write/update workflow. Recall that the Cache Aside pattern lazy-loads items into the cache. Thus, it only caches items that clients have requested. On an update, however, it first updates the primary datastore, then *invalidates* (or potentially updates) the cache.

In the Write-Around Caching pattern, items are lazy-loaded just like in the Cache Aside pattern. However, on a write, the Write-Around Caching pattern typically skips the cache altogether. The sequence diagram in Figure 8-5 illustrates writes and updates in the Write-Around Caching pattern.

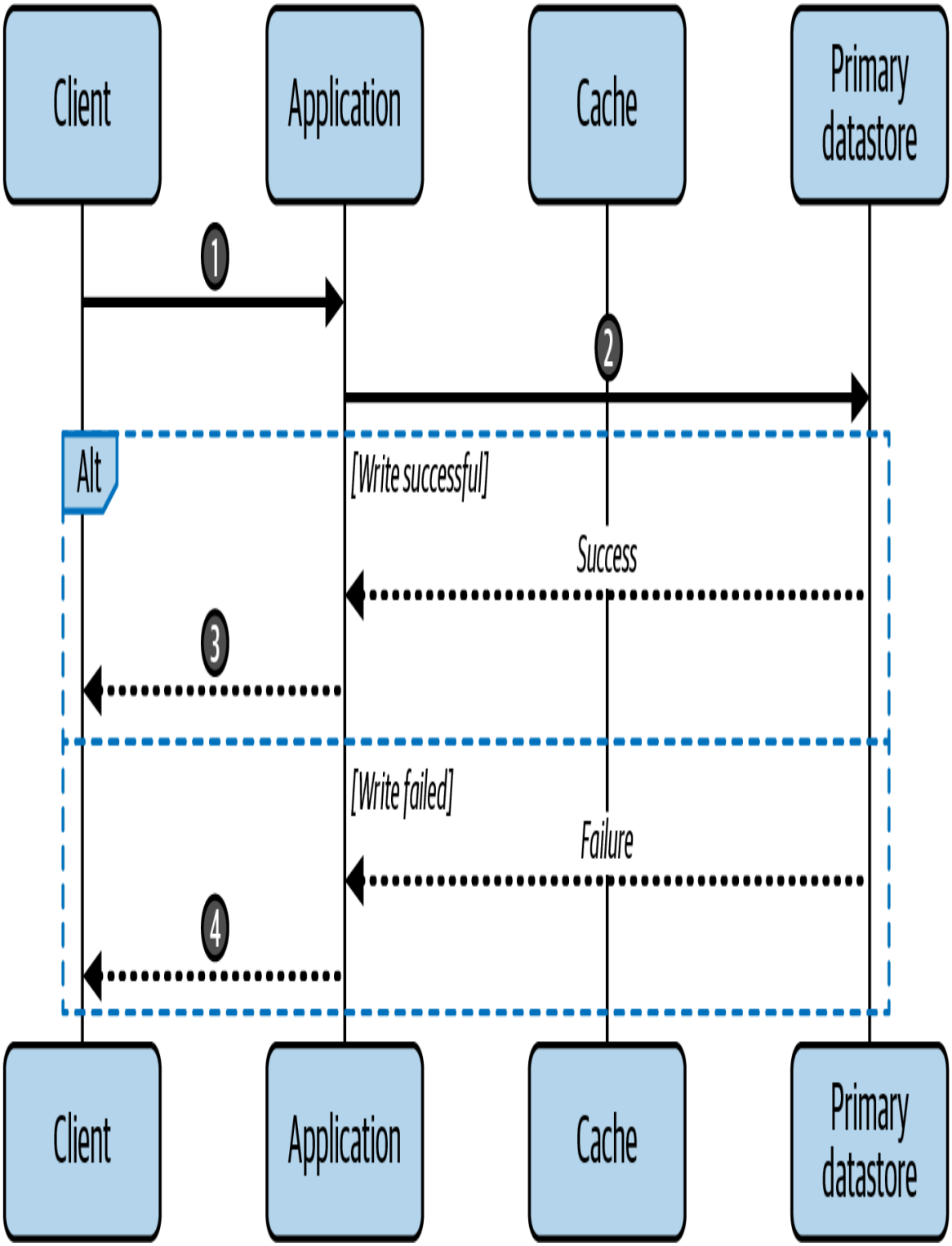


Figure 8-5. Write-Around Caching pattern write/update sequence diagram

Sequentially, the steps are:

1. The client sends new or updated data to the application.

❷ The application writes the new data or updates the existing data in the primary datastore.
❸ If the write/update is successful, the application responds to the client that the write was successful (perhaps with an HTTP `200` message).
❹ If the write/update to the primary datastore fails, the application acknowledges the failure to the client.

As Figure 8-5 shows, the cache has no role to play in this workflow. The application simply *writes around* the cache. An update is highly likely to affect a previously cached entry. Since the cache doesn't participate in the update sequence, it could serve up stale data on the next request. One technique for avoiding this is to maintain an aggressive time-to-live (TTL) or set eviction policies.

As we mentioned earlier, pure Write-Around Caching implementations skip the cache entirely. But, in hybrid approaches, implementations may invalidate the recently updated data in the cache without repopulating it with updated data. Just like in the Cache Aside pattern, this should result in a cache miss on the next read.

The Write-Around Caching pattern is very effective for write-heavy workloads where items are infrequently accessed.

Trade-Offs

The Cache Aside pattern is popular, perhaps simply because people think of a *cache* as a place to store often-used items. However, it does bring several advantages and disadvantages to the table:

Pros

The advantages of adopting this pattern include:

Flexibility

Since the application contains the logic of what and what not to cache, it can be even more stringent about what enters the cache. For example, it might preemptively load "hot" (or frequently used) items into the cache to improve response times. The application can also set the TTL for each entry individually, choosing to set longer TTLs for "popular" entries and shorter ones for those that change quickly. These settings can greatly improve cache-hit ratios and system responsiveness, while optimizing cache size and cost.

Cons

The Cache Aside pattern shifts the responsibility of managing which items will be cached onto the application. This comes with several downsides:

Data consistency and integrity

The TTL in the cache represents a window in which the cache and primary datastore may be mutually inconsistent. At scale, in systems with highly dynamic data, this inconsistency is easily exacerbated. For example, assume that a popular rideshare application, in an attempt to maximize revenue, applies surge pricing during times of increased demand, like rush hour or the end of a major event. Computing the price of a ride can be a complex operation involving real-time traffic and demand data, driver availability, starting and ending locations, and more. To optimize responsiveness and efficiency, the application caches pricing data for specific routes. When demand surges and multiple requests for similar routes arrive simultaneously, without an aggressive TTL, the application may end up serving stale or incorrect pricing information. Failing to keep up with increased demand could cost the company revenue.

Complexity

Making the application logic responsible for managing the cache adds complexity to the system. In high-scale environments, especially with dynamic data, there might be competing requests for the same data. In

such cases, a cache miss can lead to a "cache stampede,"[1] which can overwhelm the primary data source. This defeats the purpose of caching, which is to prevent the system from being overloaded! This is usually mitigated by implementing a locking mechanism, such as request coalescing within the application logic.[2] However, this does introduce significant complexity to the application.

Comparison Chart

Table 8-2 shows the qualitative ratings for various capabilities when using this pattern with a cache, versus avoiding a cache altogether and using only a data source. It must be noted that the With Pattern column is essentially an average of the breakdown indicated in Table 8-3.

Table 8-2. Comparison chart for the Cache Aside pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Elasticity	★★	★★★★
Flexibility	★★★	★★★★
Responsiveness	★★	★★★
Scalability	★★	★★★★
DEGRADED		
Data Consistency	★★★★★	★★★
Data Integrity	★★★★★	★★★
Simplicity	★★★★★	★★

The following sections describe each of the capabilities found in Table 8-2.

Improved

The capabilities that are improved when applying this pattern include:

Elasticity

> Using a cache can significantly help with bursty traffic, especially with the correct caching topology. With a cache, the datastore can be insulated from surges in traffic, helping with elasticity.

Flexibility

Flexibility is improved with the Cache Aside pattern, since the application can decide which items should reside in the cache and for how long (perhaps using different TTLs for different items).

Responsiveness

Caching "hot" or frequently read items speeds response times on reads for these items. Cache misses, however, come with a penalty: having to reach across to the primary datastore to fetch an item may incur a network hop.

Scalability

Having a cache prevents the primary datastore from being a bottleneck by relieving it from always having to look up and serve frequently read data, particularly for read-heavy workloads. While this can help with scalability, its trade-offs are worth considering, particularly in light of the caching topologies used (see "Caching Topologies").

Degraded

The capabilities that are degraded when applying this pattern include:

Data Consistency

Using a cache always carries a risk of serving stale data, particularly at scale. This risk can be exacerbated by the kind of data being cached and the caching topologies being used. Architects will need to be very astute when it comes to fine-tuning TTLs and eviction policies, and reconsider them regularly as the architecture evolves over time.

Data Integrity

The cache and the primary datastore could always fall out of sync, particularly after updates or if cache-invalidation strategies are not well designed.

Simplicity

Having the application interact with the cache, ensuring atomicity between the primary datastore and the cache, and perhaps even using specialized algorithms to select what data to cache can add a lot of complexity. However, at scale, having just a primary datastore could become a bottleneck, so architects must think through how to manage that.

Influences

Caching topologies have a significant influence on this pattern's effectiveness and on what architects must consider.

Caching Topologies

Let's consider a few caching topologies and their considerations and trade-offs when using the Cache Aside pattern:

Single in-memory cache

Implementing the Cache Aside pattern with a single in-memory cache is straightforward and works really well in single-instance scenarios. Read and write latencies to the cache will be extremely fast, since the cache resides in the same process as the application itself. On a cache miss, the latency will be primarily dictated by the cost of a network hop to the primary datastore. Since this cache is in memory, restarts would mean a cold cache and, consequently, high cache miss rates until the cache is sufficiently warmed.

In cases where scalability is achieved *horizontally* (that is, multi-instance applications), architects must be careful about what kind of data is cached. While this pattern lends itself well to caching static data or data scoped to a request, dynamic data can quickly become problematic. This is because, in the Cache Aside pattern, the cache is indeed set *aside* from the primary datastore. Any updates thus only

affect the local (in-memory) cache. This can lead to inconsistency between various instances of the application, each of which has its own cache. While this can be mitigated using techniques like sticky sessions or by caching only specific kinds of data, these techniques might counter some of the expected gains from horizontal scaling.

Finally, each instance having its own cache leads to larger memory footprints and higher costs.

Client-server cache (remote cache)

The Cache Aside pattern is better suited for working with the client-server cache than with a single in-memory cache, especially when scalability is achieved through horizontally scaling out application instances. However, this too comes with trade-offs.

When restarting a single-instance setup, the application will wake up to an already warmed-up cache. In a multi-instance setup, since the cache is shared, new replicas of the application will also benefit from a prepopulated cache.

Having all instances share the same cache avoids data duplication, which can help limit cache size and costs. A single cache is also a lot simpler to update or invalidate — all it takes is one instance to update the cache, and other instances benefit.

On the flip side, read latency on cache misses will be higher, since the application needs to make multiple network hops: first to the cache only to get the miss, followed by a call to the primary datastore, then returning to the cache to make an entry before responding to the request. Writes take longer as well, since invalidating or updating the cache is another network call.

At scale, since all instances of the application share the remote cache, it can become a bottleneck or even a single point of failure. Any performance degradation in the cache will ripple across the entire architecture. Furthermore, any network latency or cache sluggishness could easily cause a cache miss to trigger a cache stampede, which

could overwhelm the primary datastore — especially if no guardrails are present.

Replicated in-memory cache

Using the Cache Aside pattern with a replicated cache can be the best of both worlds for high-scale applications, but the trade-offs are big increases in cost and complexity.

Having an in-memory replica of the cache keeps read and update latencies low, and the only hop needed is when interacting with the primary datastore.

Since the cache is automatically replicated across all instances, restarts and additional instances can benefit from a prepopulated cache. Automatic cross-instance consistency means that the system doesn't have to rely on techniques like sticky sessions or locking mechanisms: all instances will eventually have the same data in their cache. This has a direct positive effect on the whole system's availability, scalability, and elasticity. However, since each application has an in-memory replica, the memory footprint per instance is larger.

Consistency guarantees are a function of the type and architecture selected for the cache. Some replicated in-memory caches offer strong consistency, using techniques like leader selection. However, this comes at the cost of latency at write time: all writes must go through the leader, since writes to the cache in the Cache Aside pattern are typically blocking calls.

Alternatively, some replicated in-memory caches will opt for eventual consistency, thereby significantly improving latency and responsiveness. This option brings with it all the complexity of eventual consistency (versus strong consistency), and the associated challenges of maintaining synchronization across all replicas. In this model, after one instance updates the cache, other instances may temporarily serve stale data until replication propagates. One solution to this is to publish invalidation messages to notify other instances to delete the cache entry.

Another is to install extremely aggressive TTLs on cache entries (particularly on dynamic data) to evict potentially stale data automatically. Both options affect latency and responsiveness, since reads across replicas for recently updated data will be cache misses until repopulated.

Lastly, as in any distributed state store, a conflict-resolution strategy is needed in case of multiple concurrent writes to the same piece of data.

Overall, the Cache Aside pattern with a replicated in-memory cache will consume more bandwidth and incur more network overhead due to cross-instance chatter and a higher memory footprint (and therefore a higher cost) per instance.

Operationally, all of the complexities of managing a distributed system emerge when attempting to use the Cache Aside pattern with the replicated in-memory cache.

Caching Topology Comparison

Table 8-3 compares how the Cache Aside pattern performs across three different caching topologies.

Table 8-3. Comparison chart for the Cache Aside pattern across caching top

Capability	With Pattern	Single In-Memory Cache	Client-Server Cache (Remote)	Rep In-M Cac
IMPROVED				
Elasticity	★★★★	★★	★★★★	★★
Flexibility	★★★★	★★★★★	★★★★	★★
Responsiveness	★★★	★★★	★★	★★
Scalability	★★★★	★★	★★★★	★★
DEGRADED				
Data Consistency	★★★	★★	★★★★	★★
Data Integrity	★★★	★★	★★★★	★★
Simplicity	★★	★★★	★★★	★

Improved

The following list outlines the capabilities that improve across caching topologies when using this pattern:

Elasticity

Elasticity is about how quickly new instances of applications or services can be spun up to accommodate bursty traffic. Here, in-memory cache suffers, since new instances will have a cold cache. In other words, new applications take a while to achieve the same responsiveness as their peers.

Flexibility

Overall, this pattern exhibits great flexibility regardless of caching topology, since the application has full control over the kinds of data and items being cached.

Responsiveness

Caching in general improves responsiveness. This can be observed across all caching topologies.

Scalability

Scalability suffers for in-memory cache due to a lack of automatic replication, forcing architects to use strategies like sticky sessions or extremely aggressive TTLs. For both client-server and replicated in-memory caches, new or restarted application instances benefit from a warm cache.

Degraded

The following list outlines the capabilities that degrade across caching topologies when using this pattern:

Data Consistency

The client-server cache topology is a natural choice here, finding a good balance between the other two topologies. However, the single in-memory topology suffers here, since any workarounds to improve consistency will significantly increase the complexity of the overall system. As for replicated in-memory, it varies: some architectures naturally offer strong consistency as opposed to eventual consistency.

Data Integrity

> Maintaining data integrity across cache topologies requires ensuring that updates or cache-invalidation strategies prevent serving out-of-sync or corrupt data across application instances.

Simplicity

> No caching topology can simplify what this pattern actually asks: that the application still manage interactions with both the primary datastore and the cache. The replicated in-memory caching topology further complicates the picture by introducing a distributed system that attempts to attain either strong or eventual consistency.

How to Govern This Pattern

Architecturally, the key characteristic of this pattern is that the cache is set *aside* from the primary workflow, which has implications for the system's overall reliability. Since the cache is not in the critical path, its becoming unavailable should not affect overall system availability, though it might impair responsiveness. However, the system should still be available to service requests. This aspect can be governed using techniques such as *chaos engineering*, which involves simulating cache unavailability while verifying the system's ability to handle requests without the cache.

In addition to ensuring that the system is topologically correct, architects should consider building out additional fitness functions to measure cache effectiveness, memory usage, and cost. At scale, putting appropriate checks and alerts in place helps to prevent cache stampedes, which allows architects to better tune the TTL and apply eviction policies appropriately.

Related Patterns, Antipatterns, and Pitfalls

The following patterns are related to the Cache Aside Pattern:

Read-Through/Write-Through Cache pattern (see Chapter 29)

When choosing a caching pattern, architects must consider workload patterns, the kind of data being cached, and a multitude of other factors. The Read-Through/Write-Through Cache pattern is an alternative caching pattern, as are its variants.

Event Sourcing pattern (see Chapter 16)

The Cache Aside pattern is commonly used in Event Sourcing architectures for managing projections and read models. Projections are materialized views built by replaying events, and they function as cached representations of aggregate state optimized for specific query patterns.

The lazy-loading characteristic of Cache Aside aligns naturally with how projections are typically managed: they're built on demand and can be invalidated or updated when new events are appended to the event stream. The Cache Aside pattern's flexibility allows Event Sourcing systems to maintain only the projections that are actively being queried, rather than maintaining every possible view of the data. This helps optimize memory usage and operational costs in Event Sourcing architectures.

---

1 Also known as the "thundering herd," this is a scenario where multiple concurrent requests arrive simultaneously, seeking the same piece of data and overwhelming the primary datastore.

2 *Request coalescing* is a technique where duplicate concurrent requests for the same data are combined into a single backend request, with all requesters waiting for the result and receiving it once it's fetched.

Chapter 9. Choreography Pattern

Choreography is one of two primary communication patterns for distributed architectures such as event-driven and microservices (the other is the Orchestration pattern, covered in Chapter 27). In the Choreography pattern, domain services coordinate directly on workflows to handle state, boundary conditions, and errors, eschewing external coordination.

The table in Table 9-1 summarizes the metadata for the Choreography pattern.

Table 9-1. Metadata for the Choreography pattern

CATEGORY	METADATA
ALSO KNOWN AS:	Broker
CONTEXT:	Event-Driven Architecture, Message-Based Architectures, Microservices, Workflow
IMPROVED:	Adaptability, Availability, Domain Alignment, Elasticity, Evolvability, Fault Tolerance, Responsiveness, Reusability, Scalability, Throughput
DEGRADED:	Data Consistency, Data Integrity, Debuggability, Error Handling

How Should I Manage My Workflows?

In both microservices and event-driven architectures, an obvious and intuitive communication pattern is Choreography, which is generally contrasted with Orchestration. With dancers, a choreographer creates a plan for what steps the dance should include — the actions of the participants aren't arbitrary. However, unlike in an orchestra, which has a conductor, a choreographed dance has no central coordinating force. The Choreography and Orchestration patterns in software architecture work similarly. In Choreography, each service must know how to react to workflows and exceptional conditions.

Applying the Pattern

The building blocks for the Choreography pattern are basically the same as those in microservices and event-driven architectures — a collection of domain services or event processors. The difference in this pattern is that there's no coordinating service, like an orchestrator. Each service listens for particular events to occur within the architecture and reacts to them by performing some processing and, potentially, generating new events. The general layout for a product workflow appears in Figure 9-1. The only architectural elements are services and event queues.

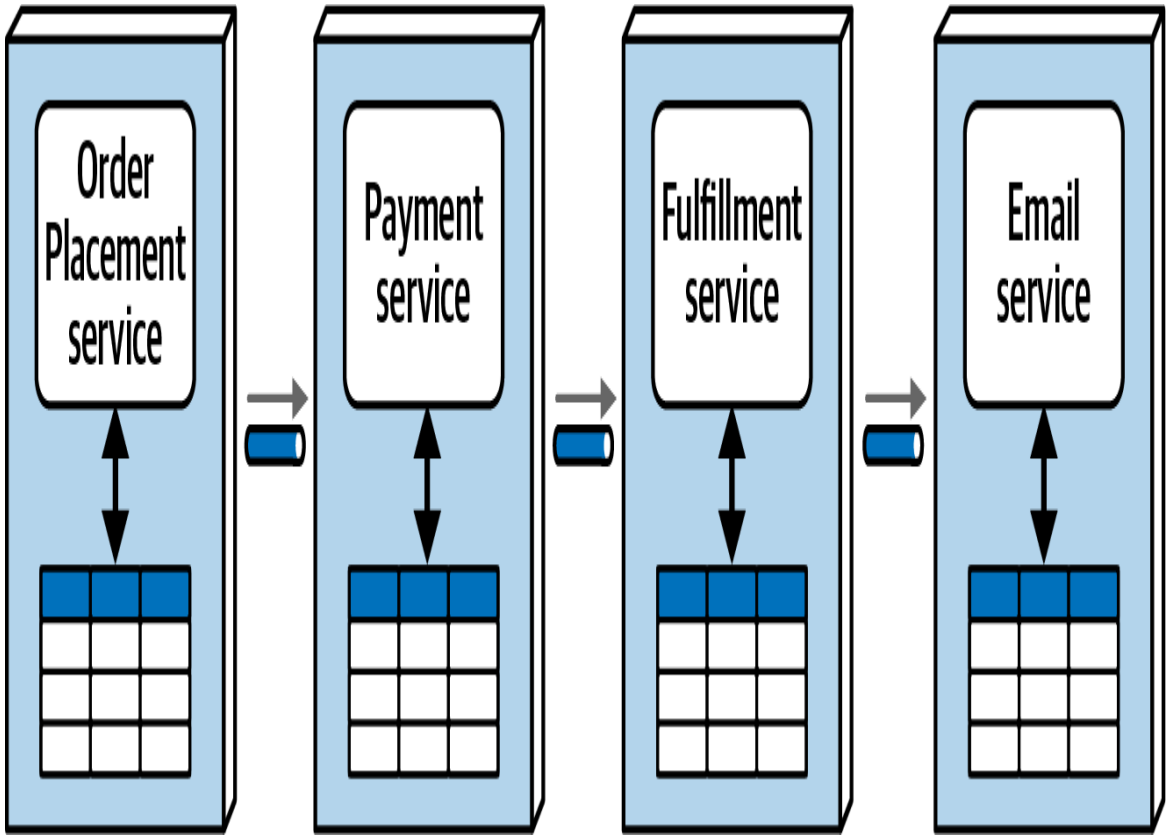


Figure 9-1. The general topology of a choreographed architecture

What Happens When Everything Works

We've modeled the happy path for the now-familiar workflow for ordering a product (such as a book) in several chapters. That communication workflow in choreography is illustrated in Figure 9-2.

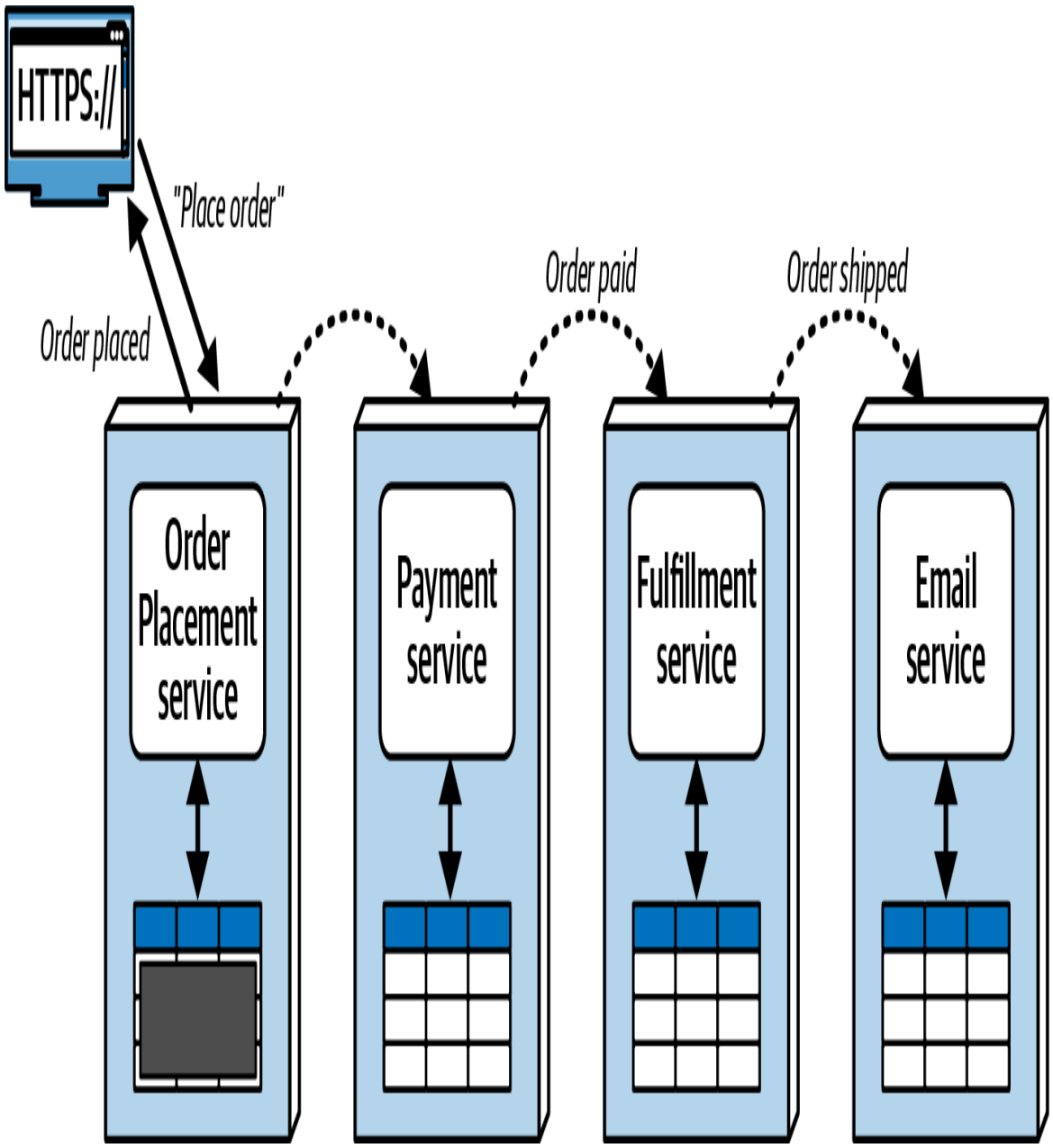


Figure 9-2. A simple choreographed product-ordering workflow

In Figure 9-2, the web frontend places the order via the `OrderPlacement` service, which updates its internal state to reflect the order. Then, typically using events rather than messages (see "Events Versus Messages"), that service advertises the fact that the order has been placed. In Figure 9-2, the `Payment` service listens for an event emitted by `OrderPlacement` and reacts to it, applies payment, and then generates

another event. This chain of events continues through the `Fulfillment` and `Email` services.

In each case, domain services listen for and react to relevant messages. That means it's trivial to add new behaviors to these types of workflows. For example, if an `Audit` service is needed to record each payment, it can also consume the events emitted by the `Payment` service. In fact, the most common design decision architects make when using events for communication is whether to make that communication point-to-point or multicast, which allows for many consumers and easy extensibility.

When everything works, events propagate around this architecture in a practically organic way, with services that react just in time to relevant events. This is attractive to many architects because of its simplicity and (seeming) lack of moving parts, especially compared to Orchestration (Chapter 27). However, as we consider the less happy path — the one where something breaks — it will become more apparent why we *also* discuss error pathways and boundary conditions.

What Happens When an Error Occurs?

Error-handling logic must reside somewhere within the domain services. When there's an orchestrator, the architect can dispatch error conditions to it when an error or boundary condition occurs. However, in the absence of a dedicated service, each domain service must know how to react.

In Figure 9-3, the `OrderPlacement` service issues an event and `Payment` reacts. `Payment` then issues an event to which `Fulfillment` reacts. However, `Fulfillment` can't handle the order, for whatever reason, so now this workflow can't succeed. Unlike in Orchestration, in Choreography the `Fulfillment` service must contain logic to handle this contingency: for example, issuing a compensating update to undo the `Payment` event destined for the `Payment` service. Either the `Fulfillment` or `Payment` service should also issue an event for the `OrderPlacement` service. Which one? That's an implementation detail.

(We cover the added complexity of compensating updates using Choreography in Chapter 11.)

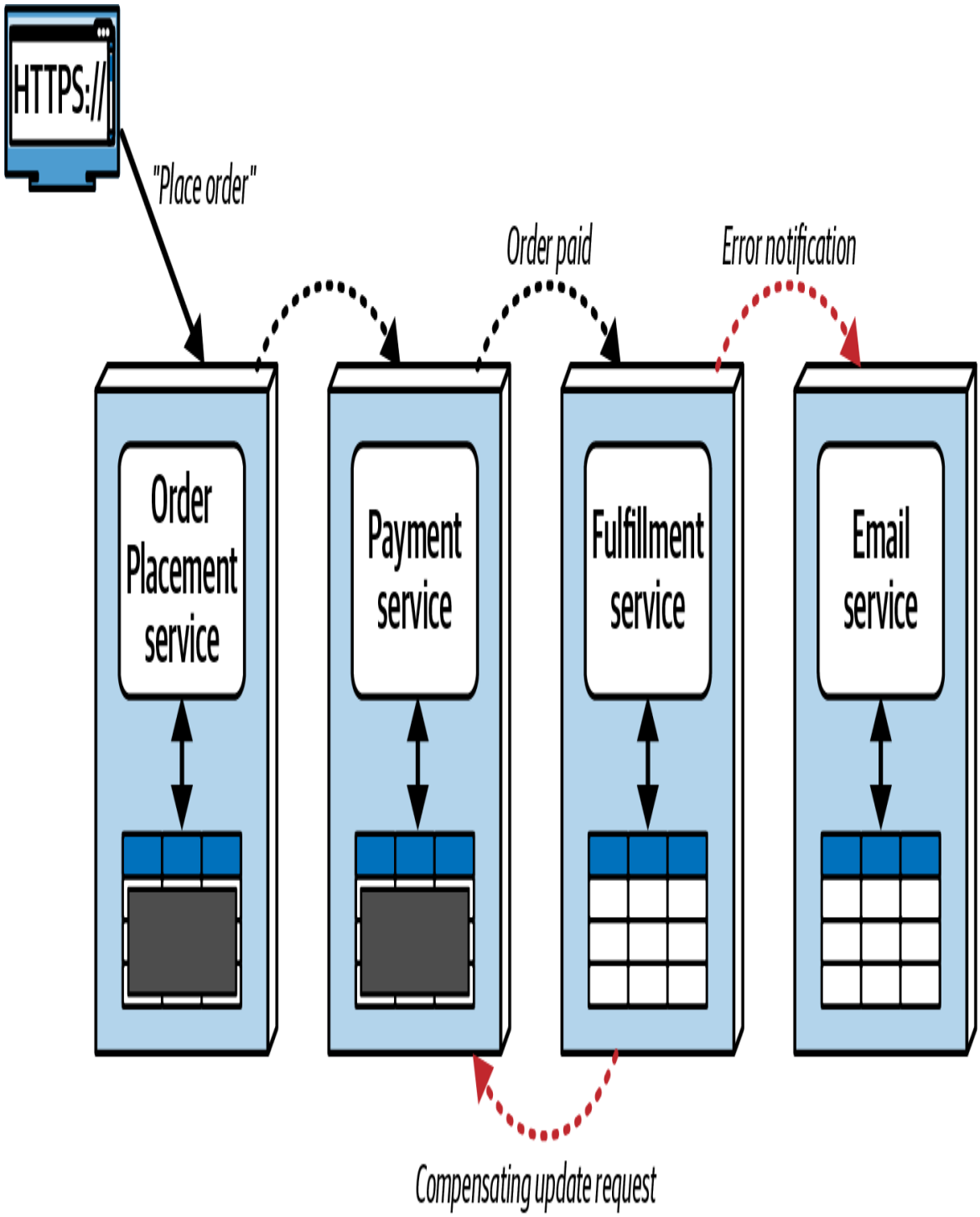


Figure 9-3. When an error occurs, each domain service must react accordingly

However, the scenario shown in Figure 9-3 is the "happy" error path — it shows how an error will be handled if nothing *else* goes wrong. But consider the scenario illustrated in Figure 9-4, where something unlikely but possible has occurred: after the payment was issued, but before the compensating update request, someone made a fraudulent charge on the user's credit card, so the bank invalidated that card number and issued a new one. The `Fulfillment` service immediately issued a request to `Payment` to undo the charge, but the payment could not be reversed because the credit card number was no longer valid. What should the `Payment` service do?

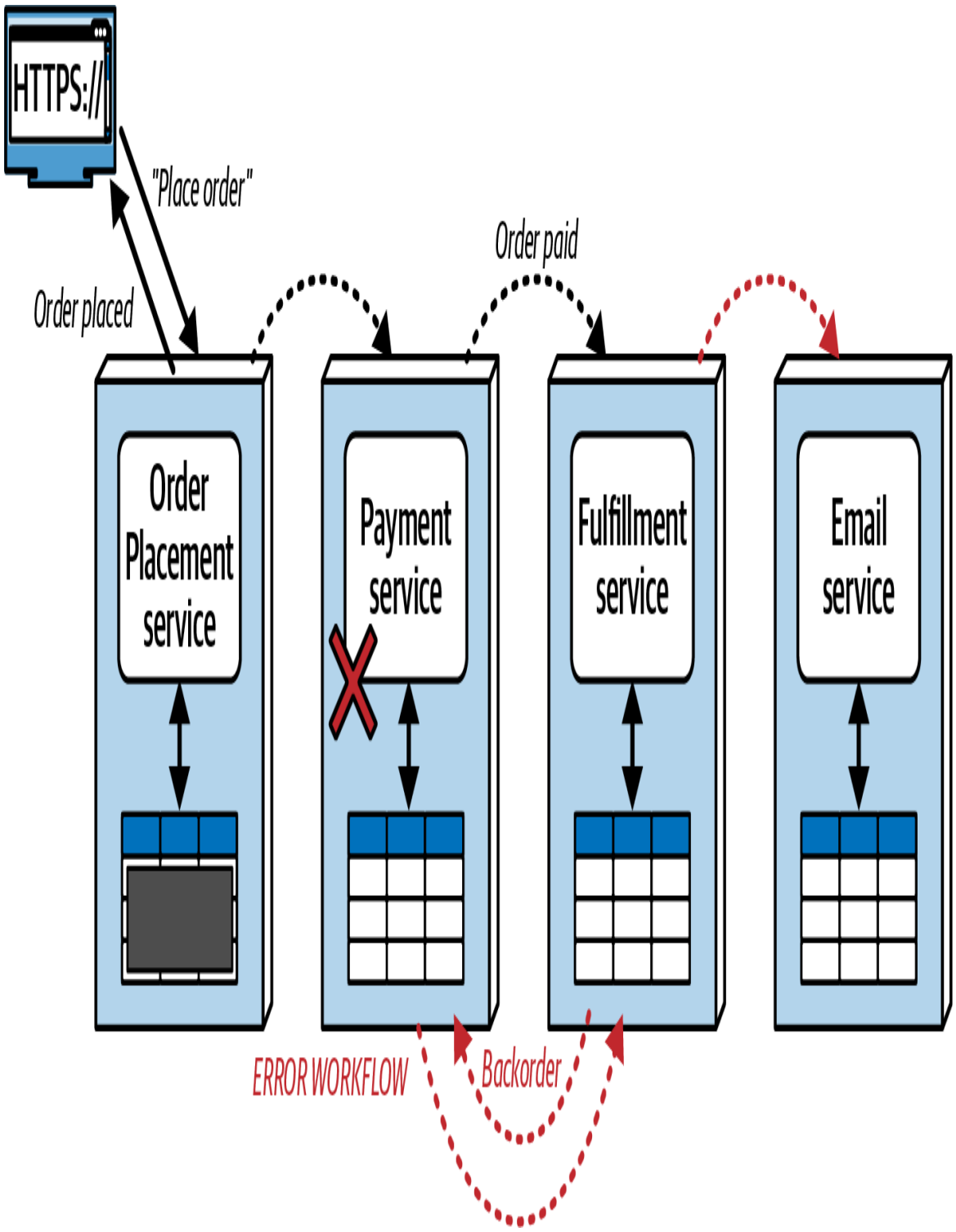


Figure 9-4. What happens in a choreographed workflow when cascading events occur?

A major issue with Choreography is that all the workflow logic is "smeared" across all the services. In Chapter 37, we discuss the difference between *semantic* and *implementation* coupling, which this pattern

highlights. The error and boundary conditions are part of the semantics of the workflow, and the code for that logic must reside somewhere. In Choreography, because we don't use a central coordinating service, it must reside with the domain services.

WHY NOT USE AN ERROR-HANDLER SERVICE?

Many architects ask us why they should spread the error-handling logic around the domain services, when the team could instead create an error-handler service and dispatch exceptions to it? That *is* a good solution. If we introduced such an error-handler service, what would we call it? An *orchestrator!*

Using an external service for a responsibility as narrow as error handling means the architect is no longer using the Choreography pattern. At best, they're using a hybrid. Thus, while it's true that mingling domain and error-handling logic is a headache for architects, that is a defining characteristic of the pattern itself.

One of the major issues with choreographed workflows is determining the state of a started, but not yet complete, workflow. For example, in the system pictured in Figure 9-2, there's no easy place to query to find out the state of a particular order. In the Orchestration pattern, the orchestrator tracks the state of the workflow, making it easy to discover. That type of centralized state is missing in Choreography, but the need arises enough to warrant its own pattern, the Front Controller (Chapter 19).

Implementing the Pattern

To implement Choreography, the architect must "wire together" event handlers that each platform will handle differently. Implementing Choreography in a set of event processors is illustrated in Example 9-1.

Example 9-1. Implementing Choreography with event processors

```
define event_processor domain1
define event_processor domain2
```

```
define event_processor domain3
define error_processor error
define system requester
message_queue = {SUCCESS, ERROR}

domain1.listen_for INITIATING_EVENT, output error to ERROR
domain2.listen_for domain1.output via domain1.SUCCESS, output error
to ERROR
domain3.listen_for domain2.output via domain2.SUCCESS, output error
to ERROR
error.listen_for domain1.ERROR, domain2.ERROR, domain3.ERROR
   send ERROR state to requester
end error

requester.listen_for ERROR
send requester.INITIATING_EVENT to domain1
```

In Example 9-1, we define the event processors along with two event channels, `SUCCESS` and `ERROR`. Each event processor listens for the output of the previous one and sends errors to the error queue. To kick things off, an initiating event invokes the first domain service by inserting an event.

Trade-Offs

Which coordination pattern to use (Orchestration or Choreography) is an important architectural choice: it's structural, it's hard to change later, and it offers significant trade-offs.

Pros

Choreography is quite attractive for its inherent simplicity — upon first glance, architects like the simplicity of the happy path. However, the real complexity lies in boundary and error conditions. The advantages of applying this pattern include:

Excellent support for operational capabilities

> Because it offers the fewest possible structural bottlenecks and encourages asynchronous communication (which itself adds

significantly to responsiveness), this workflow pattern is one of the best for maximizing capabilities like scalability, elasticity, and responsiveness.

Great for event-driven workflows

Many workflows are event-driven: an event occurs that leads to a reaction that leads to another event. For example, ordering a book would be easy to model as a series of events and event handlers: order placement, payment, shipping, and notification.

Cons

While it offers many advantages, this pattern also has some sharp edges. The disadvantages of applying this pattern include:

Error and boundary options are "smeared" across domain services

Because Choreography doesn't have a coordinating service like an orchestrator, any workflow logic required for error cases or other boundary conditions must reside in domain services.

This "smearing" effect has other negative side effects: if an architect wants to reuse a domain service, that service must contain the workflows and error handling for *all* workflows that utilize it. For example, if three different choreographed workflows use the `Payment` service, it must contain logic to react to workflow and error possibilities in each one.

Resolving workflows and asynchronous communication

Because it lacks holistic coordination, when an architect builds a workflow that must come together in the end, Choreography makes it more difficult to implement. For example, consider a workflow that assigns different discount rates for different types of credit cards and transactions. At the end of the workflow, all the numbers must add up correctly; when using Choreography and asynchronous events (the most common combination), architects must use synchronization techniques

(like semaphores, mutexes, or other techniques) to ensure consistency at the end.

Comparison Chart

When architects build workflows in distributed architecture, they can either coordinate them (some form of orchestration) or they can allow the services to coordinate amongst themselves (choreography), so the Without Pattern column with Choreography implies Orchestration, as reflected in the chart. Table 9-2 shows the default trade-offs for the Choreography pattern; its opposite is Orchestration.

Table 9-2. Comparison chart for Choreography (versus Orchestration)

Capability	Orchestration	Choreography
IMPROVED		
Adaptability	★★	★★★★★
Availability	★★	★★★★★
Domain Alignment	★★★★	★★★★★
Elasticity	★★	★★★★★
Evolvability	★★★★	★★★★★
Fault Tolerance	★★	★★★★★
Responsiveness	★★	★★★★★
Reusability	★★★	★★★★★
Scalability	★★	★★★★★
Throughput	★★	★★★★★
DEGRADED		
Data Consistency	★★★★★	★★★

Capability	Orchestration	Choreography
Data Integrity	★★★★★	★★★
Debuggability	★★★★★	★★★
Error Handling	★★★★★	★

In Table 9-2, we compare the Choreography pattern with workflows that don't use it (implying at least some level of orchestration, if not the full-blown Orchestration pattern, discussed in Chapter 27).

Improved

The following capabilities are improved when using Choreography:

Adaptability

> Adaptability — adding new behavior while preserving existing behaviors — is easier in a choreographed architecture because an architect can add new collaborators just by extending events and messages (see the Extensible Events pattern in Chapter 18) without also updating an orchestrator.

Availability

> Availability is generally improved when an architect can remove potential fault tolerance bottlenecks such as an orchestrator. While architects can build more resiliency in distributed architectures, single points of failure (like orchestrators) complicate the design.

Domain Alignment

> Domain alignment tends to be slightly better in Choreography as it allows architects to model workflows exactly like messages and events flow through the problem domain. However, the difference isn't much,

as teams can model domains with orchestrators as well, but the combination of Choreography and events allows a good isomorphic match between Choreography and DDD.

Elasticity

Elasticity relies on Mean Time To Start (MTTS) for each service. Because Choreography relies solely on domain services (and no orchestrator), it's easier to build elastic systems with Choreography: the workflow startup time is close to service startup time. In Orchestration, the orchestrator and domain services must start, which will tend to be slower because of increased dependencies.

Evolvability

Evolvability is slightly better when using this pattern because Choreography tends to use asynchronous messaging, allowing for easier workflow evolution. For example, if an architect wants to add some behavior to a choreographed workflow that uses events, they can add new event listeners without affecting any other part of the architecture.

However, from a behavior standpoint, Choreography tends to degrade evolvability slightly because it requires architects to build workflow logic into services, making it harder to evolve those services to different purposes later. For example, a `Payment` service in an orchestrated workflow can avoid all error, boundary, and other workflow logic because it can rely on the orchestrator to handle it. Conversely, in a choreographed workflow, services must at least be aware of both error states and the workflow context. Our previous `Payment` may be used by several different workflows, with different error outcomes for each, and thus requires unique workflow-specific code. However, in the context of the pattern, Choreography makes evolving the workflow slightly easier.

Fault Tolerance

Like availability, patterns without central bottlenecks tend to be more fault tolerant.

Responsiveness

Responsiveness is outstanding in choreographed workflows because they allow for so few potential bottlenecks. Services respond to each other as quickly as events occur, rather than waiting on message dispatches from an orchestrator.

Reusability

In this context, reusability is similar to evolvability where workflow reusability is improved but domain (behavioral) reusability is degraded. From a workflow standpoint, reusability is improved because of the flexibility of asynchronous events, the common communication technique within this pattern. From a behavioral standpoint, reusability suffers because of the requirement to add workflow awareness to choreographed services. For example, consider a `Payment` service that teams use in three different workflows. In choreographed workflows, the service must include error handling and boundary logic, making reusability more difficult. Conversely, when using orchestration, each domain service can be simpler, relying on the orchestrator for workflow concerns.

Scalability

This pattern is outstanding for *scalability* and *throughput* because each service spends as little time as possible waiting on coordination.

Throughput

Similar to scalability, this pattern is also outstanding for *throughput* because each service spends as little time as possible waiting on coordination.

Degraded

The capabilities that are degraded when applying this pattern include:

Data Consistency

Data consistency is harder for architects to achieve in Choreography because of the lack of central coordination.

Data Integrity

Similar to data consistency, data integrity is harder to ensure across multiple independent collaborators without Orchestration.

Debuggability

Debuggability is degraded in this architecture, particularly at scale, because it becomes difficult to understand complex interactions between services with no coordinator.

Error Handling

Error handling is difficult within Choreography because it lacks a designated error-handling component. Instead, error-handling logic is usually distributed across services, forcing each service to be more aware of the workflow context in which it operates. Further complications arise when architects use individual services (such as `Payment`) in more than one choreographed workflow, which requires that service to be aware of multiple error states and remediation strategies.

This architecture pattern trades off operational capabilities such as scale and responsiveness for workflow management and error handling. When architects need high scale or responsiveness and have relatively simple workflows, Choreography is an excellent choice.

Influences

Choreography is one of the key decisions involved in building distributed architectures, so it isn't surprising that it influences and is influenced by lots of factors. We cover three of the common influences that change the default trade-offs for this pattern: coordination, communication, and consistency. (For more detail, see Part II of *Software Architecture: The Hard Parts*, which deals with how these three forces affect building transactional workflows in a distributed architecture, with or without an orchestrator.)

Coordination

> *Coordination* refers to the type of workflow: either Orchestration or Choreography. Clearly, this pattern deals with Choreography as one of the two options.

Communication

> *Communication* refers to the architect's choice of communication protocol: either synchronous or asynchronous. As discussed in "Architecture Quantum", choosing synchronous communication often changes the extent of coupling in the system, which may in turn affect consistency.

Data consistency

> *Consistency* refers to the common choice in microservices to favor *eventual consistency* over *atomicity*. *Atomicity* implies the same level of transactionality offered by relational databases that feature ACID (Atomic, Consistent, Isolated, Durable) transactions (see Chapter 37 for a definition). For reasons beyond the scope of this book, distributed architectures can't have full ACID transactions. However, architects can "mimic" the *atomic* part of ACID by designing workflows that feature *compensating updates* (covered in Chapter 11).

Next, we need to analyze the impact of communication and consistency on the Choreography pattern's trade-offs.

Communication (Synchronous Versus Asynchronous)

Table 9-3 shows the trade-offs of choosing synchronous versus asynchronous communication.

Table 9-3. Choreography's trade-offs change based on the communication style used

Capability	Synchronous	Asynchronous
UNCHANGED		
Domain Alignment	★★★★★	★★★★★
IMPROVED		
Data Consistency	★★★★★	★★★
Data Integrity	★★★★★	★★★
Debuggability	★★★★★	★
Error Handling	★★★★★	★
DEGRADED		
Adaptability	★★★	★★★★★
Availability	★★★	★★★★
Elasticity	★★	★★★★★
Evolvability	★★★	★★★★★
Fault Tolerance	★★★	★★★★★

Capability	Synchronous	Asynchronous
Responsiveness	★★	★★★★★
Reusability	★★★	★★★★★
Scalability	★★	★★★★★
Throughput	★★	★★★★★

In Table 9-3, the trade-offs are virtually the same as in the contrasting Orchestration pattern. That makes sense, since Choreography is a form of *coordination*, which often influences trade-offs.

The following sections describe each of the capabilities found in Table 9-3.

Unchanged

The capabilities that remain unchanged with regard to this influence include:

Domain Alignment

Domain alignment is unaffected by the choice of communication.

Improved

The following capabilities improve when architects choose synchronous communication. Conversely, the charts also show the trade-offs of asynchronous communication by inverting the charts and focusing on its strong capabilities.

Data Consistency

Generally, data consistency is more difficult when a pattern removes more ways of coordinating consistency (such as an orchestrator or synchronous communication). Asynchronous communication adds

multithreading to consistency concerns, which adds additional complexity.

Data Integrity

Similar to data consistency, data integrity is harder to ensure with asynchronous communication.

Debuggability

Debuggability is further degraded in this architecture when teams use asynchronous communication — tracking asynchronous errors at scale presents many challenges. Note that in the default trade-off chart, debuggability was marked as degraded, yet here it's improved. When we discuss influences, it's within the context of the overall pattern. In other words, if an architect is using Choreography for specific reasons, then using synchronous communication will make debugging easier than asynchronous communication.

Error Handling

Error handling becomes even worse with asynchronous communication. Like with debugging, handling errors at scale with asynchronous communication is difficult because of timing and responsibility — who handles the error and when?

Degraded

The capabilities that degrade when architects choose synchronous communication include:

Adaptability

Adaptability becomes more difficult with synchronous communication because each domain service blocks waiting for results, harming the ability to add more behavior to the workflow. Conversely, using Choreography with asynchronous communication allows event subscription, pub/sub capabilities, and other decoupling techniques for

communication, making it easier for architects to adapt the existing behavior toward new behavior.

Availability

Availability is improved with asynchronous communication because it allows architects more flexibility in terms of circuit breakers, timeouts, and other asynchronous mechanisms. Synchronous calls tend to stress availability because if the service isn't available, the call will fail.

Elasticity

Elasticity is greatly enhanced by asynchronous communication, allowing services to respond with the fewest possible blocking calls.

Evolvability

Architects find evolvability, like adaptability, more difficult when using synchronous communication because each call blocks, preventing communication flexibility.

Fault Tolerance

Fault tolerance is degraded with synchronous communication because it's more difficult for architects to handle availability concerns with synchronous communication.

Responsiveness

Synchronous communication significantly impacts *responsiveness* in distributed architectures. Moreover, in Choreography, synchronous communication creates many "one-way streets," providing few options when things break. For example, consider the diagram in Figure 9-5, which uses synchronous communication. When the first service calls the second, it blocks and waits until the second service returns. The second service, in turn, makes a blocking call to a third service. Throughout this communication chain, the first service must wait for an

eventual return. If one of the services times out, the blocked service just has to wait. Because of the blocking nature of synchronous communication, most architects choose to use asynchronous communication within Choreography, using events rather than messages.

Responsiveness is one of the most common challenges in microservices. Because microservice architectures encourage architects to create fine-grained services with data fidelity, they often suffer from poor responsiveness because of data isolation. Choreography provides the fewest bottlenecks and coupling points; using the Choreography pattern with asynchronous communication is the best combination for operational capabilities (but it has lots of challenging trade-offs).

Reusability

Reusability is much better with asynchronous communication because it allows the architect to easily extend the workflow by adding new events/listeners.

Scalability

Scalability is much better with asynchronous communication because of the potential for parallel execution.

Throughput

Similar to scalability, asynchronous communication provides outstanding throughput for this pattern.

Atomicity (Data Consistency and Integrity)

Atomicity is more complex in Choreography than in Orchestration, since an orchestrator can help coordinate compensating updates and other implementation details. Choreographed workflows lack a central coordinator, so architects implement compensating updates.

In Figure 9-5, the first two domain services are successful, but the third is not. It must issue events or messages to the other services telling them to issue compensating updates before returning the error state to the user.

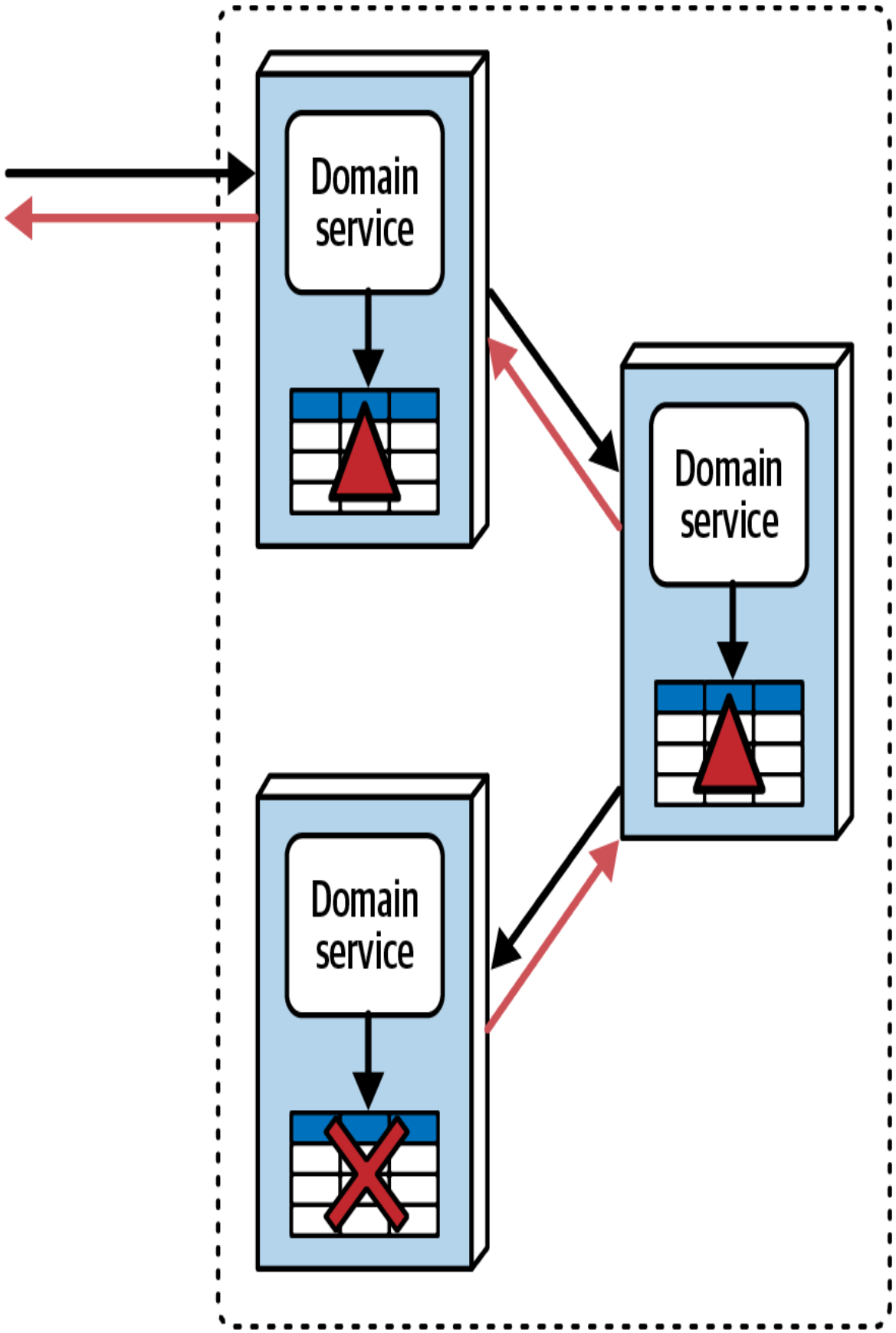

Domain service
Domain service
Domain service

Figure 9-5. Compensating updates in a synchronous choreographed workflow

Architects who design workflows that include atomicity discover that the trade-offs shift significantly when using the Choreography pattern, as shown in Table 9-4.

Table 9-4. The trade-off shifts due to the attempt at better data consistency through atomicity

Capability	Atomic	Eventually Consistent
UNCHANGED		
Domain Alignment	★★★★★	★★★★★
IMPROVED		
Data Consistency	★★★★★	★★
Data Integrity	★★★★★	★★
Debuggability	★★★	★★
DEGRADED		
Adaptability	★	★★★★★
Availability	★★	★★★★★
Elasticity	★★	★★★★★
Error Handling	★	★★★
Evolvability	★	★★★★★
Fault Tolerance	★★	★★★★★

Capability	Atomic	Eventually Consistent
Responsiveness	★★	★★★★★
Reusability	★★	★★★★★
Scalability	★★	★★★★★
Throughput	★★	★★★★

The following breakdown shows the impact of atomicity on Choreography.

Unchanged

The capabilities that remain unchanged with regard to this influence include:

Domain Alignment

> Domain alignment is largely unaffected by the data consistency strategy the architect chooses.

Improved

The capabilities that improve when architects focus on atomicity include:

Data Consistency

> Because the architect chose Choreography plus atomicity, data consistency is fantastic.

Data Integrity

> Data integrity is also highly improved in this architecture, although it still requires a bit more coordinating to verify both consistency and integrity.

Debuggability

Debuggability becomes slightly easier when architects use Choreography with atomic because the architect is debugging a known data state with a known end condition: atomicity. Eventual consistency, on the other hand, has a time element: *when* does it become consistent, adding a temporal element to debugging that isn't present for atomic workflows.

Degraded

The number and severity of the capabilities that degrade when architects focus on atomicity illustrate the data behind the common conventional wisdom in distributed architectures to avoid atomicity whenever possible. These capabilities include:

Adaptability

Adaptability is highly degraded when architects introduce atomicity to Choreography: domain services that must coordinate for data consistency must include a lot of code that hinders adaptability.

Availability

Availability is also lessened when architects introduce atomicity because now domain services must rely on more workflow infrastructure to be available for error handling, boundary conditions, and so on.

Elasticity

Elasticity is degraded because architects must build more coordination logic into domain services.

Error Handling

Error handling is degraded because adding atomicity to Choreography adds a notoriously difficult error-handling scenario to the workflow:

what happens if the compensating update fails? This is a common outcome of the Compensating Update pattern (covered in Chapter 11).

Evolvability

It's harder to evolve domain services that have workflow logic around data consistency inside them.

Fault Tolerance

The scope of fault tolerance goes up when architects add atomicity to Choreography because of the required coordination.

Responsiveness

Responsiveness takes a big hit when architects choose to make workflows atomic. The lack of an orchestrator means that transactional logic must reside within the domain services, requiring more logic and coordination when errors occur.

Reusability

Reusability is also degraded because adding coordination logic to domain services makes them less reusable.

Scalability

Scalability is degraded when combining atomicity and Choreography because atomicity requires coordination. This adds extra communication overhead to the workflow.

Throughput

Throughput is also degraded when combining atomicity and Choreography because atomicity requires coordination.

Many architects find Choreography appealing because of its decoupled nature, but also need high data consistency, leading to this common combination. And, in fact, all of the poor scores around data from the

original trade-off charts are greatly improved…at the expense of virtually everything else! Most of the beneficial trade-offs of Choreography go away when teams add something as coupling as atomicity.

In microservices, architects try to avoid atomicity within workflows as much as possible; the negative consequences illustrated within this chapter partially show why. Choreography makes atomicity in particular more difficult, because atomicity requires coordination.

Architecture Quantum

As discussed in "Architecture Quantum", architects must consider five different aspects of the coupling and architectural scope as it relates to communication patterns like Choreography and Orchestration: *static*, *dynamic*, *temporal*, *contract*, and *atomic*:

Static coupling

> *Static coupling* refers to the dependencies and "plumbing" of the architecture. Workflows using Choreography have the least amount of static coupling at the service level — if the services are well defined, they should stand alone operationally without external coupling, which leads to low static coupling.

Dynamic coupling

> *Dynamic coupling* is required when services communicate. In Choreography, each service communicates directly using either events or messages (see "Events Versus Messages"), reducing communication coupling to the smallest possible degree. When combined with asynchronous communication, this workflow style creates the fewest possible bottlenecks and coupling points.

Temporal coupling

> *Temporal coupling* doesn't really apply in the Choreography pattern unless one or more of the services uses a cache.

Contract coupling

In the absence of an orchestrator, the workflow state must reside somewhere (we discuss one strategy in the Front Controller pattern; see Chapter 19). Some architects include the extra needed state within the contract that is passed between services. Managing state within a workflow requires cross-service information, and the contract may include extra fields to hold that information. The downside is that this approach can gradually lead to the Stamp Coupling antipattern, covered in Chapter 31.

Transactional coupling

We discussed this specific influence in "Atomicity (Data Consistency and Integrity)".

Atomic coupling represents data consistency and ACID characteristics across service boundaries. When workflows require atomicity across domain services without an orchestrator, compensating updates must be used, which significantly increases atomic coupling across the services.

None of the above factors are inherently good or bad, but they do all require architects to think about important aspects of the architecture. In particular, combining Choreography with asynchronous communication creates a highly scalable architecture, but the lack of coupling makes coordination more difficult — a classic software architecture trade-off.

How to Govern This Pattern

The possibilities for governing the Choreography pattern revolve around communication and cumulative responsibility.

Governing Communication

Architects use Choreography in distributed architectures, which means it's subject to all of L. Peter Deutsch's Fallacies of Distributed Computing, including availability, bandwidth, and so on. Thus, architects must often govern the availability of services within a choreographed workflow. For example, in an event-driven workflow, how can the architect determine if one of the services is down when the event occurs? Tools like durable message queues can ensure delivery, but a host of problems can occur in choreographed communication. It's more complex than orchestrated communication because an orchestrator can look after completeness and other workflow factors. These are more difficult in Choreography.

If this is a concern, the architect can build a reactive verification mechanism for each workflow that periodically reads the log files of coordinating services and checks that the workflows are complete. For example, in a book-ordering workflow, the architect could flag entries that were `ORDERED` and `PAID` but never `SHIPPED`, which is the usual terminus of this workflow.

Cumulative Responsibility

Most solutions start small and then grow incrementally over time. For example, a simple order-processing workflow may grow validation rules, internationalization, esoteric shipping options, and a host of unpredictable behaviors. Managing domain growth within architectures is a complex issue in itself (covered in the book *Building Evolutionary Architectures*), but architects should be on the lookout for choreographed services that grow to encompass too much responsibility.

Determining whether a system has the right level of service granularity is complex. Signs that a service has accumulated too much responsibility over time include:

Having one godlike event processor

> It's easy to fall into the trap of continuing to add responsibility to a single service or event processor, rather than refactoring to break it up when it becomes too large. For example, a `Shipping` service might

grow to add international shipping, along with immense complexity. Even if it originally made sense for all shipping-related behavior to be in a single service, it might be beneficial to break such a service up as it evolves.

One service becoming a message/event hub

A choreographed service that becomes a message hub is another warning sign. For example, if most of the services in a workflow call (or trigger) one or more services, the architect should ensure that they haven't stumbled into Orchestration. As an illustration of this, it's perfectly fine for all services to emit logging events that are captured by an observability service, but routing *all* events through a domain service probably indicates a design flaw (and a form of Orchestration).

Related Patterns, Antipatterns, and Pitfalls

These patterns are related to Choreography:

Orchestration pattern (see Chapter 27)

The other primary communication pattern for distributed architectures.

Front Controller pattern (see Chapter 19)

The Front Controller pattern provides a mechanism to add some workflow state back into Choreography by adding it to the first-called service.

Cross-Cutting State pattern (see Chapter 13)

Often, architects who implement Choreography find that they need to store external state beyond just the domain services (such as boundary conditions or errors), and the Cross-Cutting State pattern is a way to implement that behavior.

Chapter 10. Command Query Responsibility Segregation (CQRS) Pattern

In most traditional architectures, particularly those performing CRUD-like operations, a single data model is usually used for both read and update (create, read, update, delete) operations. This usually works well, up to a certain point. Then architects have to decide whether to optimize the system for reads or for writes. This is where the Command Query Responsibility Segregation (CQRS) pattern can help. It separates the concerns of reads versus writes, though at the cost of introducing significant complexity.

Table 10-1 summarizes the metadata for the CQRS pattern.

Table 10-1. Metadata for the Command Query Responsibility Segregation pattern

CATEGORY	METADATA
CONTEXT:	Data, Event-Driven Architecture, Message-Based Architectures, Microservices, Monolithic Architectures
IMPROVED:	Availability, Elasticity, Fault Tolerance, Flexibility, Responsiveness, Scalability, Throughput
DEGRADED:	Cost, Data Consistency, Data Integrity, Maintainability, Simplicity, Testability

How Can I Optimize My Data for Reads and Writes?

Consider Figure 10-1, which illustrates a typical CRUD application with a user interface (UI) backed by a service, using a data source. The application performs both reads and writes, utilizing the same data model for both.

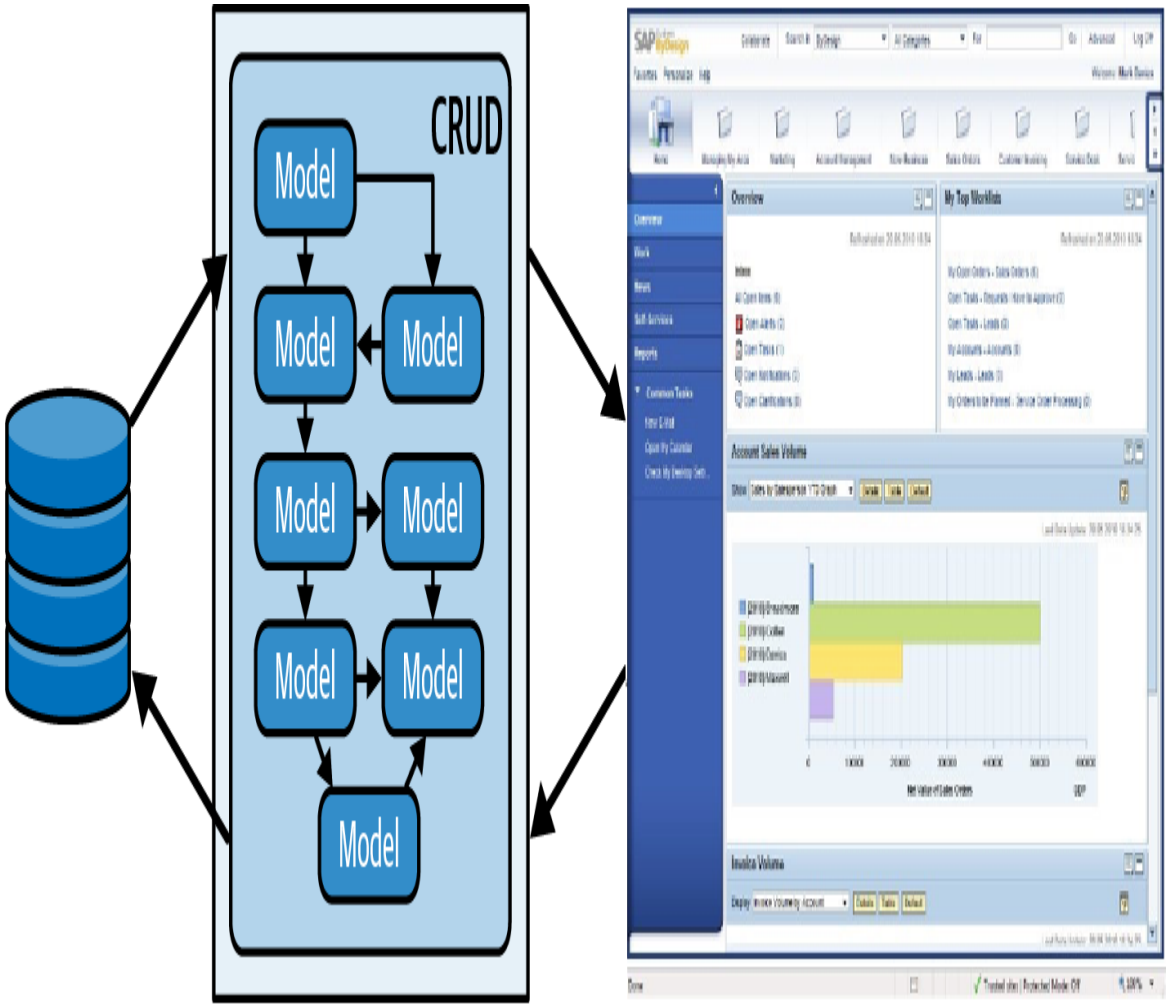


Figure 10-1. Typical CRUD architecture

As the system starts to scale, architects are confronted with a multitude of challenges. To improve responsiveness on reads, they might choose to introduce indexes in the datastore, making every read operation more performant. However, the trade-off is that writes now start to take longer and longer, because each insert, update, or delete must now update those indexes. Conversely, if they optimize the system for writes by removing

indexes, data lookup becomes more expensive, since the system can't sufficiently leverage the datastore of performance-enhancing primitives (like indexes for relational stores).

Alternatively, users might need to *view* the information differently in different scenarios. This would require multiple representations of the data, and consequently of the data model. But the write side has no need for this — not to mention that it introduces an impedance mismatch between the two sides of a data model designed to be shared between two kinds of operations.

In terms of security, due to compliance or regulatory controls, reads may be subject to more authorization/authentication controls than writes, while writes require validation. Imposing these constraints on a data model designed to serve both complicates the system's workflows and design.

All in all, architects have to balance the trade-offs between reads and writes across various dimensions, reducing both sides of the equation to the lowest common denominator. This is where CQRS can provide great assistance.

Applying the Pattern

CRUD operations can be categorized into two distinct and separate types: *commands* that affect the state of the application, and *queries* that retrieve data from the system without affecting any state or causing any side effects. CQRS starts by separating the data model into a command side and a query side, as depicted in Figure 10-2.

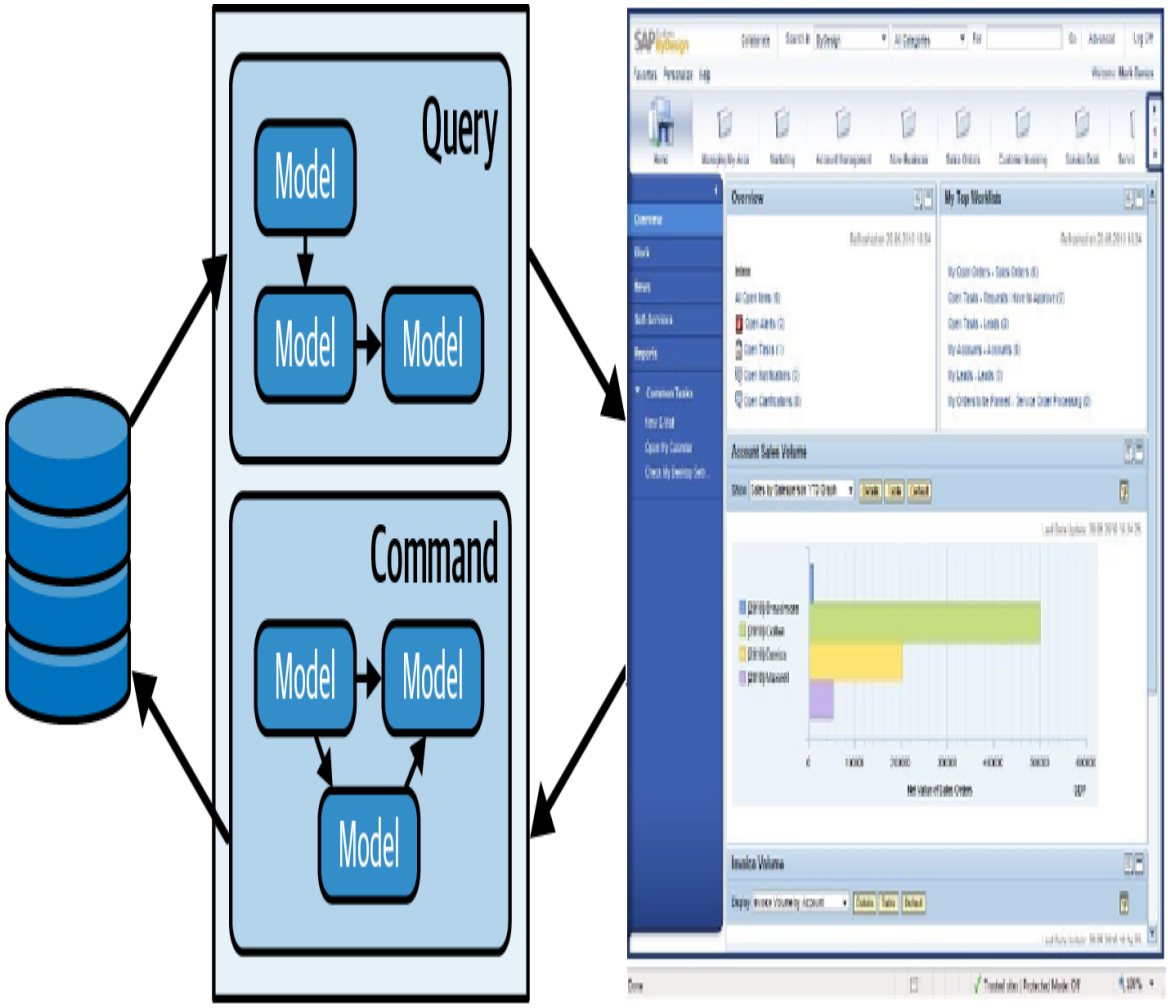


Figure 10-2. Segregating command and query

Figure 10-2 represents the first step in adopting CQRS. The command side of the model, which is responsible for create, update, and delete operations, is typically an entity-based model. It works well for enforcing business rules, validating state transitions, and imposing security constraints. The command side might also focus heavily on optimizing around workflows or user intent, as opposed to generic create, update, and delete operations.

The query side simply offers presentation-style objects like data transfer objects (DTOs) and data projections, and is highly optimized for reading data. This step helps with building specific data models that align with the kinds of operations they support.

This kind of segregation will be familiar to many teams. However, notice that in Figure 10-2 both the command and query sides are still using the

same datastore. Consequently, while the code is sufficiently demarcated, the shared datastore is still the optimization bottleneck. It can be optimized for either reads or writes, but not for both.

The second stage of adopting CQRS is to split the datastore into a *read datastore* and a *write datastore*. This opens the door to a whole slew of optimizations that focus on supporting specific kinds of operations. For example, the command and query sides might use completely different datastore technologies (NoSQL for writes and a Graph database for reads). The write side will most likely eschew any and all indexes, but will be highly normalized to ensure data integrity, avoid duplication, and better update performance. On the query side, a wide range of indexes can be installed based on query patterns. Data can reside in denormalized fashion, reducing the need to aggregate data at the application layer. Teams can then build materialized views and projections on top of the data, precomputing results and offloading the work at read time.

Of course, separating reads from writes will require the architect to design a mechanism to synchronize data from the write datastore to the read datastore, as depicted in Figure 10-3.

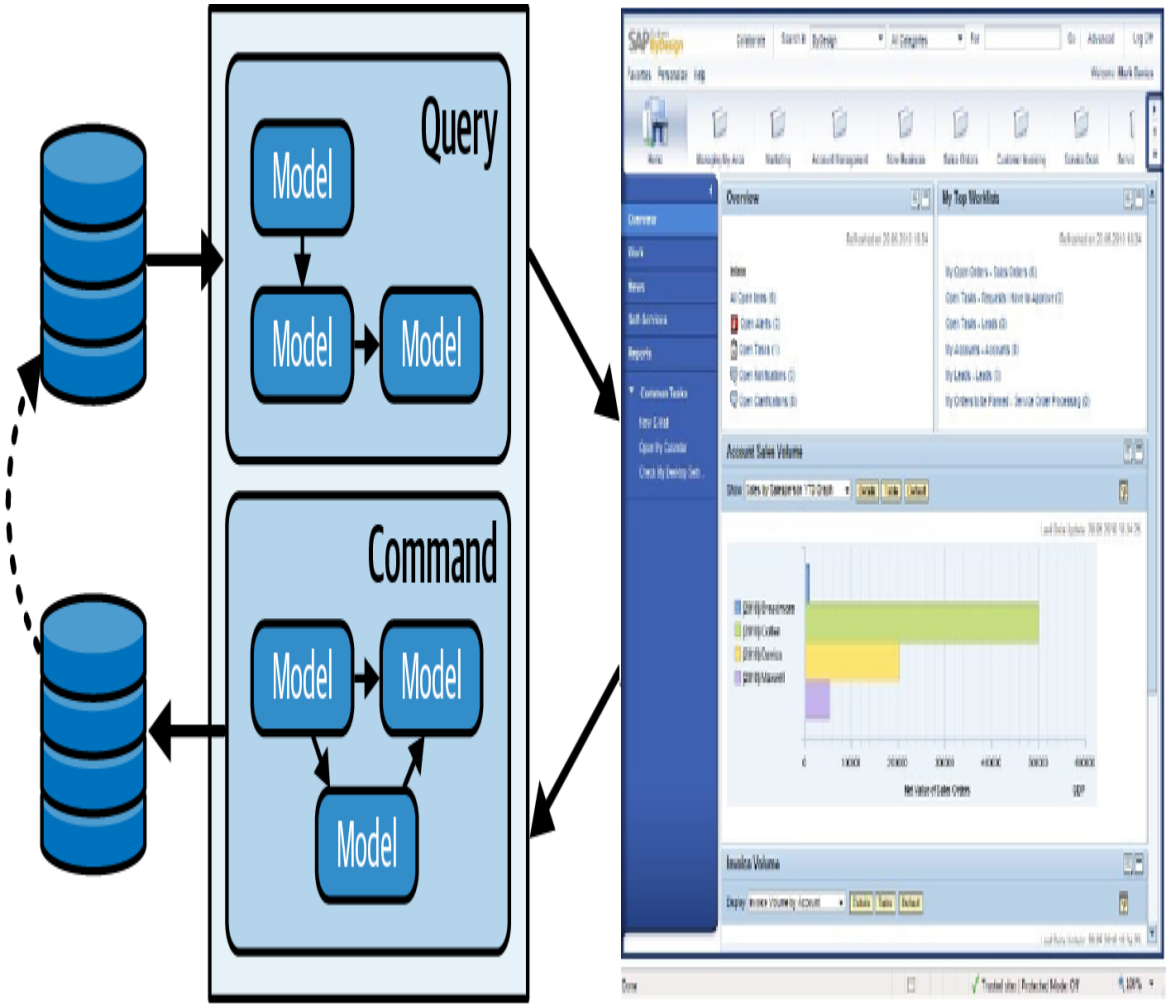


Figure 10-3. Segregating datastores between command and query

Implementing the Pattern

CQRS can be implemented in many different ways. Each one affects the topology of the architecture and brings with it distinct benefits and trade-offs.

Single Service for Both Command and Query

In the CQRS topology depicted in Figure 10-3, both command and query are single-deployment units. This might work well in some cases, but invariably, teams that adopt CQRS do so because their systems have very different read versus write needs. If those needs are operational capabilities, as opposed to domain-level concerns (like in systems with extremely

complex domain logic) or perhaps security needs, then separating command and query into distinct services might be a better fit. For example, if the read workloads far outnumber the write workloads (as often is the case), it might make sense to separate the command and query portions of the monolith into distinct services.

Separate Services for Command and Query

Separating command and query into distinct deployment units brings a slew of benefits that aren't obvious right off the bat. Consider Figure 10-4.

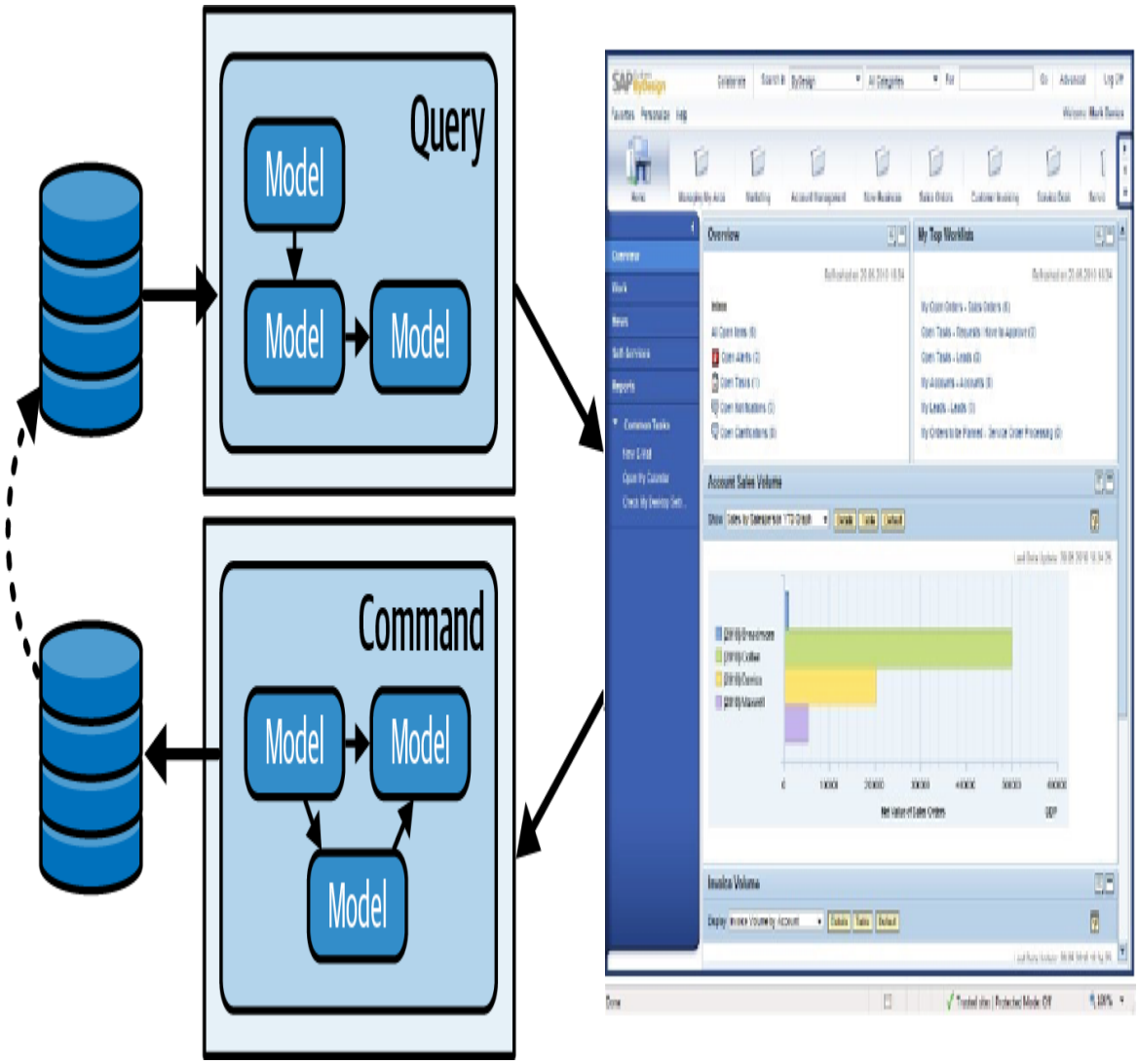


Figure 10-4. Segregating command and query into distinct deployment units

Take a moment to consider Figure 10-4. It's essentially two microservices! That means this particular setup reaps all the benefits of microservices, including structural capabilities (like using runtimes best suited to the problem at hand), process capabilities (like separate release cadences and easier testability and deployability), and operational capabilities (like individualized scaling). However, the two datastores need to remain in sync. And microservices come with their own significant trade-offs, which architects *must* account for when considering this topology for CQRS.

A note on eventual consistency

The full benefits of this pattern only manifest when *both* steps of the implementation are complete: separating the application logic *and* the command store from the datastore. The latter needs further consideration, because it means that the two stores need to be synchronized. This makes the entire system eventually consistent, with the query side lagging behind the command side (see Figure 10-4). That's important, because keeping the query side in sync with the command side can be challenging, and each mechanism that can be used to synchronize the two has specific trade-offs:

Asynchronous event-based synchronization

> Here, the write side publishes events to a broker, and the read service subscribes to those events to update its own read model. This has the benefit of completely decoupling the write side from the read side, while retaining the bounded context boundaries. The downsides include the operational overhead of introducing middleware, as well as handling failures. What happens if the write-side update succeeds but publishing the event fails, if updating the read side fails, or if events appear out of order? Teams must think all of these failure cases through and implement appropriate logic to address them.

Transactional Outbox pattern

> Another approach available to architects is to reach for yet another pattern: the Transactional Outbox pattern. In this pattern, the write side updates its own business tables and, *in the same transaction*, writes an

event message into the "outbox" table. A background process (owned by the write service) then picks up the event message and publishes it to a broker. The read service subscribes to these events from the broker and updates its own read model, maintaining proper bounded context boundaries. This technique avoids the dual-write problem, where business tables might get updated but the event publishing fails. It also guarantees that events are naturally ordered by a transaction timeline. One downside here is that the outbox table can grow aggressively, requiring architects to think through archiving and/or purging strategies. Another is that running a continuous polling job adds constant CPU and I/O load on the write database.

Change data capture (CDC)

For change data capture, architects can reach for modern CDC platform tools like Debezium, which monitors transaction logs and streams them as events to a broker. The read service subscribes to these events and updates its own read model. This relieves application teams of having to write synchronization logic. However, *transactional* logs are often too low-level from a *business* perspective; updating a value or a set of values in a row or a column is rarely sufficient to explain the "why." Fortunately, tools like Debezium can be configured to publish semantically meaningful messages through transformations and message enrichment, as opposed to publishing raw transaction logs. The trade-off is the operational overhead of introducing an additional tool to aid synchronization.

In-memory replicated caching

Another synchronization approach uses in-memory replicated caching. In this pattern, the write service maintains an in-memory cache containing the same data as its database. The read service reads from a replica of this in-memory cache, providing extremely low-latency access to synchronized data. Once the read service updates its read database, it removes the entry from the cache, which then propagates the removal to the write service's cache, thus preventing the cache from

growing unnecessarily. This technique provides very fast synchronization with minimal lag, though it requires additional memory resources and introduces the operational complexity of maintaining a distributed cache.

Various other techniques, like database triggers, have their own trade-offs. The takeaway here is that simply separating the writes from the reads introduces the need for synchronization, increasing the complexity that this pattern brings to the table.

The read-your-own-writes challenge

One of the most immediate user-facing consequences of eventual consistency in CQRS is the *read-your-own-writes problem*. In a typical application, a customer might place an order (a command in CQRS) and see the system reflect that order immediately on their order history page. However, in CQRS, if the query side has not received the synchronization update, the user won't see that order, which can be confusing. This isn't a bug: it's the fundamental nature of eventual consistency within CQRS. However, it can be jarring to users who expect immediate read-after-write consistency, so managing their expectations requires careful UX design. Architects should design the UX with this latency in mind and think through mitigation strategies like optimistic UI updates, or introduce polling or refresh mechanisms (websockets, server-sent events).

Trade-Offs

This section examines the trade-offs of the CQRS pattern.

Pros

Benefits of this pattern include:

Independent scaling

The primary purpose of this pattern is arguably allowing systems with very disparate workloads to scale parts of the system differently.

Depending on the system's workloads and read and write patterns, architects can not only optimize their datastore utilization, but (when fronted by independent services) also get many of the operational benefits of distributed architectures, like microservices.

Flexibility

If the implementation results in separating both the read and write application logic *and* the datastore, the result is effectively building out two microservices (with a synchronization mechanism). This brings a high degree of agility, since it allows architects to utilize different technologies for the application and the datastore. It also enables different team cadences and operational characteristics.

Cons

Downsides of this pattern include:

Complexity

Complexity is this pattern's Achilles' heel. CQRS is *hugely* complex, so teams should only adopt it when they absolutely *need* its capabilities. To start with, the query side will always lag behind the command side, introducing eventual consistency not between services, but within the system itself! This creates the read-your-own-writes challenge discussed earlier, where users may not immediately see the results of their own actions. This is often surprising to engineers who aren't used to this pattern. Using it may require significant training on the "duality" of the model, stricter architectural governance, and careful UX design to manage user expectations. Using dual datastores with a synchronization mechanism makes schema evolution harder (on either side) and adds operational overhead and complexity. Testing and debugging pose challenges as well because of the inherent timing issue between the write and read sides. Finally, this pattern has a lot of moving parts:

multiple services, datastores, and synchronization mechanisms, all to support what are essentially two sides of the same coin (command and query).

All this means that architects must be *very* diligent in assessing this pattern's trade-offs before adopting it.

Comparison Chart

Table 10-2 summarizes the trade-offs associated with this pattern. It compares using a single service backed by a singular datastore (the Without Pattern column) with using CQRS (the With Pattern column).

Table 10-2. Comparison chart for the CQRS pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Availability	★	★★★★
Elasticity	★★	★★★★
Fault Tolerance	★★	★★★★★
Flexibility	★★	★★★★★
Responsiveness	★	★★★★★
Scalability	★★	★★★★★
Throughput	★	★★★★★
DEGRADED		
Cost	$	$$$$$
Data Consistency	★★★★★	★★★
Data Integrity	★★★★★	★★
Maintainability	★★★★	★★
Simplicity	★★★★★	★

Capability	Without Pattern	With Pattern
Testability	★★★★	★★

The following sections describe each of the capabilities found in Table 10-2.

Improved

The capabilities that are improved when applying this pattern include:

Availability

> Availability is the most obvious upside to this pattern. Separating the command from the query side, particularly with an asynchronous pump to synchronize data, means that even if one side is temporarily unavailable, the other side can continue to function. By contrast, in a monolithic system using a singular datastore, the whole system is either available or unavailable. Availability does take a bit of a hit, however, because an unsynchronized write (due to either the command side or the synchronization mechanism failing) will leave the query side available, but stale.

Elasticity

> Decoupling the command and query sides allows each to scale independently, based on demand. This is particularly useful on the query side, since the datastore can be heavily optimized to allow newly created instances or replicas to boot up and start serving traffic quickly. On the write side, elasticity can be a challenge. A sudden spike in writes introduces an additional burden on the write datastore as well as the synchronization mechanism, and *may* introduce a larger lag between writes and reads until the spike subsides.

Fault Tolerance

Fault tolerance is about the degree to which, when fatal errors occur in one part of the system, other parts continue to function. Fault tolerance goes hand in hand with availability but does a little better (see Table 10-2), because this pattern separates the command and query side, allowing (within some tolerance) each side to continue functioning even if the other side is unavailable.

Flexibility

Separating command behaviors from query behaviors, and separating their respective datastores, introduces great flexibility in implementation. Teams can choose the most appropriate technologies for each side's runtimes and datastores.

Responsiveness

Responsiveness, along with throughput (see Table 10-2), is the key reason for adopting this pattern: the command and query datastores can be explicitly designed to match their separate needs. The command side, which doesn't really care how the data is read, might only need indexes to ensure data integrity. The datastore no longer needs to be optimized for reads, since that concern is explicitly decoupled by this pattern.

Similarly, the query side can be optimized for reading, perhaps even using a completely different implementation (like a Graph database) that's highly attuned to that purpose.

Scalability

Much like microservices' operational characteristics,[1] the read and write sides can scale independently based on their respective demands, allowing systems to optimize resource allocation and respond efficiently to varying workloads.

Throughput

Traditional setups on reads or writes can lock up rows in the datastore, severely affecting the *entire* system's throughput. Separating the two means that, even during a heavy-batch write process or a sudden spike on the read side, the overall architecture will remain mostly unaffected.

Furthermore, each datastore can be tuned to its individual workload patterns to aid throughput.

Degraded

The capabilities that are degraded when applying this pattern include:

Cost

Costs can easily balloon when teams adopt this pattern. This starts with the initial design and analysis required to separate the command from the query side. As development continues, the two sides must continuously coordinate to ensure that any changes to one side are accommodated on the other side, particularly if the changes touch the data model. If the two teams working on the command and query side happen to work at different cadences, this introduces the need to orchestrate deployments and synchronize releases.

Data Consistency

Consistency issues are the obvious detrimental side effect of eventual consistency. With the query side lagging behind the command side, and the system relying on a synchronization mechanism (which can fail), it's always possible for the two sides to drift out of consistency.

Data Integrity

Synchronization failures can introduce data integrity problems. For example, a nonidempotent update could end up processing the same update twice, corrupting the read model, or a partial update might leave data in an invalid state. Detecting this may require additional tooling in

place to detect, and correcting it may take compensating logic or manual intervention.

Maintainability

Separating the command and query sides implies a larger surface area of code, as well as schemas that must remain aligned. Any change affects multiple places: two services, two datastores, *and* the synchronization mechanism.

If separate teams maintain the two sides, that introduces yet another moving part to the architecture, requiring more coordination and increasing the chances of things falling through the cracks.

Simplicity

Right off the bat, separating the command and query sides of things with multiple datastores in play means that eventual consistency is the name of the game. In this case, the query side of the architecture will always lag behind the command side, which architects must carefully consider from the UI and UX perspectives, in addition to keeping the two datastores in sync.

Speaking of data synchronization, any technique an architect uses to update the query side with updates from the command side will have trade-offs. Evaluating them requires analyzing how the system is used and its read and write patterns, and ensuring that the system behaves the way it's supposed to (fulfilling domain requirements) and has the correct capabilities (fulfilling architectural needs). Data consistency and integrity concerns always arise due to the need for (and, possibly, failures of) synchronization, further complicating things.

As the system evolves, any schema changes require coordination between both sides, which becomes particularly arduous with different teams working on the command and query sides. Furthermore, architects must carefully devise migration strategies for dual datastores, which is even more difficult if the two stores have different schemas.

Testability

> Eventual consistency always makes testing harder, since the query side lags behind the command side. Integration tests must be carefully designed to test either the command side or the query side to avoid any timing or synchronization issues. However, any test that writes on the command side and reads on the query side must account for the lag. End-to-end tests are quite challenging, since the test *must* cater for eventual consistency. The need for mechanisms, like polling or retry, makes tests slower and potentially flakier.

Influences

This pattern can influence team topologies when different teams are built around the command and query sides.

Team Topologies

Separating the command side from the query side can encourage teams to align around those boundaries, with one team owning write-side behavior, business rules, and data integrity, while another owns read-side projections, query performance, and presentation-oriented data shapes. This can improve the architecture's overall agility by allowing each team to leverage the most appropriate runtime and datastore for its particular domain and separate out its release cadences. This separation enables each team to operate more independently within their bounded context, though it does require coordination around the data synchronization mechanism, which becomes the primary coupling point between the teams. Updating either side will require careful consideration of its impact on the other side.

How to Govern This Pattern

The single biggest concern with this pattern's implementation is the eventual consistency introduced by the synchronization mechanism

between the command and query side. It can introduce data consistency and data integrity concerns, and must be carefully governed. Teams can use a basic script, often running in a batch process, to routinely sample and compare the command and query records, alerting the team if it detects any drift.

Another concern, particularly if the same team maintains both sides, is governing against the possibility of the command side updating the query side directly, bypassing the synchronization mechanism. While this may not be immediately obvious, it subtly introduces a coupling point that undermines the greatest asset this pattern brings to the table: decoupling each side's operational characteristics! This check can be done using static analysis tools, such as ArchUnit in Java, to check if the command side references any domain or aggregate objects on the query side.

The synchronization mechanism between the command and query side is *the* decoupling point, affording a buffer between spikes on the write side and overloading the read side. However, if this fails to keep the two in sync within acceptable limits of lag, that's a signal that the mechanism needs to be reviewed or even rearchitected. This can be monitored using a simple performance test that magnifies the write load by 10x or even 100x, then measures how long it takes for the read side to catch up, and whether that lag is within the specified tolerances. Architects can even design fitness functions to draw metrics around lag time and failed sync count, calibrating them against an SLA as a further guardrail around synchronization failures.

Related Patterns, Antipatterns, and Pitfalls

The following is a related pattern for CQRS:

Event Sourcing pattern (see Chapter 16)

> Event Sourcing and CQRS often go hand in hand. The Event Sourcing pattern attempts to capture business-level updates to the system as an event log, which can then be replayed to build different query-side projections optimized for specific read use cases. In that sense, Event

Sourcing provides a natural synchronization mechanism between the command side and the query side of a CQRS implementation, in which the event store acts as the write-side persistence, and the events themselves act as the synchronization mechanism between the command and query sides.

---

1 See *Fundamentals of Software Architecture*, 2nd edition.

Chapter 11. Compensating Updates Pattern

The Compensating Updates pattern allows architects in distributed architectures to mimic the "atomic" feature of ACID transactions, implementing pseudotransactional behavior for multistage workflows.

The comparison details for this pattern appear in Table 11-1.

Table 11-1. Metadata for the Compensating Updates pattern

CATEGORY	METADATA
ALSO KNOWN AS:	Saga
CONTEXT:	Event-Driven Architecture, Message-Based Architectures, Microservices, Workflow
IMPROVED:	Data Consistency, Data Integrity
DEGRADED:	Elasticity, Responsiveness, Scalability, Simplicity, Throughput

Compensating Updates is a pseudotransactional pattern for workflows in distributed architectures. It's necessary because traditional database transactions won't work with distributed databases, so architects must implement the update behavior directly.

How Do I Roll Back a Distributed Transaction?

It's relatively easy to implement transactional behavior in monolithic architectures because of the relational database's ACID transactional capabilities. However, architects can't achieve the goals of ACID (Atomic, Consistent, Isolated, Durable) across databases.

As we cover in "ACID Versus BASE", architects can't rely on the automatic transactional mechanisms in relational databases when building microservices and similar architectures, and have to build the desired behavior themselves. One common pattern that appears in many microservices is *compensating updates*, where the architect builds their own safeguards for data consistency.

Applying the Pattern

Architects can implement this pattern in multiple ways, depending on the other decisions driven by the architectural concerns. However, architects must also be on the lookout for a subtle, sometimes ignored error condition: what happens when the compensating update fails?

What Happens When Everything Works Correctly?

Consider the orchestrated workflow (see Chapter 27) shown in Figure 11-1. A request comes into the orchestrator, which then dispatches it to each of the domain services in turn. Once the domain services have successfully updated, the orchestrator returns success for the workflow.

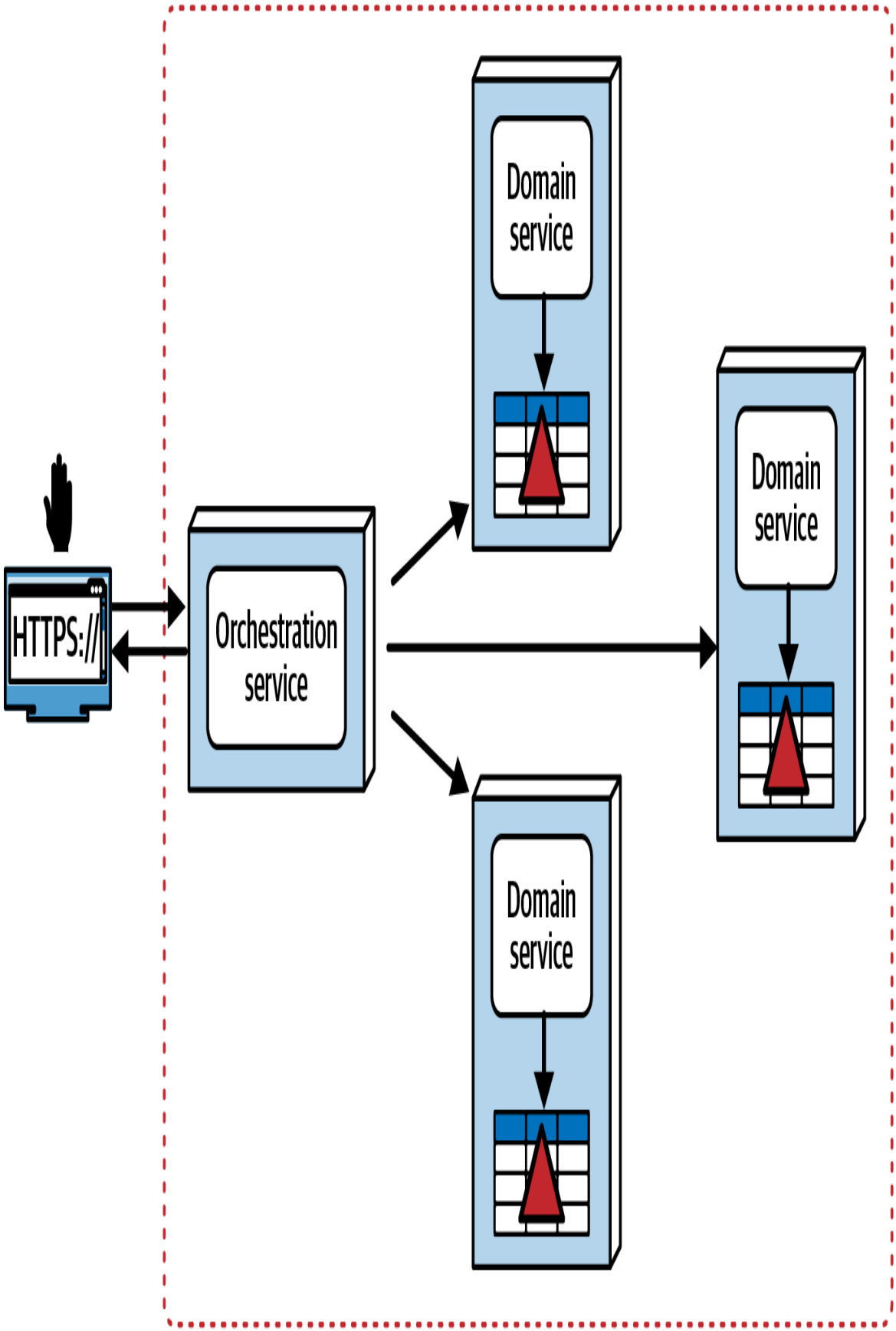

Domain service
Domain service
HTTPS://
Orchestration service
Domain service

Figure 11-1. A simple order workflow with multiple steps, implemented in microservices

In Figure 11-1, the orchestrator calls each of the services in order (either synchronously or asynchronously) and gathers the returns, then dispatches the result to the caller.

But what happens if one or more of the domain services can't update? In Figure 11-2, the bottommost domain service can't perform the update the orchestrator has requested, so it returns an error status to the orchestrator. The orchestrator, in turn, issues commands to the other domain services to "undo" whatever they did before.

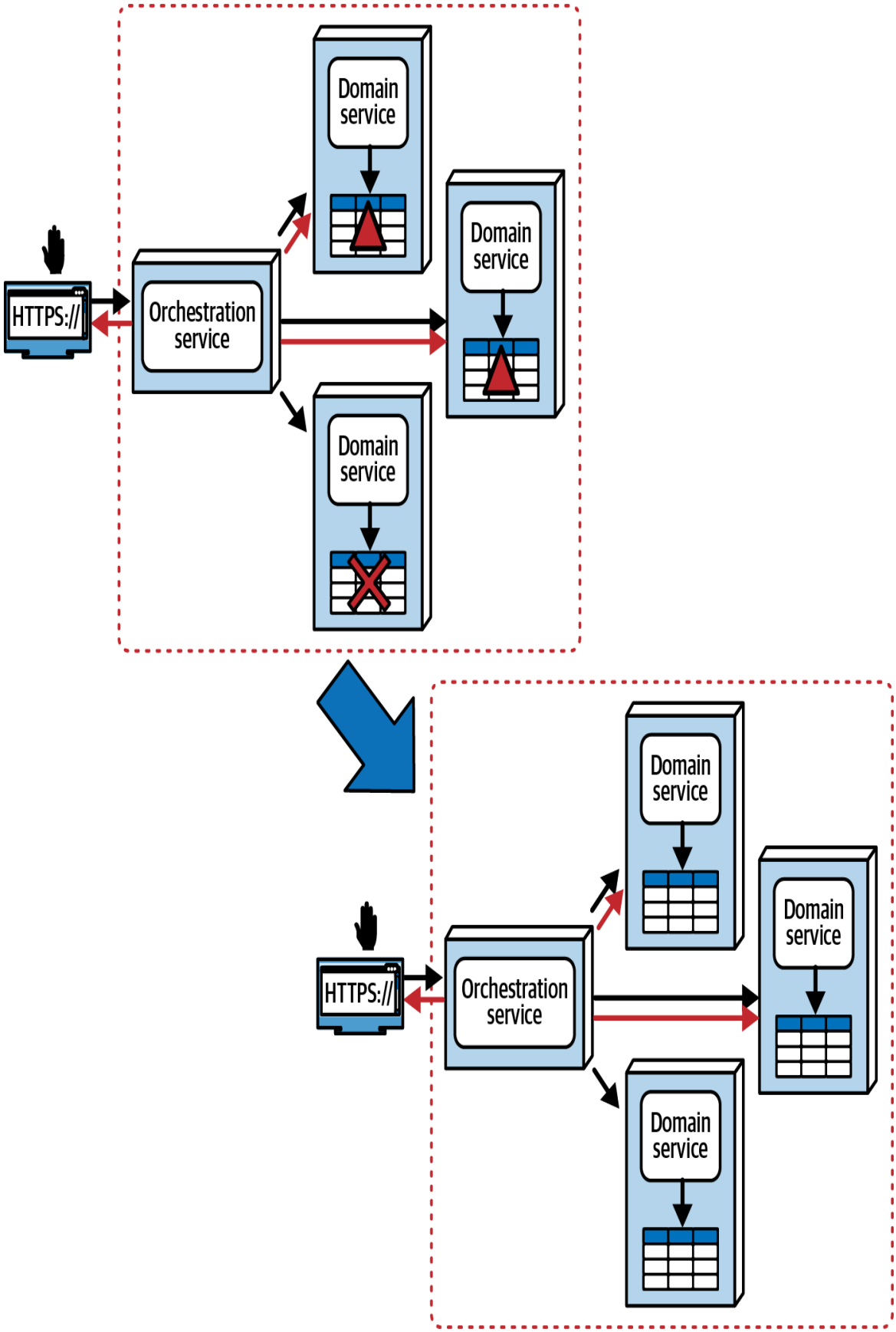

Domain service
Domain service
HTTPS://
Orchestration service
Domain service
Domain service
HTTPS://
Orchestration service
Domain service
Domain service

Figure 11-2. In a compensating update, the orchestrator undoes updates to restore the previous state

Architects may choose to implement this behavior in several different ways: for instance, they might require the domain service to keep track of the old value, or perhaps have the domain service return the old value to the orchestrator as part of the initial workflow, which then sends it back for the compensating update.

Regardless of how the architect implements it, the Compensating Updates pattern requires that some artifact issue compensating updates with appropriate values. Once the orchestrator has coordinated the updates, it returns an error message to the caller indicating that the workflow was not successful. Like a transaction in a traditional ACID database, the end result of the failed workflow leaves the system in the same state as before the workflow executed. The primary difference is that the architect must code the behavior to ensure atomicity and can't rely on an automatic mechanism such as a relational database to handle it.

Compensating Update Failures

Distributed architectures pose many architectural challenges, including how many common patterns have boundary conditions and other important considerations. A common data pattern in these architectures is highlighted above — the idea of compensating updates. However, what if the compensating update fails?

It's an inconvenient truth that using many patterns in the real world requires using other patterns. In the scenario outlined in Figure 11-2, an error in one domain service causes the orchestrator to issue compensating updates to the other domain services, restoring them to their previous state.

It's virtually impossible to derive a generic pattern for recovering from this particular annoying contingency — each solution will be tied closely to the nature of the data and the workflow. For example, many trading firms that use distributed workflows have an *error console*, where errors that the system can't resolve automatically appear for human intervention.

Implementing the Pattern

As we mentioned in Chapter 27, it's common in microservice architectures for each service to feature *data fidelity*. Atomicity resides at the service boundary, as each service "owns" its data within a bounded context. Thus, when an error occurs in a workflow such as Orchestration, the architect must take care to return the fact of the error *and* make sure that compensating updates run.

Consider the simple orchestrated workflow from Chapter 27, which we'll use as a baseline.

Example 11-1. Orchestrated workflow with no compensating update

```
define service orchestrator
define service domain_service1
define service domain_service2
define service domain_service3
define system requester
workflow_state = {SUCCESS, ERROR}

orchestrator.add_services(domain_service1, domain_service2,
domain_service3)
state = SUCCESS
on receive message M from requester
   orchestrator.each |service| do
      if (not service.handle(message)) then
         state = ERROR
      end if
   end do
end on receive
return state
```

In Example 11-1, we define the same orchestrated workflow from the previous chapter except with services instead of event processors. This implies that the domain services may be stateful. That complicates the implementation somewhat, as shown in Example 11-2.

Example 11-2. Orchestrated workflow with compensating update

```
define service orchestrator
define service domain_service1
define service domain_service2
define service domain_service3
define SYSTEM requester
```

```
orchestrator.add_services(domain_service1, domain_service2,
domain_service3)
message = orchestrator.receive_message(initiating_event)
handled_messages = empty list
orchestrator.each |service| do
  if (service.handle(message)) then
     handled_messages.add(message)
  else // error state
     handled_messages.each |service| do
       service.issue_compensating_update()
     end do
     break
   end if
end do
```

In Example 11-2, we add another possible workflow state — `COMPENSATION_FAILURE`. As we've mentioned, two error states are possible with Compensating Updates — the workflow fails for domain reasons or the compensating update fails. In the case of a domain failure, we iterate through the services that have already been updated and issue a request to restore their previous state.

If the compensating update fails, as shown in Example 11-3, as we iterate through each service that has already been processed, we ensure that it can perform the rollback successfully.

Example 11-3. Handling both possible error states

```
define service orchestrator
define service domain_service1
define service domain_service2
define service domain_service3
define system requester
workflow_state = {SUCCESS, ERROR, COMPENSATION_FAILURE}

orchestrator.add_services(domain_service1, domain_service2,
domain_service3)
state = SUCCESS
handled_messages = EMPTY LIST
on-receive message M from requester
   orchestrator.each |service| do
      if (service.handle(message) == ERROR) then
         state = ERROR
         handled_messages.each |service| do
```

```
            if (service.issue_compensating_update() == ERROR)
               state = COMPENSATION_FAILURE
            end if
         end do
      end if
      break
  end do
end on-receive
```

In these examples, we've used status variables to indicate the state of workflows. Architects can implement either of these in other ways, such as using exceptions for compensating update failures.

So far, we've used Orchestration to illustrate the Compensating Updates pattern, because the orchestrator provides a convenient "driver" for the workflow and the corresponding data consistency. It's also possible to implement this pattern in Choreography (see Chapter 9), as shown in Example 11-4.

Example 11-4. Building compensating updates using Choreography

```
define event_processor domain1
define event_processor domain2
define event_processor domain3
define error_processor error
define system requester
message_channel = {SUCCESS, ERROR}

domain1.listen_for initiating_event
domain2.listen_for DOMAIN1_OUTPUT via domain1.SUCCESS
domain3.listen_for DOMAIN2_OUTPUT via domain2.SUCCESS
error.listen_for domain1.ERROR, domain2.ERROR, domain3.ERROR do
   foreach service S in workflow
      S.compensating_update()
   end for
  send ERROR state to requester
end error

send requester.initiating_event to domain1
requester.listen_for domain3.SUCCESS, ERROR
```

In Example 11-4, in addition to the event handlers we wire into the domain workflow, we also create a series of error messages, along with a handler for those messages that reconciles them back to the requester.

Architects must also be on the lookout for boundary conditions; just as above, the compensating update could fail. In that case, the architect should create another event path to handle the additional possible outcome, as illustrated in Example 11-5.

Example 11-5. Creating a channel for compensating update failures

```
define event_processor domain1
define event_processor domain2
define event_processor domain3
define error_processor error
define system requester
message_channel = {SUCCESS, ERROR, COMP_UPDATE_FAILURE}

domain1.listen_for initiating_event
domain2.listen_for domain1.output via domain1.SUCCESS
domain3.listen_for domain2.output via domain2.SUCCESS
error.listen_for domain1.ERROR, domain2.ERROR, domain3.ERROR do
   foreach service S in workflow
      S.listen_for COMP_UPDATE_FAILURE do
         requester.COMP_UPDATE_FAILURE
      end listen_for
      S.compensating_update()
   end for
  send ERROR state to requester
end listen_for

send requester.initiating_event to domain1
requester.listen_for domain3.SUCCESS, ERROR, COMP_UPDATE_FAILURE2
```

Trade-Offs

Compensating Updates is a very popular pattern in microservices because of requirements for data fidelity, which push data and persistence into the service boundary.

Pros

The advantages of applying this pattern include:

Allows pseudotransactional workflows

Many workflows include the semantics of atomicity, colloquially known as "all or nothing": either all the parts of the workflow succeed or none do, replicating the "atomic" property of ACID. However, many distributed architectures require service separation for operational reasons, so architects rely on compensating updates to mimic atomicity.

Cons

The disadvantages of applying this pattern include:

Architects must code the pattern and edge cases by hand

Traditional relational databases offer a number of extremely convenient features, such as ACID transactions. Moving to eventual consistency and compensating updates changes responsibility for the database to the architect and development team.

Pseudotransactional, not transactional

This pattern mimics atomicity but doesn't completely replace it.

Comparison Chart

The Compensating Updates pattern exhibits the trade-offs shown in Table 11-2, where this pattern is contrasted with no effort by the architect to handle compensating updates — architects ignore atomicity (data consistency and integrity) as a capability.

Table 11-2. Comparison chart for the Compensating Updates pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Data Consistency	★	★★★★★
Data Integrity	★	★★★★★
DEGRADED		
Elasticity	★★★★★	★★★
Responsiveness	★★★★★	★★
Scalability	★★★★★	★★★
Simplicity	★★★★★	★
Throughput	★★★★★	★★★

The following sections describe each of the capabilities found in Table 11-2.

Mimicking transactional behavior comes with lots of complexity and trade-offs, as highlighted in Table 11-2. The impacts on each capability appear below.

Improved

The capabilities that are improved when applying this pattern include:

Data Consistency

The purpose of this pattern is for architects to build the same outcomes as *consistency* in ACID, so it isn't surprising that this is the strongest trade-off. Architects use the pattern despite the negative trade-offs to achieve data consistency.

Data Integrity

Similar to data consistency, this is a strong driver for this pattern. For example, a failed compensating update doesn't just leave services disagreeing (consistency) — it can leave an individual service holding a value that's wrong relative to the business invariant (integrity), such as a payment captured for an order that was never fulfilled.

Degraded

The capabilities that are degraded when applying this pattern include:

Elasticity

Any coordination (like data consistency) in a distributed workflow will degrade elasticity because the naming service can't scale services as easily when they have increased coordination.

Responsiveness

While this pattern is excellent for data-related trade-offs, it degrades the operational architectural capabilities, such as responsiveness: issuing compensating updates is time-consuming, often necessitating retries and other slow operations.

Scalability

Like *responsiveness*, *scalability* is degraded when using compensating updates because of the operational overhead of executing more code to ensure data consistency.

Simplicity

Solutions using compensating updates are more complex than ones not using the pattern for a couple of reasons. First, some component must keep track of the state of the data transaction and issue appropriate calls for error handling and boundary conditions, which adds complexity to both topology and communication. Secondly, this pattern offers at least three success and error possibilities concerning data: the workflow was successful and all data was updated, the workflow was unsuccessful and all the data is back to what it was before the workflow started, or the workflow failed and the compensating update also failed, potentially leaving data in an unknown state.

Throughput

Like the other operational capabilities, throughput is also negatively impacted by the data coordination that defines this pattern.

Influences

The Compensating Updates pattern can be applied in any workflow that distributes state storage across data boundaries, which includes event-driven architectures and microservices.

Atomicity

This pattern is a way to "fake" the *atomicity* part of ACID for multistage workflows. In practice, architects use this pattern to achieve some of the benefits of ACID transactions without the ability to actually perform them.

Communication

When designing workflows for distributed architectures, architects choose between synchronous and asynchronous communication. Asynchronous communication allows workflows to work in parallel, which improves the responsiveness of complex workflows, especially during the compensating update phase.

How to Govern This Pattern

Two key aspects of this pattern might require automated governance. One is based on data integrity and the other on operational concerns.

Failure to Achieve the Compensating Update

To ensure that the compensating update has occurred correctly, an architect can build a fitness function that ensures that all services that return affected data return to their previous state, as illustrated in Figure 11-3.

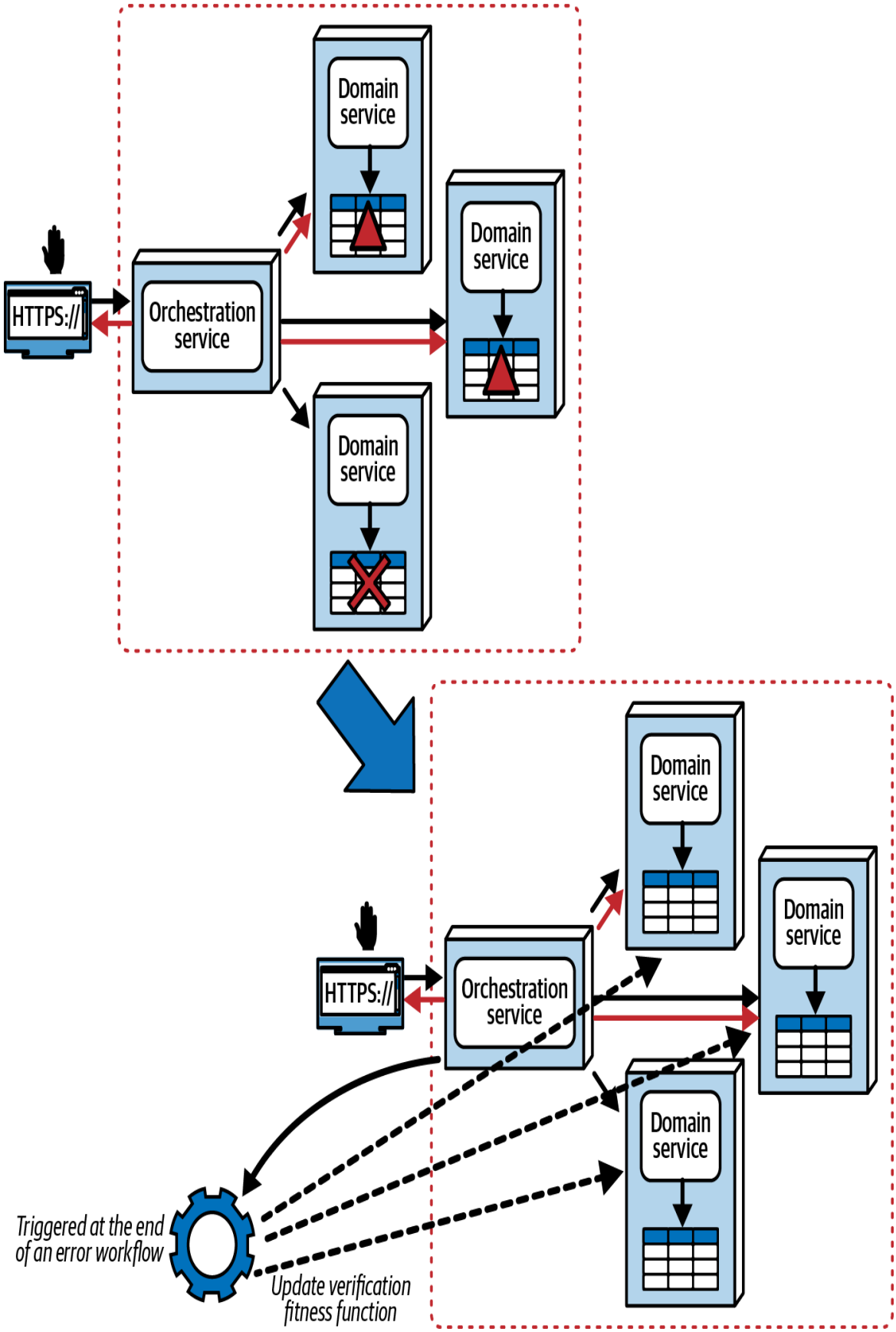

Domain service
Domain service
Domain service
HTTPS://
Orchestration service
Domain service
Domain service
Domain service
HTTPS://
Orchestration service
Triggered at the end of an error workflow
Update verification fitness function

Figure 11-3. A fitness function to ensure that the compensating update succeeds

In Figure 11-3, when the orchestrated workflow completes after an error condition, the fitness function queries each of the affected services to ensure that the data was restored successfully. Generally, this can be a *reactive* fitness function, indicating that it may run parallel to the workflow itself (in other words, this verification takes place in the background and doesn't block the return). If the verification is critical to success, the architect may block the ultimate return until they know the fitness function's outcome.

What Level of Consistency Is Proper?

Many architects like to wave *eventual consistency* around like a magic wand that solves all problems. However, the *eventual* part is important. How eventual is "eventual"? Several microseconds or a few hours?

Architects often build responsiveness fitness functions for their workflows, especially for complex workflows that utilize a lot of synchronous communication. Complex workflows are likely to use orchestrators, and synchronous communication implies that each service request blocks and waits for completion before invoking the next service.

Responsiveness and performance measures typically use operational or DevOps tools that implement observability, such as monitors and frameworks.

Related Patterns, Antipatterns, and Pitfalls

The following patterns are related to Compensating Updates:

Orchestration pattern (see Chapter 27)

> Any distributed workflow can utilize Compensating Updates. This pattern is particularly well suited to orchestrated workflows because the orchestrator provides an excellent coordination point to invoke the compensating calls.

Choreography pattern (see Chapter 9)

As the other primary workflow pattern for distributed architectures, Choreography also commonly uses Compensating Updates for pseudotransactional workflows.

Chapter 12. Contextual Events Pattern

Services in an event-driven system use event chaining (see the Async Chain of Persistence pattern in Chapter 6) to publish derived events after processing an event, advertising to the rest of the system what processing they did and the corresponding outcomes of that processing. Architects and developers can choose to have services trigger a single generic derived event, with the event payload indicating the processing outcome, or trigger multiple explicit contextual derived events, indicating the processing outcome of each specific event. The Contextual Events pattern helps architects and developers make that choice.

Table 12-1 summarizes the metadata for the Contextual Events pattern.

Table 12-1. Metadata for the Contextual Events pattern

CATEGORY	METADATA
CONTEXT:	Event-Driven Architecture, Event Granularity
IMPROVED:	Elasticity, Scalability, Testability, Throughput, Workflow
DEGRADED:	Simplicity

How Many Events Should I Trigger from a Single Service?

In an event-driven architecture, when a service (formerly known as an *event processor*) finishes processing an event, it triggers a derived event informing the rest of the system what actions it performed and the outcome of those actions. For example, consider the scenario illustrated in Figure 12-1, where a customer uses their credit card for a purchase. This diagram shows an event chain within the financial institution that issued the credit card.

The *initiating event* (the event that starts the workflow) is `credit card charged`. Next, a `Fraud Detection` service responds, uses internal algorithms to determine whether the charge is fraudulent or not, and broadcasts its outcome to the rest of the system as a single `fraud checked` event. That event has two possible outcomes: fraud is detected or no fraud is detected. If fraud is detected, the `Credit Card Locking` service locks the customer's credit card to prevent further charges, and the `Customer Notify` service notifies the customer through email and an SMS text. If no fraud is detected, the `Spending Profile` service updates the customer's spending profile to help tune the `Fraud Detection` service's algorithms.

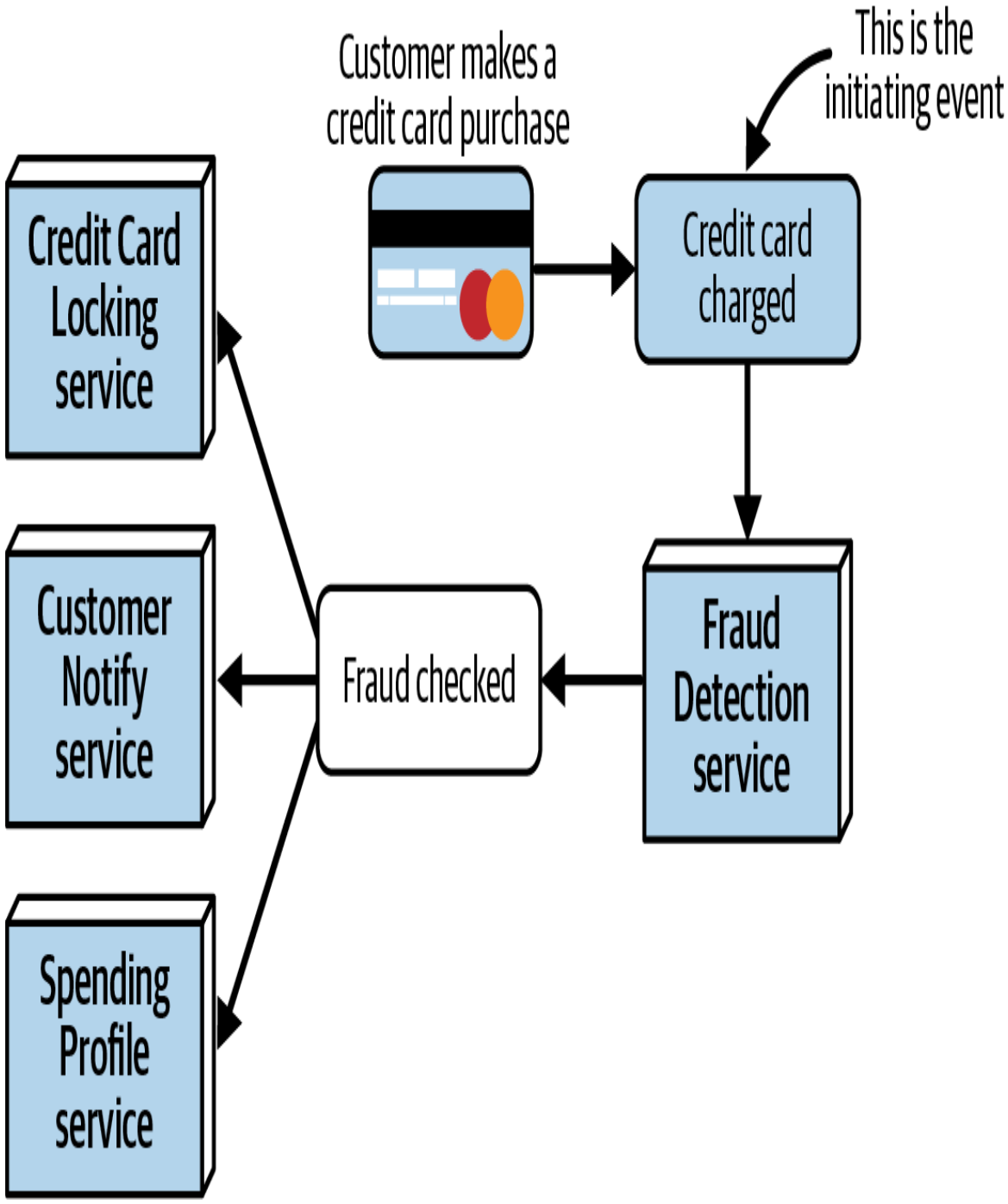


Figure 12-1. A payment triggers a single derived event; a flag in its payload indicates the outcome

In this scenario, the outcome (whether fraud was detected or not) is contained within the payload of a single generic `fraud checked` derived event in the form of a flag. While this practice reduces the overall number of derived events triggered in the system, it requires *all* services interested in the outcome of the fraud detection processing to respond to the event and

check the payload to see if they need to take any action. Because all services must respond to the single generic derived event (whether or not they need to perform any processing based on the outcome), it's not possible to determine the actual workflows of this system based on particular outcomes from the diagram alone. The architect or developer would need to dive into the source code to figure out the specific workflows and learn which services respond to which outcomes.

To illustrate this point, assume that no fraud was detected for the credit card charge shown in Figure 12-2. Because a single generic derived event has been triggered, the `Credit Card Locking` service and the `Customer Notify` service must respond to the event and interrogate the payload, just to find out that no fraud was detected and no further processing is necessary in either service. This is unnecessary. The only service that needs to do any processing if no fraud is detected is the `Spending Profile` service, which updates the spending profile.

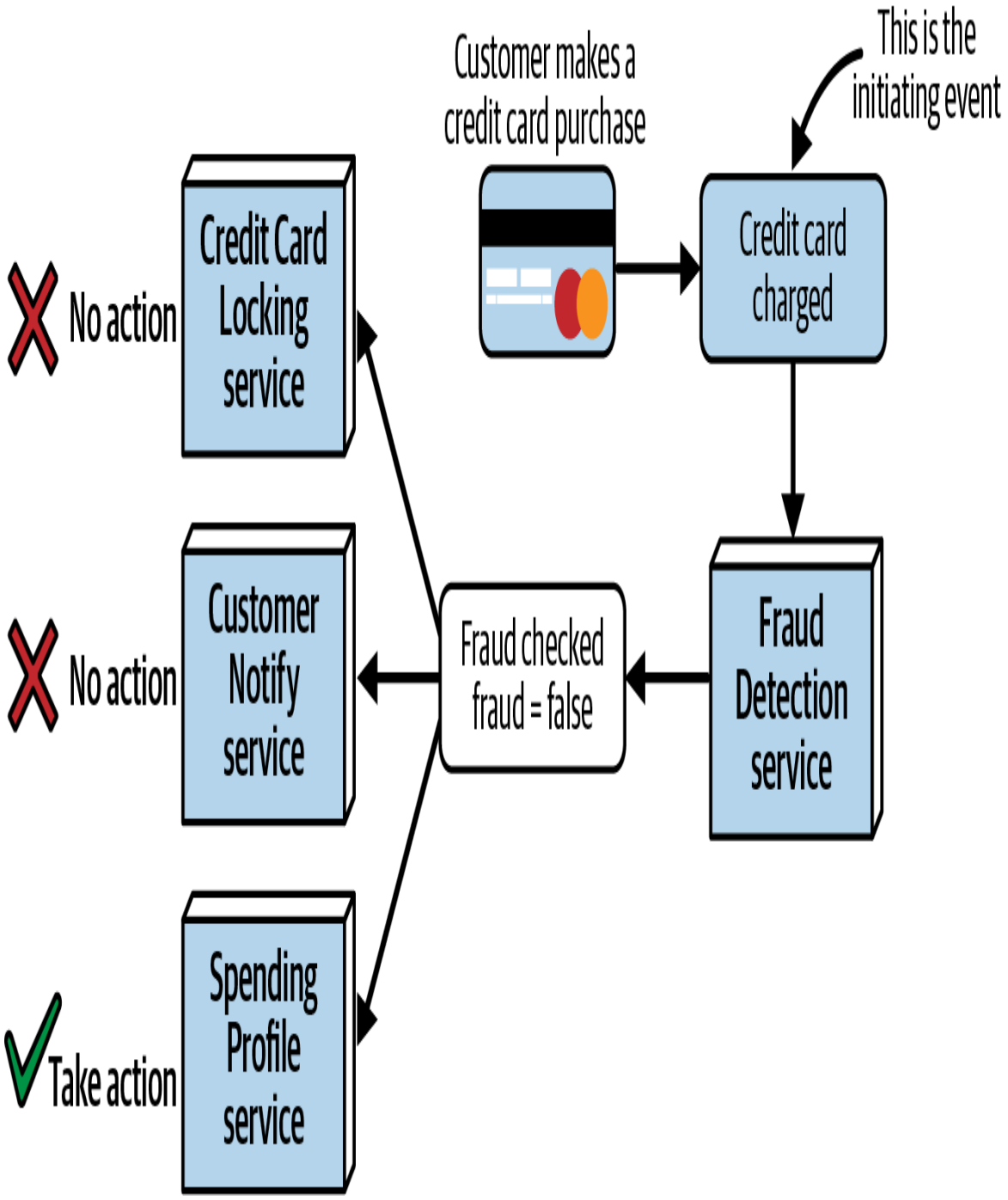


Figure 12-2. Processing workflow when no fraud is detected, with a single generic event

The same problem occurs if fraud *is* detected. In this case (illustrated in Figure 12-3), all services must still respond, but this time the `Credit Card Locking` service and the `Customer Notify` service must both take action to lock the credit card and notify the customer. However, even though the `Spending Profile` service does not take any action to

update the customer's spending profile if fraud is detected, it still must unnecessarily respond to the same event.

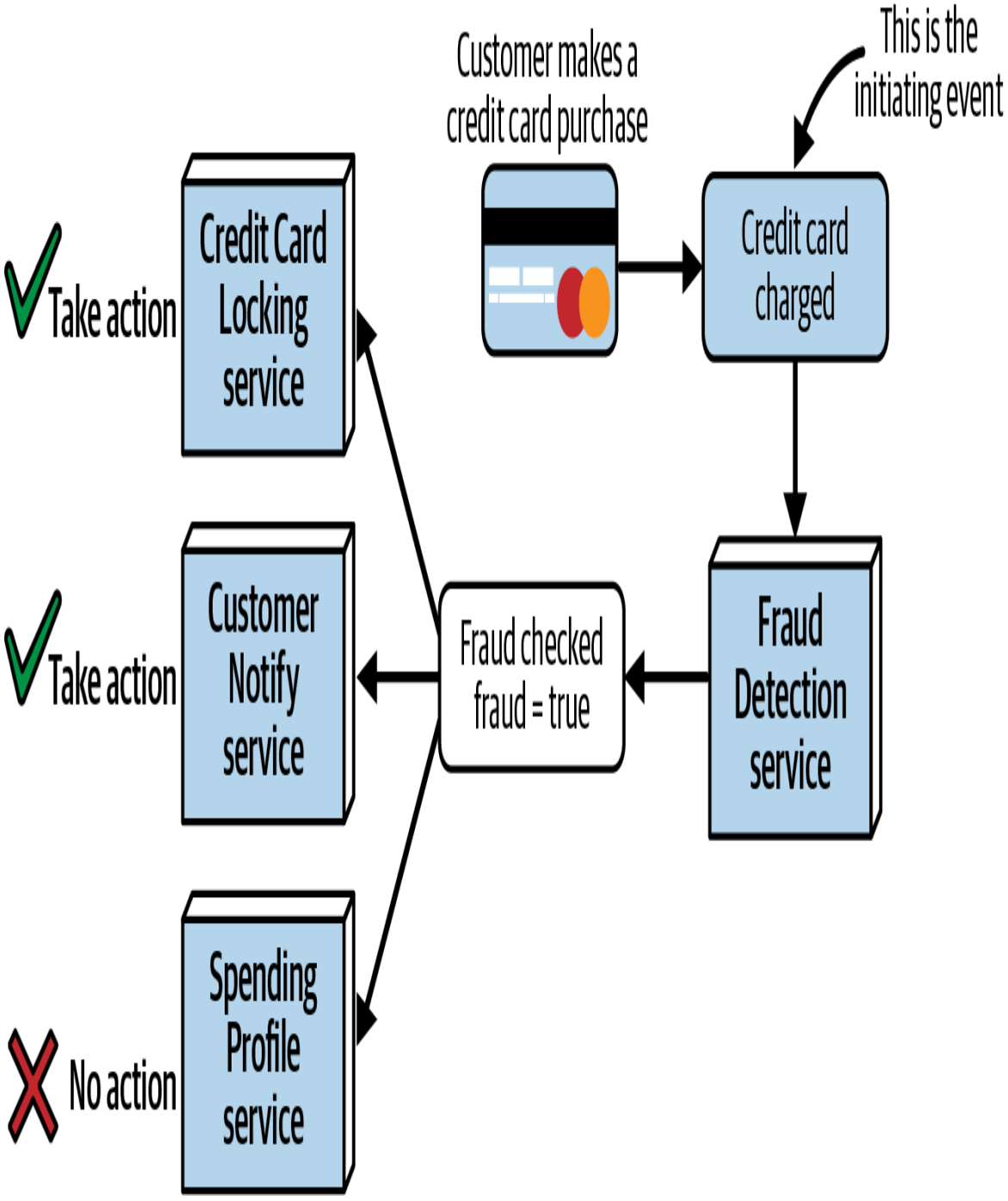


Figure 12-3. Processing workflow when fraud is detected with a single generic event

A much more effective approach to this problem is to trigger an explicit contextual derived event for each possible outcome, which is what this

pattern is all about.

Applying the Pattern

When applying the Contextual Events pattern, each service triggers an *explicit contextual derived event*: a derived event whose name or type explicitly indicates the processing outcome, without using internal payload flags. This means that if a given service has more than one possible outcome, that service would trigger more than one type of derived event.

With the Contextual Events pattern in place, the `Fraud Detection` service in this example scenario would trigger one of two possible derived events rather than just one generic event: a `fraud detected` event indicating to the rest of the system that possible fraud was detected, or a `fraud not detected` event. As illustrated in Figure 12-4, the workflow for each of these outcomes is now clearer, because only those services interested in the particular event indicating the outcome need to respond.

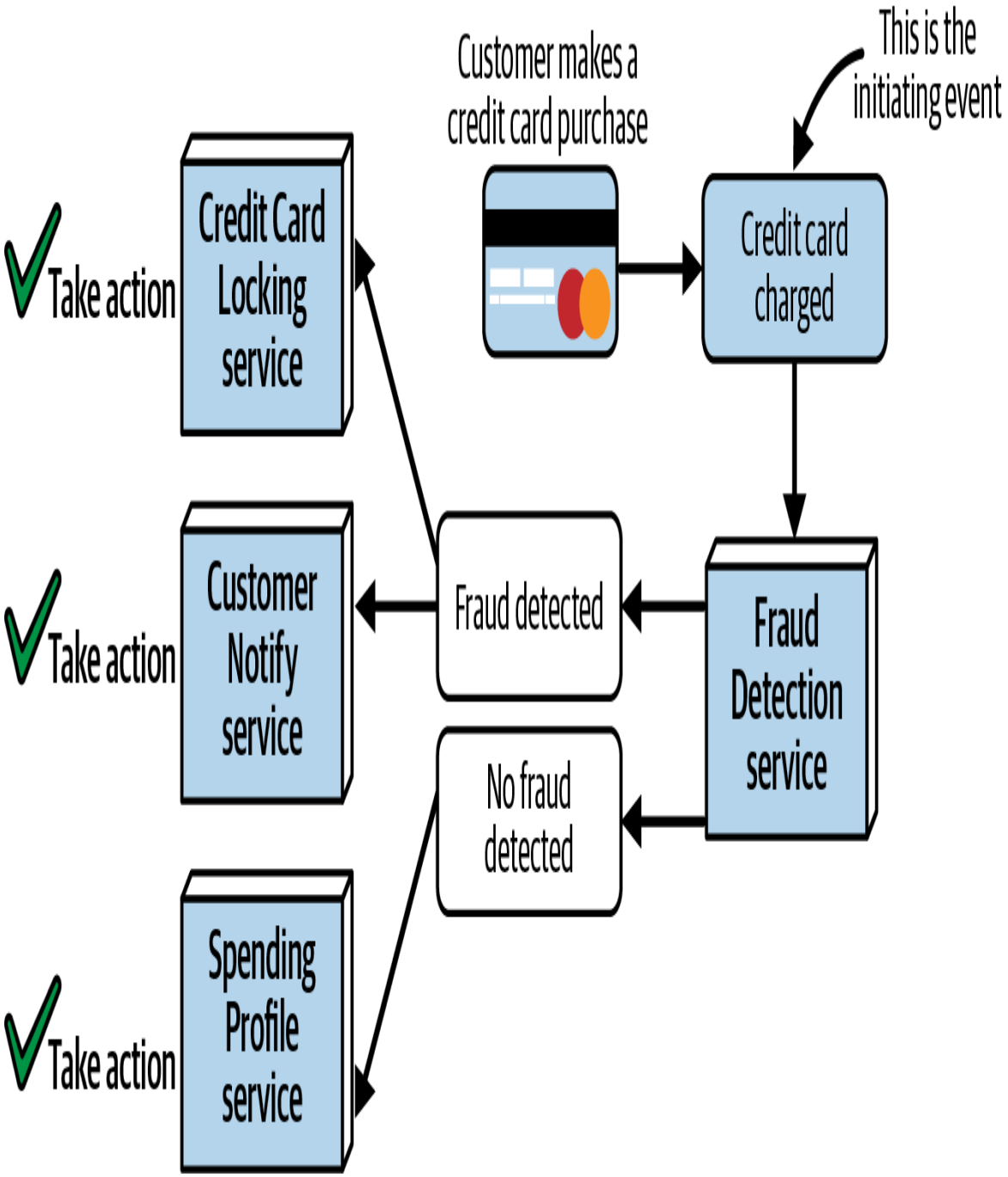


Figure 12-4. Triggering explicit contextual derived events to declaratively advertise the outcome of the fraud detection check

The services in Figure 12-4 are no longer wasting time and processing power to determine whether they need to respond. Rather, services subscribe only to those events whose outcomes would require them to take action. In this example, the `Credit Card Locking` and `Customer`

`Notify` services respond and take action only if fraud is detected, so they only subscribe to the `fraud detected` event. The `Spending Profile` service only responds to nonfraudulent charges to update the customer's purchase profile, so it only subscribes to the `fraud not detected` event.

Applying this pattern makes overall workflows clearer and easier to follow and understand. Compare the use of explicit contextual derived events in Figure 12-4 with the single generic event with payload flags in Figure 12-1: the derived event's name and context make clear which services respond to which outcomes, as well as the corresponding workflows.

When applying this pattern, it's a good idea to identify *all* the possible outcomes of a service and trigger a contextual derived event for each one, even if no other services are interested in some outcomes (see the Extensible Events pattern in Chapter 18). Here's another example: when a customer makes a credit card purchase, a `Credit Limit Verification` service checks the card's credit limit. One obvious possible outcome is that, if the purchase exceeds the credit limit, the charge is declined. However, what other possible outcomes might arise from a credit limit check? Without going too far overboard, here is a list:

- The credit limit is OK
- The customer's balance is close to the maximum credit limit
- The credit limit has been reached with this purchase
- The credit limit has been exceeded with this purchase

Applying this pattern means the service would trigger a *separate* derived event for each of these outcomes, as opposed to using a single event with flags in the payload. As illustrated in Figure 12-5, these outcomes translate into four separate derived events: `credit limit okay`, `credit limit warning`, `credit limit reached`, and `credit limit exceeded`.

In this example, an `Increase Credit Limit` service might respond to the `credit limit close`, `credit limit reached`, and `credit limit exceeded` events by simply declining the charge, or by increasing the customer's credit limit, based on their payment history and credit score. A `Customer Notify` service might also respond to the `credit limit reached` and `credit limit exceeded` events to notify the customer that the credit limit has been reached or exceeded. But suppose no other service is interested in the `credit limit okay` event. The service should nevertheless trigger this possible outcome. (See the Extensible Events pattern in Chapter 18 for more.)

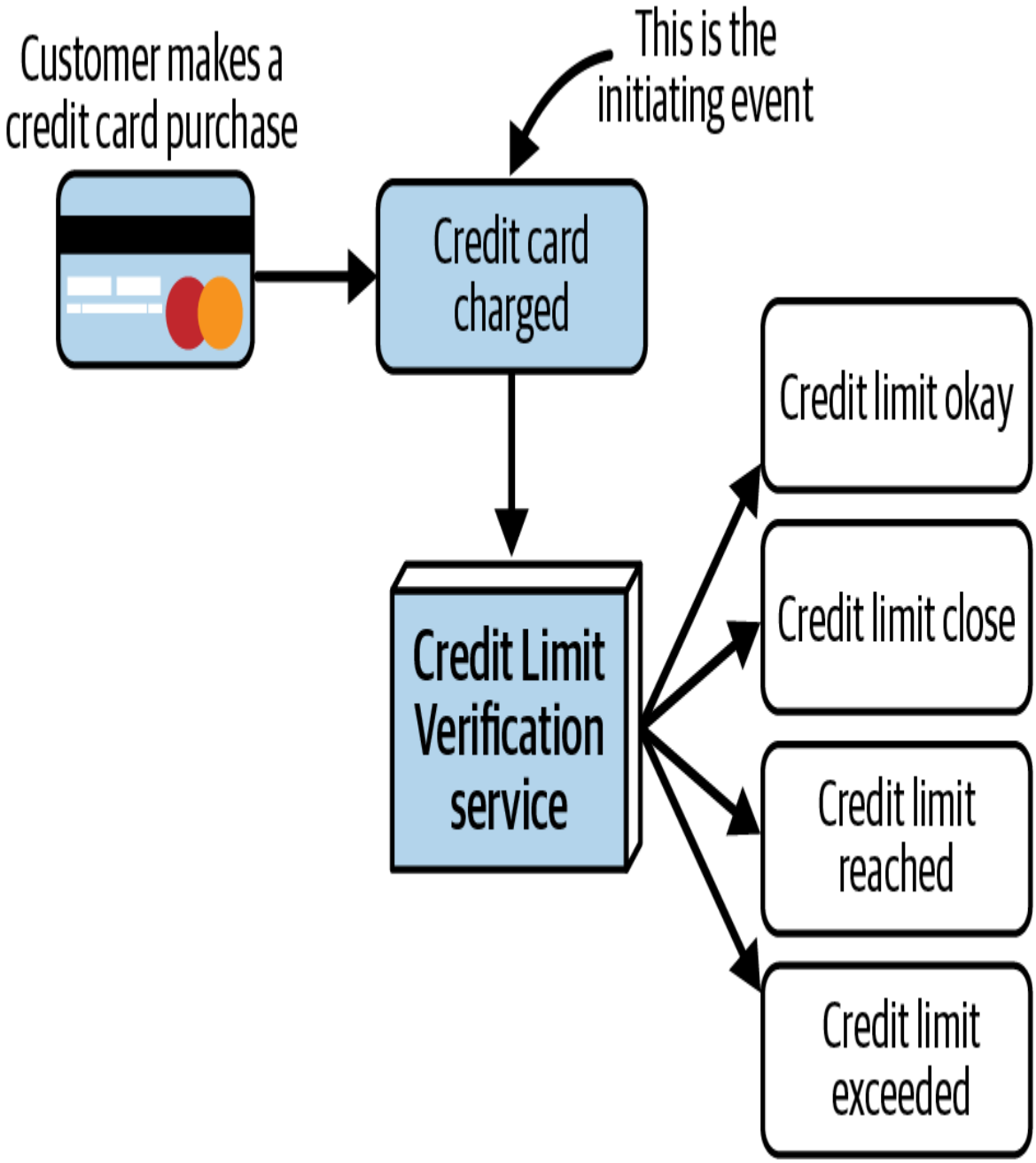


Figure 12-5. Each possible outcome has a corresponding derived event

One thing teams should be aware of when using the Contextual Events pattern is not falling into the "Swarm of Gnats" trap. Gnats (sometimes called no-see-ums) are very small insects that usually fly in large numbers called clouds. They're extremely annoying, particularly when walking in wooded areas or in fields. Being attracted to fluids secreted from the eye, gnats typically swarm directly in front of the face, making them even more

of a nuisance. The same can be said for services that trigger *too many* contextual events.

Consider the example shown in Figure 12-6, simplified to only show the `credit limit close` and `credit limit reached` derived contextual events.

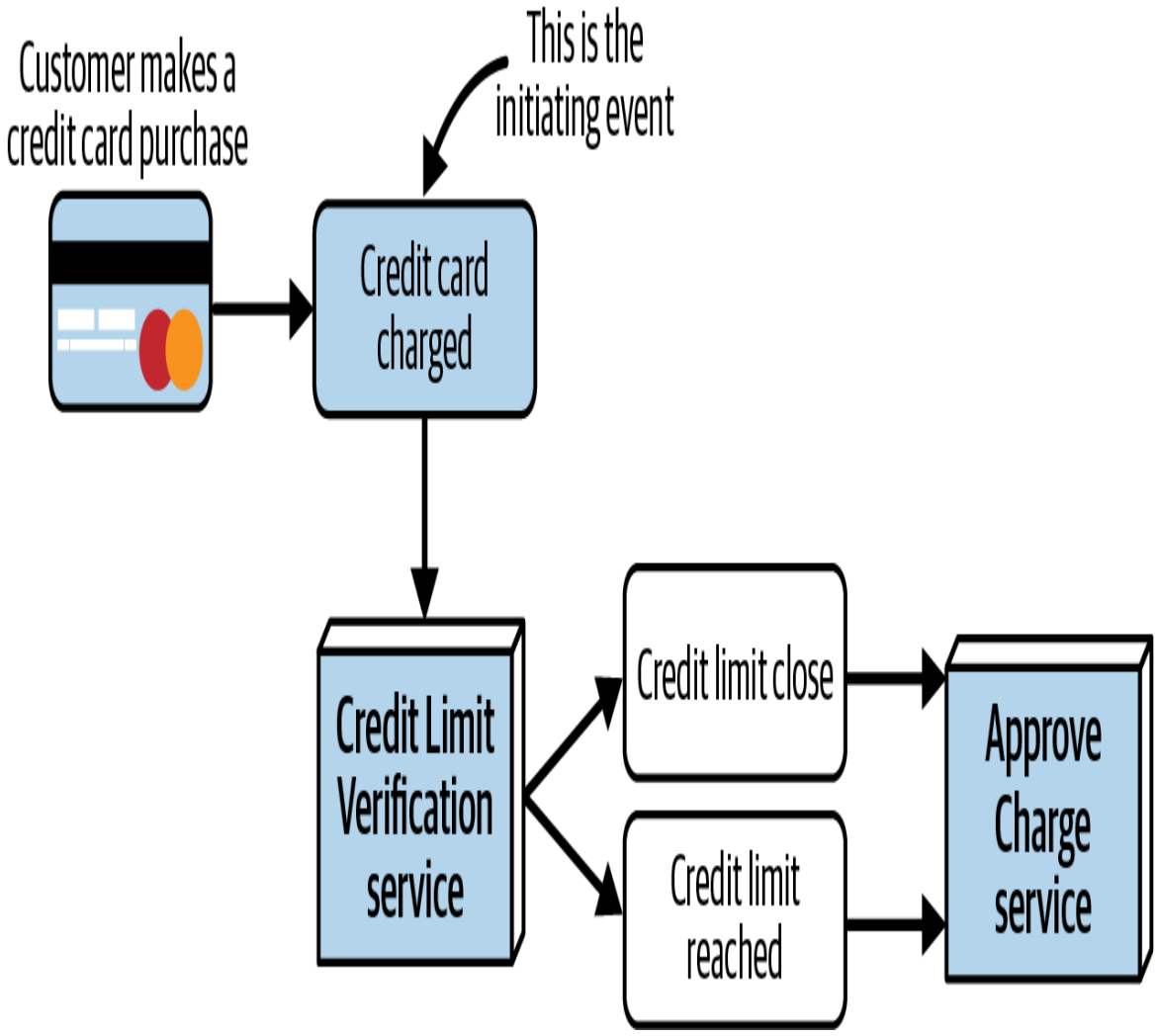


Figure 12-6. Multiple contextual events that only one service reacts to can lead to the Swarm of Gnats trap

Notice that only one service, `Approve Charge`, responds to these contextual events. While applying this pattern the team correctly identified the possible outcomes and created the corresponding contextual events, but in doing so fell into the Swarm of Gnats trap by creating contextual events that are *too* fine-grained. Since only a single service reacts to these events,

nothing is gained by separating these outcomes as the `Approve Charge` service must now subscribe to two separate events.

Teams can avoid the Swarm of Gnats trap by analyzing which services are reacting to the contextual events. If only one service is reacting to them, teams can consolidate the contextual events into a single `credit limit warning` event with data containing the remaining amount (anywhere from 10% down to 0% credit amount available), as illustrated in Figure 12-7.

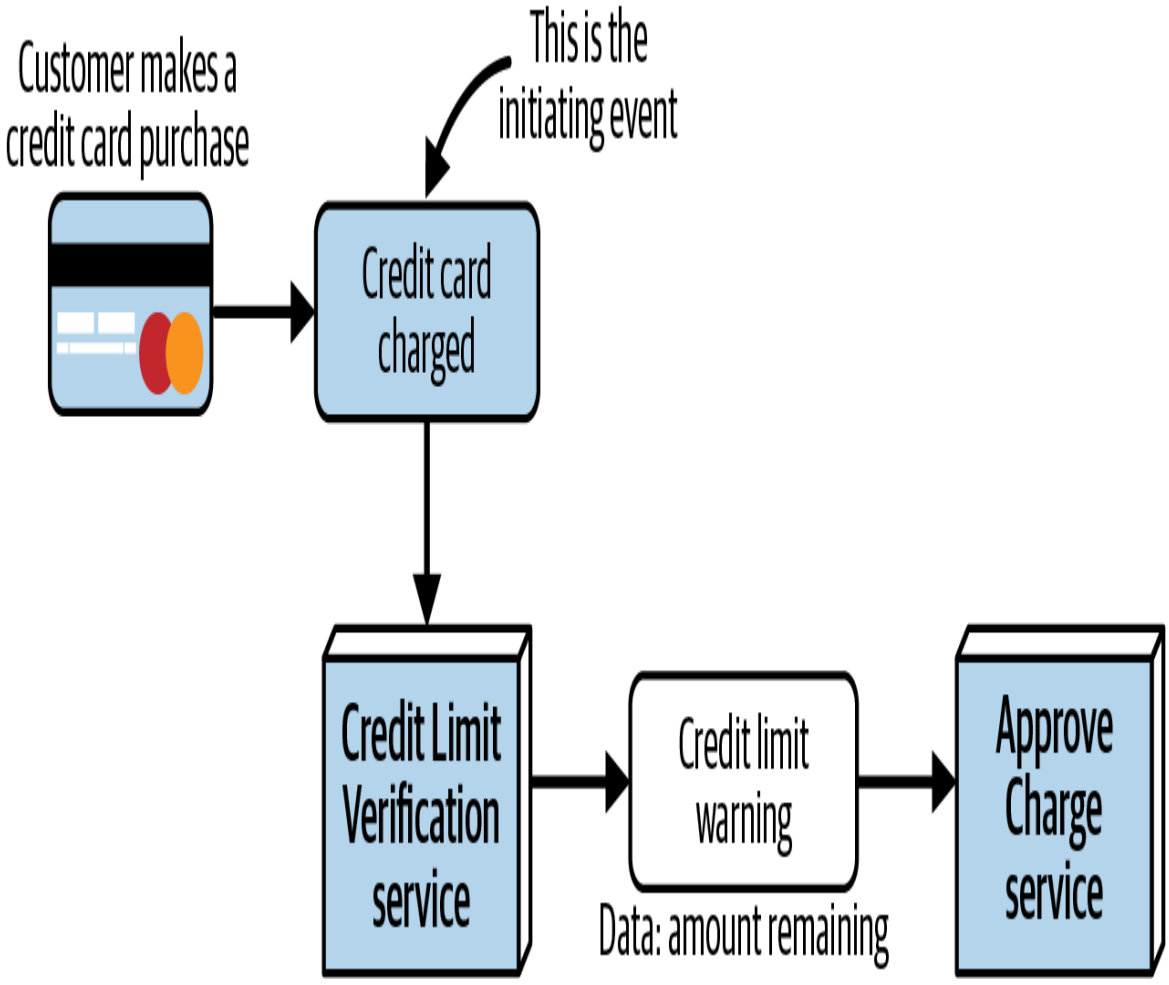


Figure 12-7. Consolidating contextual events avoids the Swarm of Gnats trap

Consolidating the events in this way simplifies the workflow and creates less work for the responding service.

Trade-Offs

Every pattern has trade-offs and consequences, and it's important to consider these when applying this pattern. We've outlined the pros and cons of this pattern in the following sections and in "Comparison Chart".

Pros

The biggest advantage of this pattern is that it improves the understanding and readability of the workflows within an event-driven system. Creating explicit contextual events for each possible outcome from a service means that only services interested in that particular outcome respond, making it easier to identify, trace, and test specific workflows.

Cons

Applying this pattern means that services trigger multiple derived events, one for each possible outcome, rather than only triggering a single derived event containing the outcome in the event payload. This tends to increase the number of derived events triggered in the system, increasing overall event noise (the number of events in the system). With this pattern, derived events become finer-grained, meaning architects and developers must manage a potentially large number of events in the system.

Comparison Chart

Table 12-2 summarizes the star ratings associated with applying this pattern. The Without Pattern column refers to using coarse-grained events with payload flags indicating the processing outcome.

Table 12-2. Comparison chart for the Contextual Events pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Elasticity	★★★	★★★★★
Scalability	★★★	★★★★★
Testability	★★	★★★★★
Throughput	★★★	★★★★★
Workflow	★★	★★★★★
DEGRADED		
Simplicity	★★★★★	★★★★

The following sections describe each of the capabilities found in Table 12-2.

Improved

The capabilities that are improved when applying this pattern include:

Elasticity

> Because services spend less time subscribing to events they don't need to respond to, those services can process more events that are relevant to their processing, therefore increasing the number of events they can

process in a given time. This is particularly important during a spike in user requests.

Scalability

This pattern improves overall scalability because services don't have to respond to events and interrogate the payload to determine whether they need to take any action. This reduces the amount of unnecessary processing each service must do, allowing those services to respond to more events within a given timeframe.

Testability

Reducing the number of services that must respond to a given event means that fewer services test for each possible outcome, so this pattern makes unit and scenario testing easier and faster. Using this pattern also makes scenario and workflow testing easier because it's more explicit and deterministic in terms of which services are involved in the event chain. Furthermore, with this pattern, teams need only perform unit and scenario testing on the *positive* workflows, not the *negative* workflows (such as a service not needing to respond to an event). This significantly reduces the amount of testing needed.

Throughput

When applying this pattern, services are only reacting to events that are relevant to them. This means services don't react to events that don't pertain to them, therefore increasing overall message throughput in the system.

Workflow

Because contextual events are deterministic in terms of which services must react to them, teams can better understand the workflows in the system. This advantage becomes more significant as the system complexity increases.

Degraded

The capabilities that are degraded when applying this pattern include:

Simplicity

> Because teams need to create additional events and corresponding topics, complexity increases a bit — but only by one star — making this an attractive pattern based on the trade-offs involved.

Influences

Since the Contextual Events pattern involves the granularity of triggered derived events, teams must collaborate with those writing the responding services or systems to understand which derived events to respond to and the context of each derived event.

Unfortunately, when integrating systems, it's all too easy for teams to simply trigger a single derived event, with flags indicating the processing outcome. This involves less coordination and communication with other teams about how to respond to the event, and it simplifies the number of topic subscriptions other systems have to worry about. However, this practice only transfers the tasks of communication and coordination to the *contract* — the payload of the event and corresponding outcome flags — so triggering a single derived event isn't much more efficient from a systems-integration standpoint than using contextual events.

How to Govern This Pattern

Leveraging fitness functions can help to govern this pattern statically and ensure that services trigger multiple derived event types. If a particular service fails a test by only triggering a single derived event type, the architect should verify that only one outcome is possible and add that service to an exception list.

It might be tempting to apply this pattern to only some derived events, and leverage generic derived events with payload flags for other types of derived events. We strongly advise against such a hybrid approach when using this pattern because it creates confusion and inconsistency in terms of workflows, scalability, and testability. Therefore, use this pattern for *all* services to avoid this hybrid scenario.

Related Patterns, Antipatterns, and Pitfalls

Because this pattern is typically applied within the context of event-driven architecture, it's related to several other EDA patterns and an antipattern described in this book:

Async Chain of Persistence pattern (see Chapter 6)

> When triggering multiple contextual events, architects should consider leveraging the Async Chain of Persistence for *each* event triggered to ensure that it's not lost during processing.

Data-Based Event Contract pattern (see Chapter 14)

> The size of the event payload (the amount of data passed in the event) can have some negative consequences, potentially degrading scalability, throughput, and responsiveness. While this is not usually an issue with contextual events, architects and developers should nevertheless be aware of the consequences of this pattern when triggering multiple events.

Extensible Events pattern (see Chapter 18)

> The Extensible Events pattern increases the number of derived events in the system. Combining this pattern with the Contextual Events pattern can further increase overall event noise. Architects should consider this trade-off when using extensible events alongside the Contextual Events pattern.

Key-Based Event Contract pattern (see Chapter 23)

Unlike the Data-Based Event Contract pattern, the Key-Based Event Contract pattern only contains the keys of the data entities in message payloads. Architects should understand and consider this pattern when triggering contextual events, since the context of the outcome is already advertised through the name and type of the event.

Stamp Coupling antipattern (see Chapter 31)

Teams should be aware of this antipattern when considering what data to include in a contextual event. While unlikely, adding too much data in a contextual event can lead to the Stamp Coupling antipattern.

Chapter 13. Cross-Cutting State Pattern

Many patterns in distributed architectures end up with negative trade-offs because they don't properly track state of some kind. For example, many workflows in distributed architectures have a *workflow* state — which services have executed along with their results, which services if any have error conditions, and so on. Some architectural patterns (such as the Orchestration pattern [see Chapter 27] and Front Controller pattern [see Chapter 19]) have a built-in location for workflow state — the orchestrator. In fact, often the only state kept in the orchestrator is the workflow state. However, patterns such as Choreography have no predefined place to store workflow state. The Cross-Cutting State pattern provides a way to address those architectural patterns that lack a natural cross-service state holder.

The metadata for the Cross-Cutting State pattern appear in Table 13-1.

Table 13-1. Metadata for the Cross-Cutting State pattern

CATEGORY	METADATA
CONTEXT:	Event-Driven Architecture, Message-Based Architectures, Microservices, Workflow
IMPROVED:	Adaptability, Data Consistency, Data Integrity, Evolvability, Testability
DEGRADED:	Availability, Domain Alignment, Simplicity

How Should I Manage My Workflow State?

One example, from the Compensating Updates pattern (Chapter 11), depicts a user trying to determine the state of their order while it's in process. In an orchestrated workflow, the user can query the orchestrator but has no convenient place to call in Choreography. In fact, the Front Controller pattern (Chapter 19) exists to address this problem. It turns out that the Front Controller is a specialized case of *externalized state*, where an architect decides to persist state to avoid common pitfalls. In this chapter, we examine the Cross-Cutting State pattern.

This pattern has quite broad applications, depending on the particular missing state. For example, in the Front Controller example, the missing state is the workflow state, and the first-called service becomes the holder of that state.

Another common use of the Cross-Cutting State pattern occurs when an architect chooses eventual consistency. Architects must always be on the lookout for side effects in distributed architectures involving data consistency. A *side effect* occurs when some data aspect of a workflow creates an unexpected (and often undesirable) change to other data that isn't supposed to be affected.

For example, when an architect designs an atomic workflow for a typical order-processing system, one step in the process is a call to a `Payment` service, as shown in Figure 13-1.

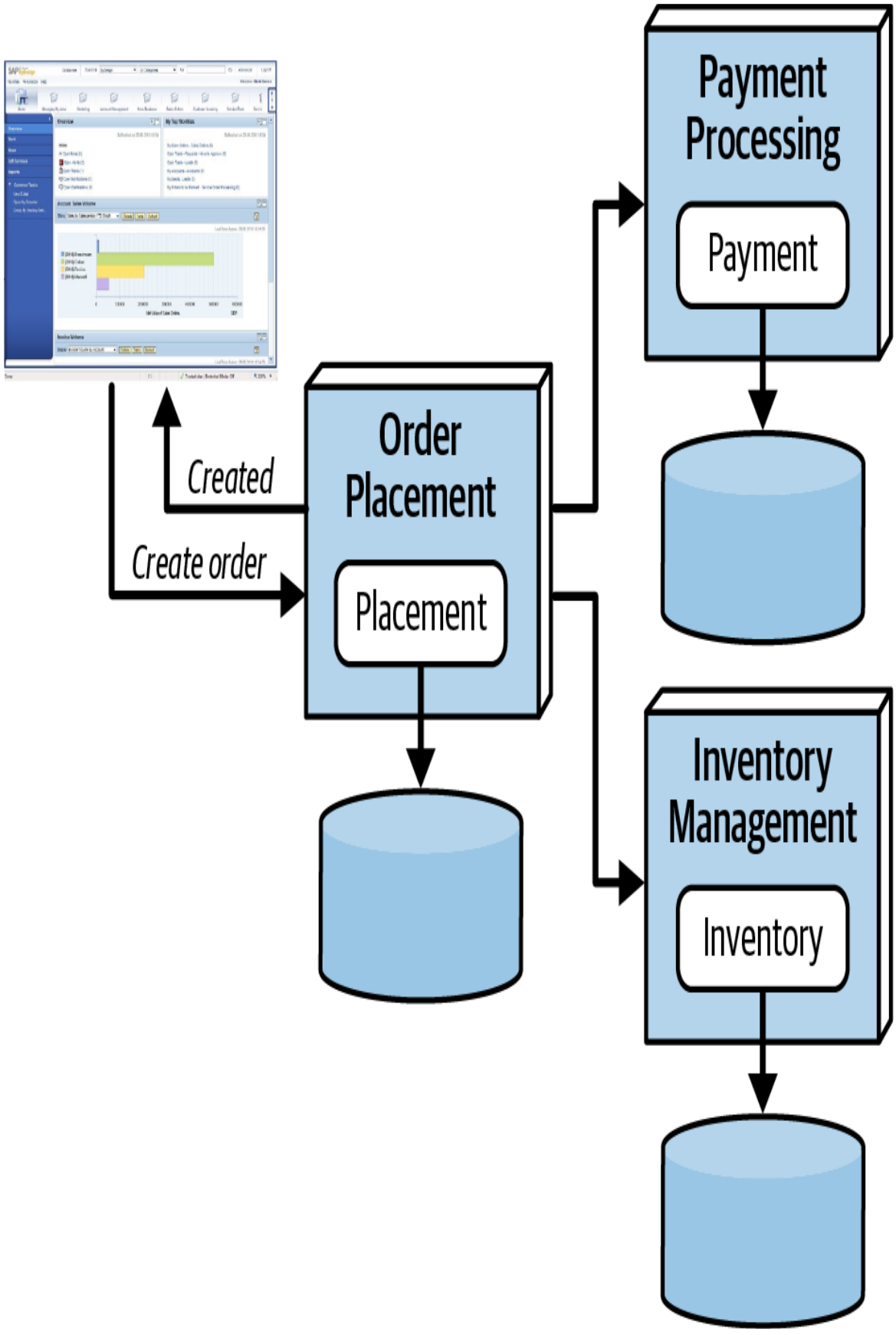

Payment Processing
Payment
Order Placement
Placement
Created
Create order
Inventory Management
Inventory

Figure 13-1. A typical order-processing workflow

For the `OrderPlacement` workflow to succeed, it needs to update both the `PaymentProcessing` and `InventoryManagement` services. However, another workflow that handles month-end sales processing needs to query the `PaymentProcessing` service, as shown in Figure 13-2.

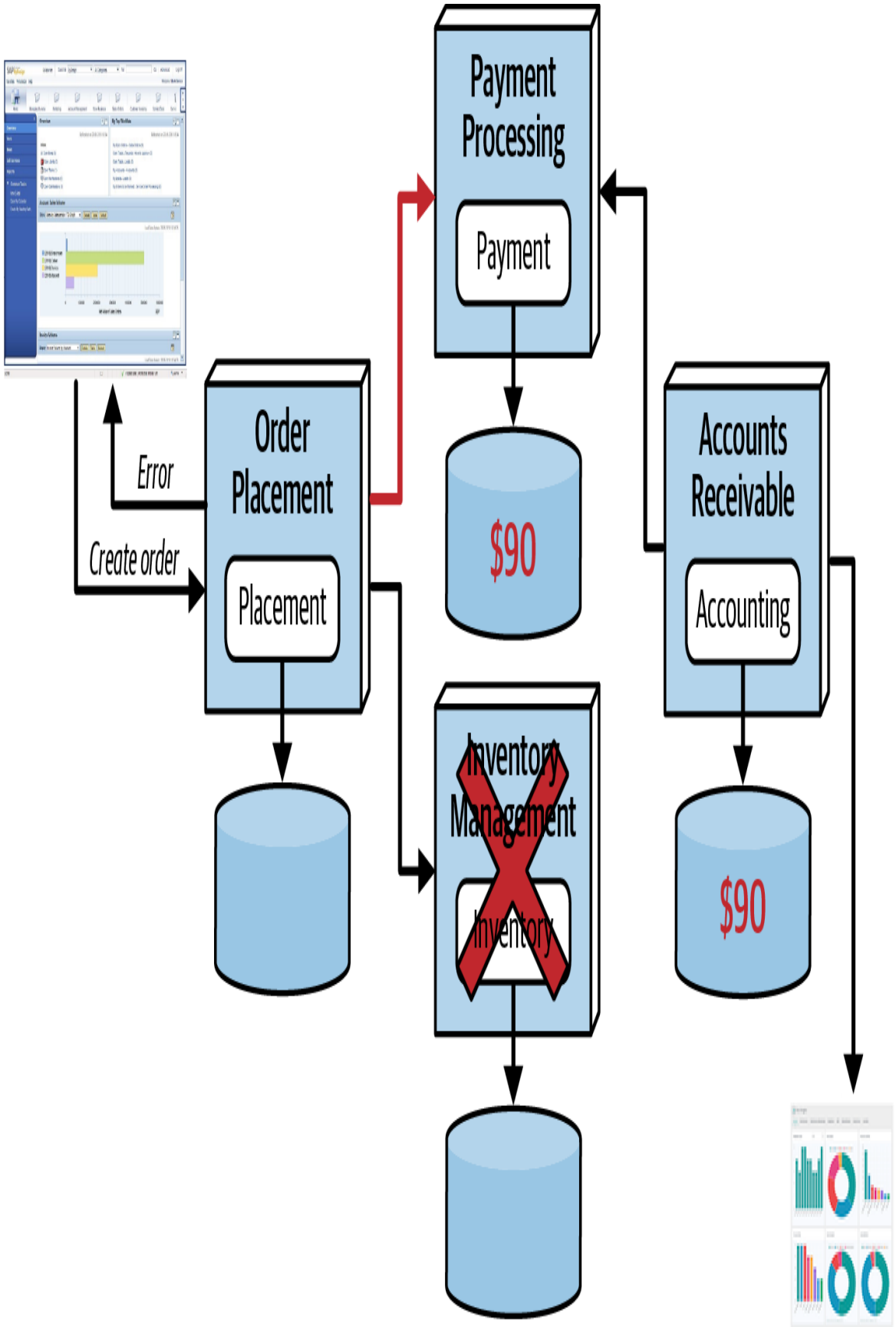

Payment Processing
Payment
Order Placement
Placement
Error
Create order
$90
Accounts Receivable
Accounting
Inventory Management
Inventory
$90

Figure 13-2. While the order-processing workflow runs, an additional workflow might utilize the same services

In Figure 13-2, the `AccountsReceivable` service reads the value of the sale from `PaymentProcessing` as `COMPLETED`. Even though the developers will eventually roll the payment back because the inventory lacks enough of the product, the additional workflow has retrieved a value it shouldn't have.

This illustrates a common problem in distributed architectures generally: *meta-workflow state*, or state about the workflow itself. In this case, the `PaymentProcessing` service maintains the state of the workflow (either `COMPLETED` or `IN PROCESS`) in addition to its domain responsibility. When the `AccountsReceivable` service queries `PaymentProcessing` for values, it's only allowed to retrieve values whose state is `COMPLETED`, not `IN PROCESS`.

The presence of side effects highlights the intersection of data issues and workflows. In fact, in our book *Software Architecture: The Hard Parts*, we closely investigate the synergistic relationship between choice of coordination and atomicity.

Applying the Pattern

To apply this pattern, the architect must determine what external state is missing and how to provide a persistence location for it, providing a wide latitude in ways of implementing it. In many ways, this is a *metapattern*, describing a common problem (keeping state) with many solutions based on problem specifics.

For example, if an architect designs an orchestrated workflow (see Chapter 27), the obvious place to store workflow state is the orchestrator itself. In fact, generally, the orchestrator doesn't contain domain data (which should ideally be the responsibility of the individual services) but rather workflow-specific metadata, like the current state of the workflow. In patterns like Choreography (see Chapter 9), the architect has no predefined

state holder. The common solution to architecture problems with no generic solution is — you guessed it — more patterns! In this case, both the Front Controller pattern (see Chapter 19) and the Backend for Frontend pattern (see Chapter 7) offer specific types of common solutions to the externalized state problem.

In the case of compensating updates side effects, illustrated in Figure 13-2, `PaymentProcessing` holds the pertinent workflow state, and different stages of the workflow update that state, as illustrated in Figure 13-3.

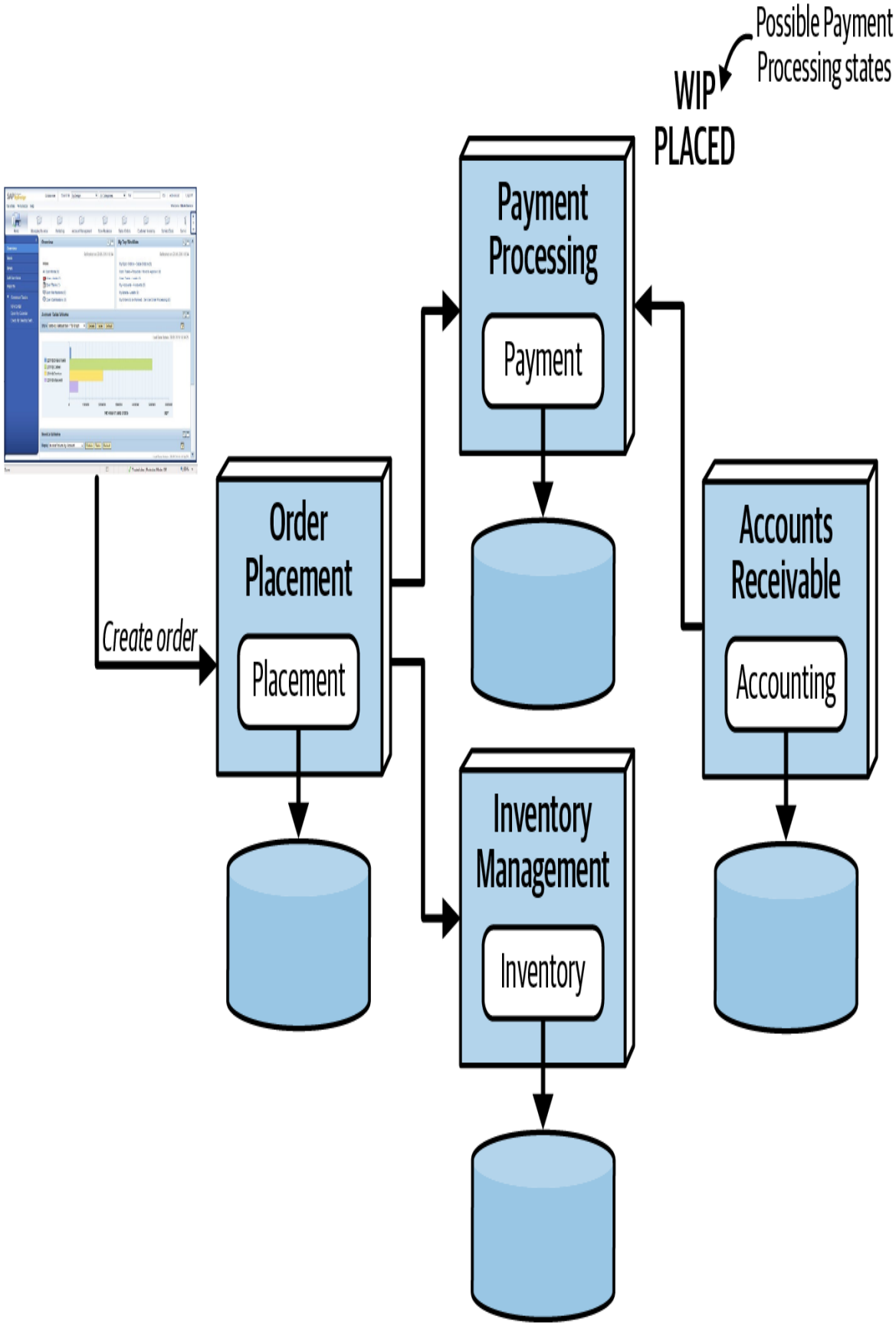

Possible Payment Processing states
WIP
PLACED
Payment Processing
Payment
Order Placement
Placement
Create order
Accounts Receivable
Accounting
Inventory Management
Inventory

Figure 13-3. In the side effect example, external state resides in the `Payment Processing` service

In Figure 13-3, the architect adds code to the `Payment Processing` service to add the state of the workflow, either `WIP` (work in progress) or `PLACED` (order is placed). Any service in the system can query that service to determine the state of the workflow.

To prevent the kind of side effects highlighted in Figure 13-2, the architect additionally adds code to the `Accounts Receivable` service to check the state of the workflow before issuing the query, shown in Figure 13-4.

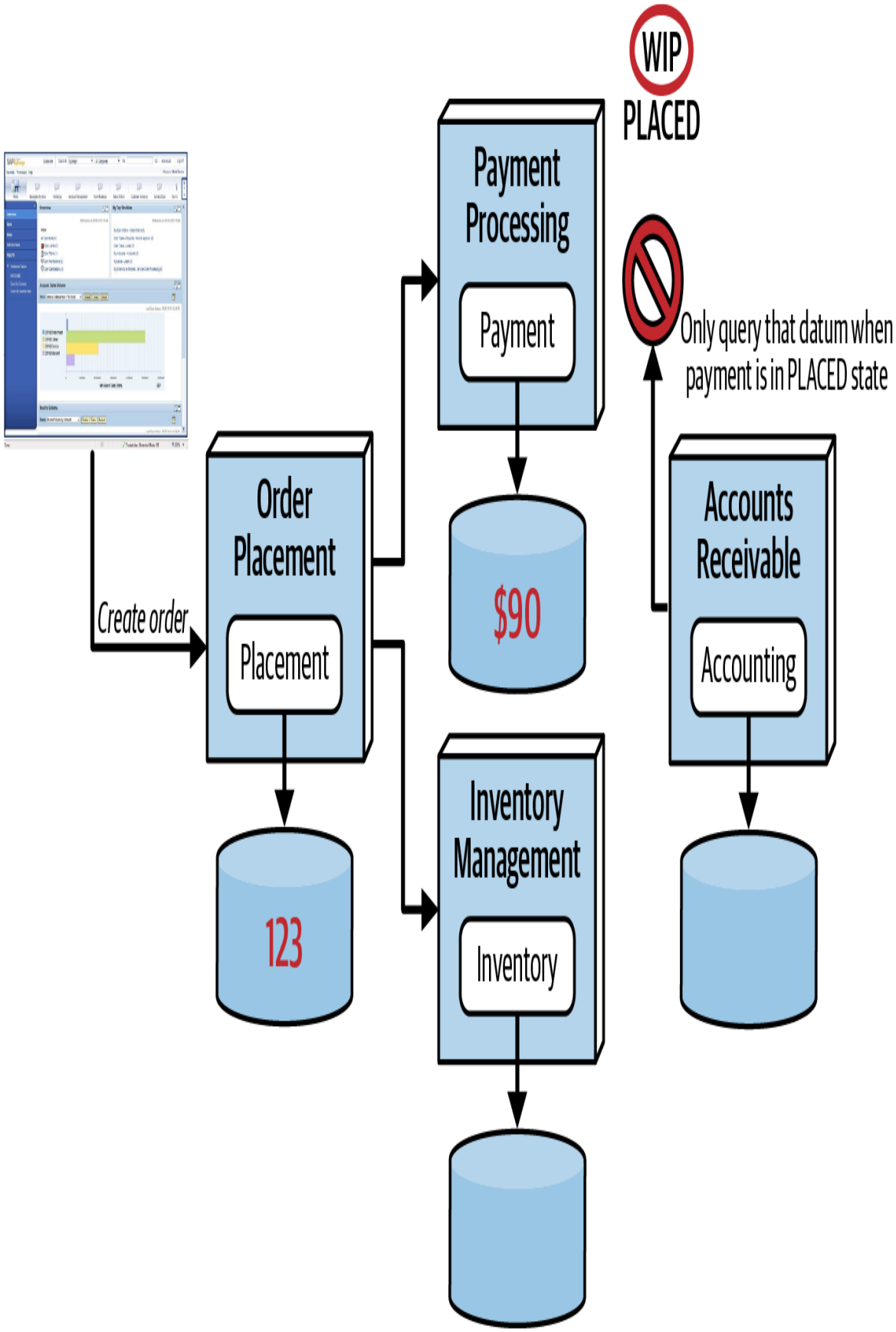

WIP
PLACED
Payment Processing
Payment
Only query that datum when payment is in PLACED state
Create order
Order Placement
Placement
$90
Accounts Receivable
Accounting
123
Inventory Management
Inventory

Figure 13-4. Collaborating services prone to side effects can check for appropriate state before queries

In Figure 13-4, the architect checks for side effects before issuing the query, ensuring that `Accounts Receivable` can't access inappropriate data.

This pattern has many specialized versions, including the Front Controller pattern (see Chapter 19), which includes another example of an implementation of this pattern, in the context of microservices choreography.

Trade-Offs

This section describes the advantages and drawbacks of the Cross-Cutting State pattern.

Pros

The primary advantage of the Cross-Cutting State pattern is to allow architects to control important metainformation about workflows.

Cons

Adding external state adds coupling because the workflow must include another element — unless, like in the Front Controller variant, the architect utilizes an existing service. Doing so degrades cohesion because the service now has two responsibilities, but preserves the existing level of coupling.

Comparison Chart

The trade-off analysis chart for the capabilities affected by this pattern appears in Table 13-2, contrasting this pattern with tracking state by adding it to an existing domain component or service.

Table 13-2. The trade-offs for the Cross-Cutting State pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Adaptability	★★★	★★★★★
Data Consistency	★★	★★★★★
Data Integrity	★★	★★★★★
Evolvability	★★★★	★★★★★
Testability	★★★	★★★★★
DEGRADED		
Availability	★★★★★	★★
Domain Alignment	★★★★★	★★★★
Simplicity	★★★★★	★★★

The following sections describe each of the capabilities found in Table 13-2.

Improved

Teams find the following capabilities generally improved when using this pattern:

Adaptability

Using this pattern allows architects to offload workflow and other incidental state to an external source, leaving domain services more generic and therefore more adaptable.

Data Consistency

One of the primary purposes of this pattern is to preserve data consistency, reflected in the trade-off analysis. Highly decoupled workflows are attractive for many reasons, but the messy reality of data coupling complicates that goal. This pattern provides ways to restore just enough state to follow the semantics of the workflow without unnecessary coupling.

Data Integrity

Like data consistency, externalized state is often used to ensure data integrity for workflows that include numerous domain service interactions.

Evolvability

Keeping state in a well-known location allows for easier evolvability because domain services don't have to hold state, which degrades the possibility for evolution: the more domain aligned the services are, the easier it is for an architect to evolve the workflow. For example, in Figure 13-4, having the external state allows other parts of the application to safely integrate with this workflow.

Testability

Testing is slightly improved with externalized state — it provides architects with well-known testing targets and outcomes.

Degraded

These capabilities are typically degraded when teams choose this pattern:

Availability

Availability is degraded because the state holder must be available in addition to the domain services, increasing the number of services that must be available for success.

Domain Alignment

Domain alignment is tricky because of the alignment difference between *behavior* and *workflow*. From an individual service behavioral standpoint, this pattern provides benefits. Externalizing state allows architects to keep services as closely aligned to their domain purpose without polluting them with added workflow information. When architects don't use this pattern, incidental state related to cross-service data interactions must reside within the domain services, making them more complex.

However, from a workflow standpoint, this pattern degrades domain alignment because each service must now coordinate with the externalized state provider, increasing coupling and aggravating operational capabilities such as availability. Thus, overall, we mark this pattern as slightly degrading domain alignment.

Simplicity

Adding state adds moving parts to workflows in distributed architectures, which correspondingly degrades simplicity.

Influences

The pattern is influenced by many parts of the ecosystem. Workflows often need state *about* the workflow that is separate from the domain data.

How to Govern This Pattern

Architects choose this pattern when they need to capture some external state, which means that state is a good candidate for governance. The development team is responsible for making sure that the states are updated correctly. However, architects may want to ensure that all services *emit* any state changes, which utilizes observability. This is an excellent example of the difference between behavior and capabilities: the actual state changes are part of the system's *behavior*, while ensuring that each service emits state changes is a *capability*.

Typically, in a distributed architecture, the state-ensuring fitness function would run as part of the *governance mesh*, covered in Chapter 26 on the Nexus Architecture pattern.

Related Patterns, Antipatterns, and Pitfalls

This pattern is related to several others dealing with state, including:

Front Controller pattern (see Chapter 19)

> This is a variant of the Cross-Cutting State pattern that places workflow state among the first-called choreographed services.

Compensating Updates pattern (see Chapter 11)

> The pursuit of eventual consistency can introduce problematic side effects. The Cross-Cutting State pattern is often the architect's solution.

Choreography pattern (see Chapter 9)

> In systems that lack an orchestrator, such as choreographed systems, architects often use the Cross-Cutting State pattern to maintain some discrete state in a separate service without going all the way to Orchestration.

Orchestration pattern (see Chapter 27)

> Orchestration can act as an implementation of this pattern, as the orchestrator component makes a perfect place to store state: it's already

part of the topology of the workflow and it already has cross-workflow responsibility.

Explicit Workflow State pattern (see Chapter 17)

The Explicit Workflow State pattern is also an implementation of the Cross-Cutting State pattern: workflow state is one of many types of state that cuts across the workflow components or services.

Chapter 14. Data-Based Event Contract Pattern

When multiple services or systems communicate, the sender and receiver need to agree on what data is being sent and received. This agreement forms a *contract* between those services or systems, specifying the contract type (such as XML or JSON) and the data contained in the payload of the transmission. However, payload contracts get a bit more complicated in event-driven architectures, because multiple services or systems receiving an event might need different kinds of data. The Data-Based Event Contract pattern describes a technique that keeps all of the data associated with the event outcome in the event contract.

Table 14-1 summarizes the metadata for the Data-Based Event Contract pattern.

Table 14-1. Metadata for the Data-Based Event Contract pattern

CATEGORY	METADATA
CONTEXT:	Contracts, Event-Driven Architecture, Message-Based Architectures, Microservices
IMPROVED:	Adaptability, Elasticity, Evolvability, Responsiveness, Scalability, Throughput
DEGRADED:	Bandwidth Usage, Data Consistency, Data Integrity

How Much Data Should I Include in the Event Payload?

In event-driven systems, when a service completes its processing, it triggers derived events advertising what it did (the processing outcome) to the rest of the system. In most cases, these derived events contain data about the outcome of the event processing, which forms the *payload contract* of the event.

For example, suppose a customer places an order for one item in a typical order-processing system. The `Order Placement` service receives the order, which becomes the *initiating event*: the event that kicks off a workflow. It then validates the order, inserts the order into the database, and returns an order ID to the customer. After this processing is complete, the service triggers an `order placed` derived event to the rest of the system advertising that it finished placing an order.

When triggering the `order placed` derived event, the `Order Placement` service must decide what data to send in the event payload (see Figure 14-1).

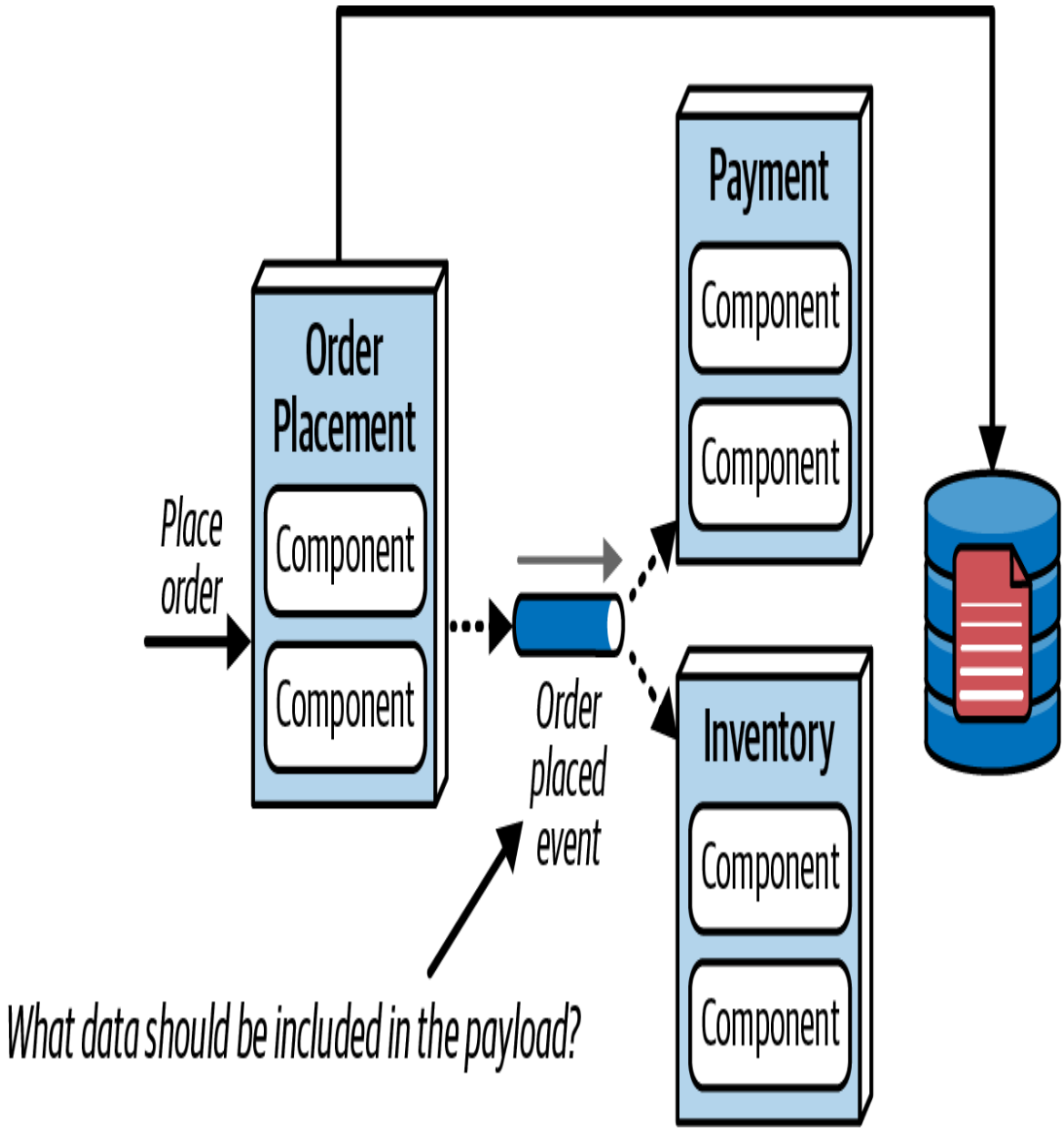


Figure 14-1. The payload of the derived event is the event contract

While this pattern applies to any form of remote communication (REST, messaging, or event processing), this chapter focuses on event-driven architectures, which are the most affected by this pattern.

Applying the Pattern

In the Data-Based Event Contract pattern, the service triggering the event includes all of the data it processed in the event payload, so that all

receiving services have all the data they need for processing without having to query a database to get that information. This pattern is particularly useful if responding services don't have access to that data. Events that have data in them are associated with a *contract*. The contract specifies what data (and corresponding data types) are contained in the event payload. These contracts can be *strict* (containing a specific schema) or *loose* (no associated schema), and can be JSON, XML, an object, and so on. For example, suppose an order processed by the `Order Placement` service in our prior example is a loose 500-KB JSON contract that contains 45 attributes (data items).

The `Order Placement` service would trigger an `order placed` event containing the following JSON attributes:

```
{
  "orderId": 98765,
  "customerId": 123,
  "items": [
    {
      "itemId": 571,
      "desc": "Superhero Cape",
      "qty": 1,
      "weight": .6,
      "unitPrice": 480,
      "backorder": false
    }
  ],
  "totalAmt": 480,
  "customerName": "Clark Kent",
  "superhero": true,
  "email": "superman@heroes.com",
  "phone": "867-5309",
  "billTo": "344 Clinton Street, Apartment 3-D, Metropolis, NY",
  "shipTo": "344 Clinton Street, Apartment 3-D, Metropolis, NY",
  "ccNumber": "6ad301d12ec45",
  "ccExpDate": "4a21b9",
  "ccCcv": "u2a3q",
  "orderCreatedDate": "04-NOV-2026",
  "estShipDate": "06-NOV-2026",
  "member": "true",
  "controlCd": "3E4R556",
  ...
}
```

All of the order information is included in the payload of the `order placed` event, with the credit card information encrypted (of course). When using this pattern, all the data contained in the JSON schema should closely match the data inserted into the database for the order. This means that *any* service responding to the `order placed` derived event has all the information it needs to process it, and doesn't need to query the database.

For example, as illustrated in Figure 14-2, the `Payment` service, upon responding to the `order placed` event, receives the customer's name, bill-to address, credit card information, and total order amount (who knew superhero capes were so expensive?). The `Payment` service can apply the payment without having to query a database for the information. The same is true for the `Inventory` service; responding to the event gives it the `itemId` and `qty` (quantity), so it can adjust the inventory without accessing a database.

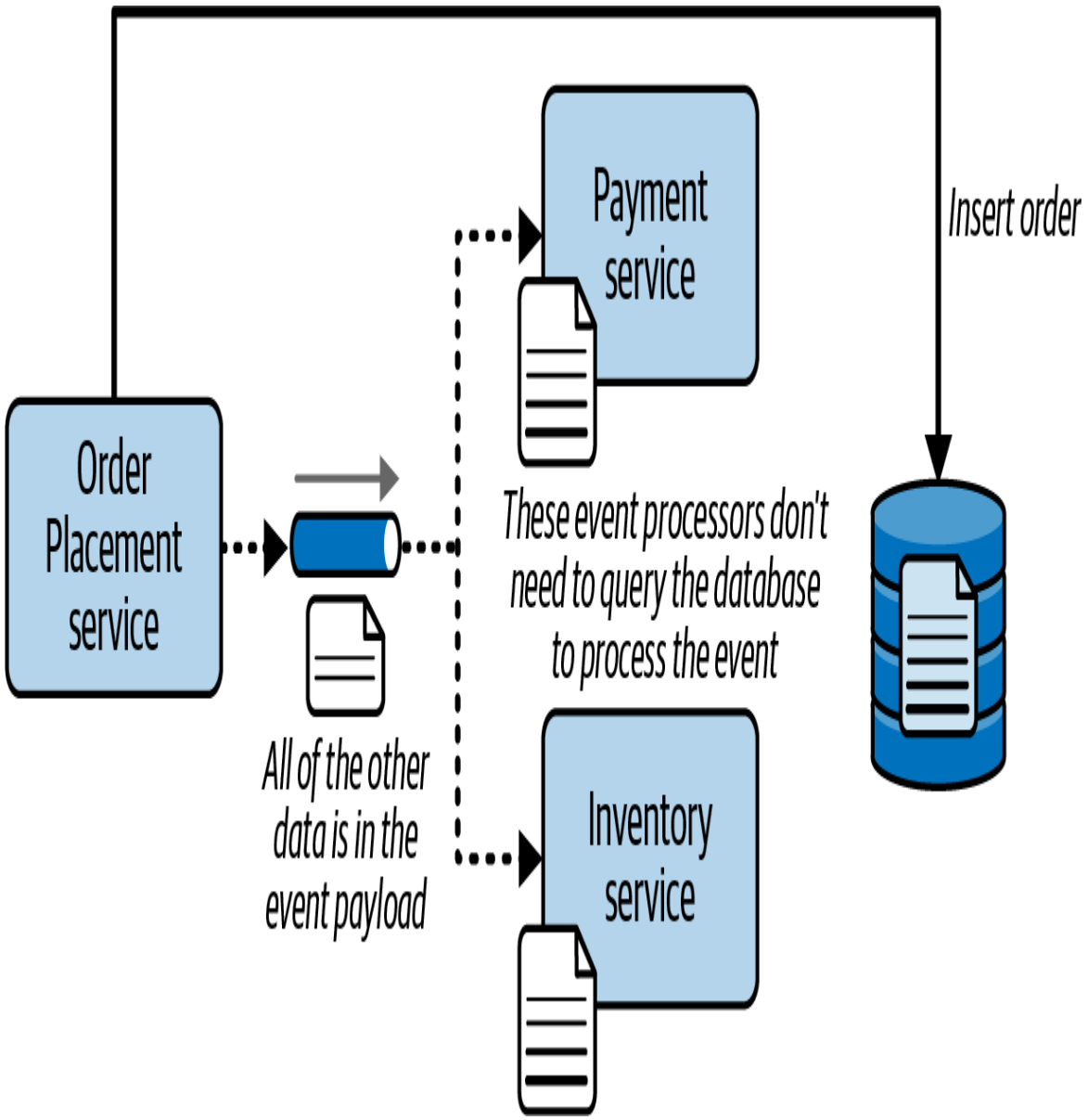


Figure 14-2. All data is included in the event payload

Not only does this pattern improve responsiveness, it improves overall scalability: since services don't have to query the database, the database gets less traffic and has more connections available for other services. The pattern also improves overall evolvability, because other services subscribing and responding to the `order placed` event (such as `Superhero Order Analytics`) will already have the order information at their disposal.

While it's nice to keep an architecture's contract patterns consistent, not *all* event contracts have to utilize this pattern. Some could leverage the closely related Key-Based Event Contract pattern (described in Chapter 23), particularly in areas of the system where scalability and responsiveness are not concerns.

One way to optimize this pattern is to only send *partial* information in the event payload. This improves bandwidth usage by sending less data in the event, and somewhat addresses the Stamp Coupling antipattern issues (see Chapter 31) because the data is limited and contextual only to those services responding to the event.

To illustrate the use of partial data, consider Figure 14-1, where the `Payment` and `Inventory` services are the only ones responding to the `order placed` event. The `Payment` service only requires the customer name, total amount, and credit card info, and the `Inventory` service only requires the item ID and quantity. Therefore, the larger payload containing all of the order information can be reduced to the following JSON payload:

```
{
  "orderId": 98765,
  "customerId": 123,
  "items": [
    {
      "itemId": 571,
      "qty": 1,
    }
  ],
  "totalAmt": 480,
  "customerName": "Clark Kent",
  "ccNumber": "6ad301d12ec45",
  "ccExpDate": "4a21b9",
  "ccCcv": "u2a3q",
}
```

While this improves bandwidth usage a bit, stamp coupling still occurs here because the `Inventory` service doesn't need the credit card or customer information and the `Payment` service doesn't care about the item ID or quantity. Furthermore, this option limits evolvability in that, if another

service subscribes to the `order placed` event, the contract might have to change to add other information, requiring a possible change in the `Payment` and `Inventory` services.

Trade-Offs

While using this pattern provides excellent levels of scalability, throughput, and responsiveness, it has some significant trade-offs architects should take into account, outlined in the following sections.

Pros

The positive outcomes of using the Data-Based Event Contract pattern are illustrated in Figure 14-3.

The advantages of this pattern include improving overall operational capabilities like responsiveness, scalability, and elasticity. Since all the data needed by other services is contained in the event payload, services reacting to the event don't need to query a database to get event information. This improves responsiveness, and services can process more events since they don't have to wait for database connections during peak volumes. "Comparison Chart" describes the operational and process capabilities improved when using this pattern.

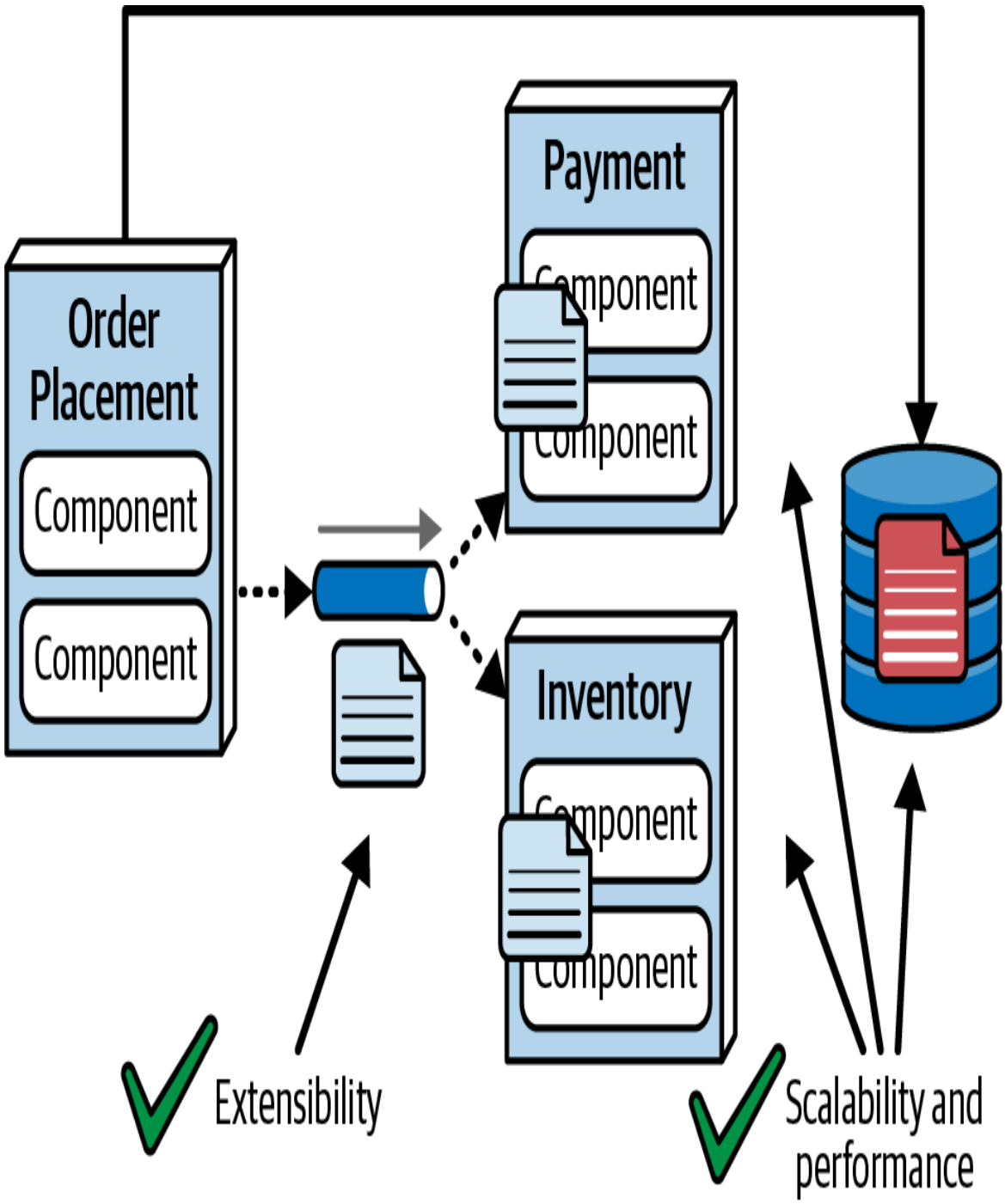


Figure 14-3. Advantages of the Data-Based Event Contract pattern

Cons

The downsides of using the Contextual Events pattern are illustrated in Figure 14-4.

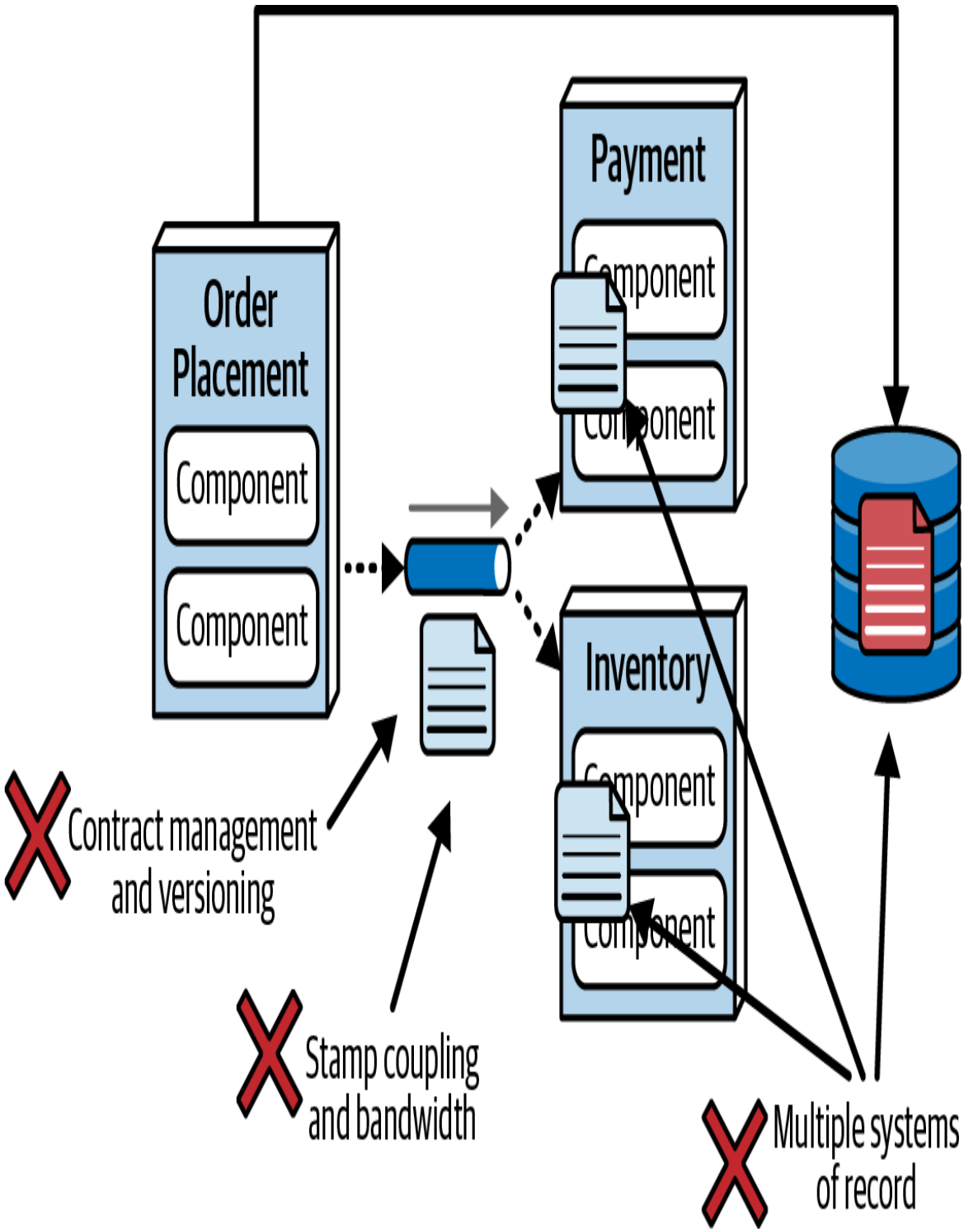


Figure 14-4. Disadvantages of the Data-Based Event Contract pattern

In most systems, the database is the *system of record* (the ultimate source of truth) for the processed data. In Figure 14-1, the order was placed and inserted into the database, so the database becomes the system of record for

that order. However, since services reacting to the event have all the data they need from the event payload, they don't query that system of record. From those services' perspective, the *event itself* becomes the system of record. This can become a problem if the customer immediately updates the order, because now the system might be processing multiple versions (the original order and the updated order) at the same time throughout the system. This issue is further compounded by the nondeterministic nature of event-driven systems and the difficulty of maintaining the event-processing order.

Contract maintenance and versioning is also a disadvantage with this pattern. The developer or architect must make myriad decisions based on the payload: the type of contract (JSON, XML, object), whether the contract should be loose (simple JSON name-value pairs without an associated schema) or strict (containing a schema), and what data should be included. Complex event payload contracts might eventually need to change as requirements change. Contract changes can be challenging in large, complex event-driven systems, because a change in the event payload contract can break responding services — and sometimes it's not clear which services are responding to which events.

One technique to protect the responding services from breaking when a strict contract changes is to introduce versioning into the contract. However, that brings its own challenges, particularly for large organizations and systems. For example, how many versions should a particular contract support before developers begin deprecating older ones? How should developers communicate a contract version change or deprecation to other teams?

Using a data-based event contract means that the responding services don't depend on the schema of the underlying database, but on the event contract's *representation* of that data. Event contracts are much easier to version than database schemas, making them much more agile in dealing with data-structure changes. This is particularly true for loose event contracts: adding a new attribute (and thus a database column) doesn't break

the responding services, but it would if those services had to query the database.

Finally, the Data-Based Event Contract pattern consumes much more bandwidth than the Key-Based Event Contract pattern (see Chapter 23 and "Comparison Chart"). For low-volume systems, this usually isn't an issue, but it can become significant if the volume of requests increases.

Comparison Chart

Because the Data-Based Event Contract pattern is directly related to the Key-Based Event Contract pattern (see Chapter 23), we built a chart comparing these two patterns directly to show their qualitative differences, as shown in Table 14-2.

Table 14-2. Comparison chart for the Data-Based Event Contract pattern

Capability	Data-Based Contract	Key-Based Contract
IMPROVED		
Adaptability	★★★★★	★★★
Elasticity	★★★★★	★★★
Evolvability	★★★★★	★★★
Responsiveness	★★★★★	★★★
Scalability	★★★★★	★★★
Throughput	★★★★★	★★★
DEGRADED		
Bandwidth Usage	★	★★★★★
Data Consistency	★★★	★★★★★
Data Integrity	★★★	★★★★★

The following sections describe each of the capabilities found in Table 14-2.

Improved

The capabilities that are improved when applying this pattern include:

Adaptability

Passing all the data in the event payload allows services to change and adapt without having to modify any event producers, particularly when responding services suddenly require data elements in the event payload they didn't need before.

Elasticity

Having limited connections in the database-connection pool can degrade elasticity and concurrency, and overstress the database's threads, memory, and CPU, particularly for the sudden concurrent demands placed on elastic systems. Because the event payload contains all of the data required to react to the event, responding services don't need to query a database. This means they use fewer database connections, allowing for higher elasticity.

Evolvability

When the event payload contains all of the data, teams can add responding services without contract changes. Using partial data in the event payload might degrade this advantage if additional data requires a change to the payload contract.

Responsiveness

Eliminating a service's need to query the database for event information removes the latency those queries would introduce (particularly when experiencing connection waits), thus improving overall responsiveness.

Scalability

Reducing database query traffic leaves more database connections and resources available for other services, allowing the overall architecture to scale more effectively under heavy concurrent loads.

Throughput

Because services don't have to query a database to retrieve event information, they can process events much faster, increasing overall event throughput.

Degraded

The capabilities that are degraded when applying this pattern include:

Bandwidth Usage

Bandwidth usage can be a significant issue when using data-based event contracts, particularly under high user load. Let's assume the payload for our example order is 500 KB of data, with 45 attributes. At 500 orders per second, this translates to 250,000 KB per second of bandwidth usage. However, the `Inventory` service only needs 30 bytes of data (the order ID and quantity). If the system processes 500 orders per second, that's only 15 KB per second of bandwidth usage — a big difference. The Data-Based Event Contracts pattern can therefore incur a significant expense from bandwidth usage, particularly in cloud-based environments.

Data Consistency

Because services don't query a database — the source of truth — for event data, it's possible for another process to change the underlying data in the database before the original event is processed. The data in the event payload remains static, a snapshot from the time it was published, and can thus get out of sync with the data in the underlying database, degrading data consistency.

Data Integrity

Stale event payload snapshots can lead responding services to perform calculations or business logic based on outdated state, introducing potential data integrity issues across the ecosystem when out-of-sync events are processed.

Influences

Because data-based event payloads require a contract, architects might need to coordinate with other teams to create and maintain the event payload contract regarding:

- The contract type
- Whether the contract is strict or loose
- The required data
- The data types for the attributes in the contract
- The versioning strategy (for a strict contract)

These items become more significant as the systems' or services' *coupling distance* increases (the physical distance between them). If two services using a data-based event contract are maintained by the same team, the coupling distance is very short and event contracts are much easier to coordinate. However, if systems maintained by separate teams or departments are communicating using a data-based event contract, their coupling distance may be very long, making event contracts harder to coordinate.

Database topology also influences the choice between data-based and key-based event contracts. Fortunately, the Data-Based Event Contract pattern is very flexible about this: because responding services don't need to query a database, it really doesn't matter much whether the system topology is monolithic, domain data, or dedicated database per service.

How to Govern This Pattern

To measure the risk areas identified in the Cons section and validate whether Data-Based Event Contract is the right pattern to use, architects can leverage many different kinds of measurements and fitness functions, including:

Bandwidth usage

Measuring bandwidth usage in relation to unused fields helps architects tune contracts for optimal bandwidth usage and determine whether this is the right pattern for their particular context. If bandwidth usage is too high, the architect can consider using partial data, using direct (point-to-point) messaging with specific contracts rather than broadcast messaging, or switching to the Key-Based Event Contract pattern.

Contract change frequency

Measuring how frequently the underlying event contract changes can help validate whether this is the right pattern for a particular situation. Too much contract churn can indicate that a key-based event contract might be more suitable.

Data inconsistency

If data becomes inconsistent or multiple events get crossed with each other, the problem might be with having multiple systems of record. Measure these inconsistencies and weigh the benefits of using this pattern against its disadvantages.

Related Patterns, Antipatterns, and Pitfalls

The following patterns are related to the Data-Based Event Contract pattern:

Contextual Events pattern (see Chapter 12)

Contextual events are specific derived events that describe the outcomes of a service processing an incoming event. Architects must decide what type of payload contract contextual events should use. Most who apply the Contextual Events pattern choose to use key-based event contracts, supplementing the key with any necessary data based on the context of the derived event.

For example, suppose a `Credit Limit Check` service checks a cardholder's credit limit when they make a purchase. If the cardholder is close to their limit, this service might generate a `credit limit warning` event. That event's data payload might therefore include the credit card number as the key, the amount left, and the limit. This is an example of a partial data payload.

Extensible Events pattern (see Chapter 18)

In the Extensible Events pattern, a service triggers a derived event based on its processing outcome, even if no other services are responding to that event. It's a way of promoting architectural evolvability and overall agility. The tricky part is deciding what payload to send, particularly without knowing what data a future service might need. The Data-Based Event Contract pattern can help the Extensible Events pattern become more extensible by sending the entire data context in the payload (but keep bandwidth usage in mind).

Key-Based Event Contract pattern (see Chapter 23)

The Key-Based Event Contract pattern is the alternative to the Data-Based Event Contract pattern. Architects should understand their trade-offs when choosing between them.

Chapter 15. Domain-Broker Pattern

Having a *single* broker facilitating communication within an event-driven system is not unusual. While this setup might work well, it also couples the entire architecture to one broker! That reduces the scope of the entire architecture to one quantum (see "Architecture Quantum"). Fortunately, the Domain-Broker pattern gives architects the option of leveraging domain partitioning. Table 15-1 summarizes the metadata for the Domain-Broker pattern.

Table 15-1. Metadata for the Domain-Broker pattern

CATEGORY	METADATA
CONTEXT:	Event-Driven Architecture, Message-Based Architectures, Microservices
IMPROVED:	Availability, Domain Alignment, Elasticity, Fault Tolerance, Flexibility, Modularity, Scalability
DEGRADED:	Cost, Simplicity

How Many Brokers Should I Have in My Distributed Architecture?

Many event-driven[1] topologies consist of having just one broker. That's a great starting point for most systems and usually serves problem domains well.

Consider an online book retailer's event-driven architecture (EDA). A subtle aspect, often missed, is that the entire topology shown in Figure 15-1 represents a single architecture quantum! Without additional safeguards in place, if the broker is down, all processors in the workflow will error out. The selling point of an architecture style like EDA is high scalability *and* high throughput, so distilling the entire architecture down to one architecture quantum defeats one of EDA's superpowers.

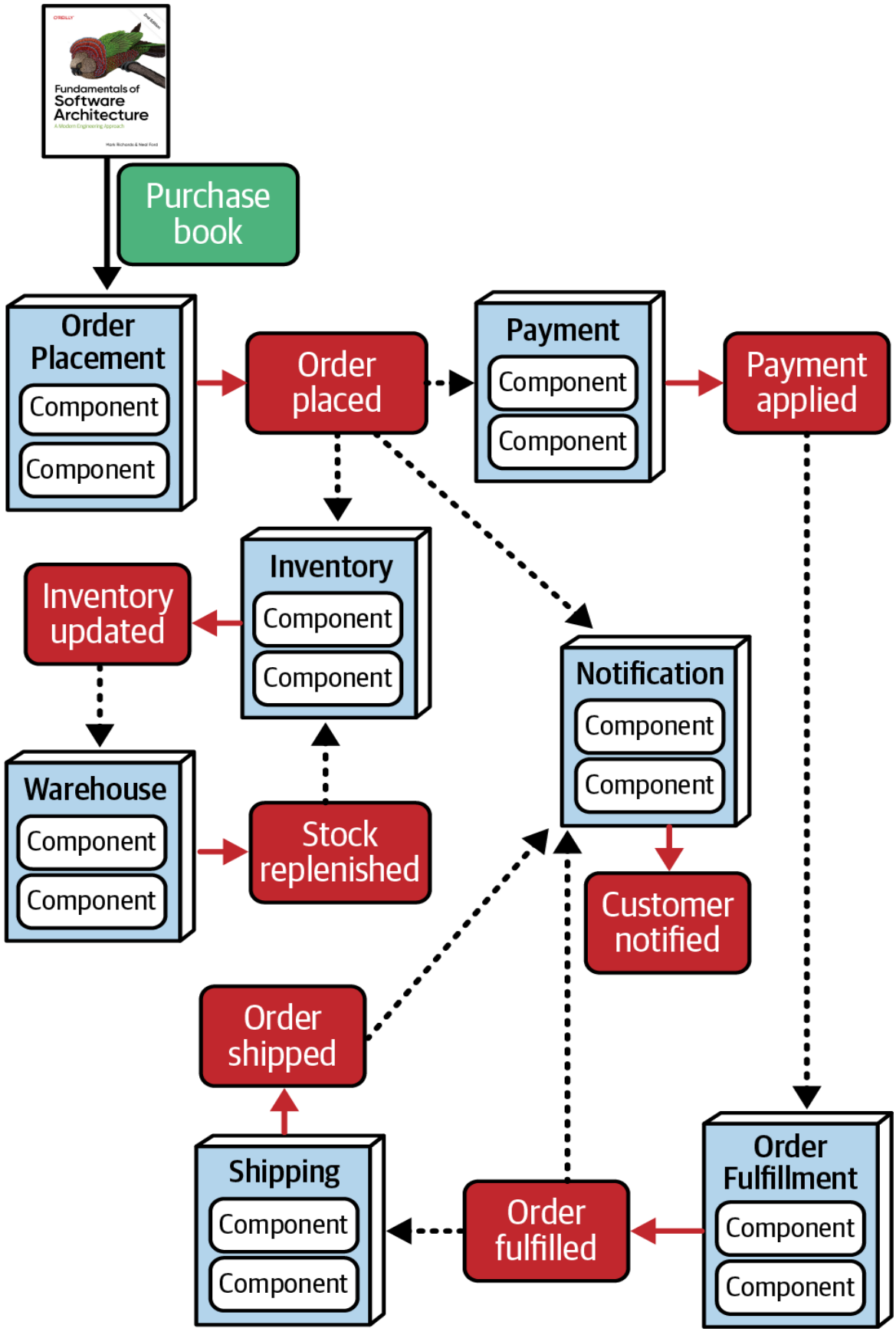

Fundamentals of Software Architecture
A Modern Engineering Approach
Purchase book
Order Placement
Component
Component
Order placed
Payment
Component
Component
Payment applied
Inventory
Component
Component
Inventory updated
Notification
Component
Component
Warehouse
Component
Component
Stock replenished
Customer notified
Order shipped
Shipping
Component
Component
Order fulfilled
Order Fulfillment
Component
Component

Figure 15-1. Sample retailer EDA topology

Applying the Pattern

The Domain-Broker pattern, as the name suggests, involves introducing additional brokers per *domain*. With this pattern, each domain gets its own broker, which helps on multiple levels. First and foremost, it breaks up the architectural quantum, since each domain now has its own broker.

A close look at the topology illustrated in Figure 15-1 reveals multiple domains, as shown in Figure 15-2. Some are customer-facing, dealing with order placement and payment. Order fulfillment and shipping make sure that customers receive their orders. Inventory and warehousing are all about keeping appropriate levels of stock on hand, and the notification domain is simply plumbing.

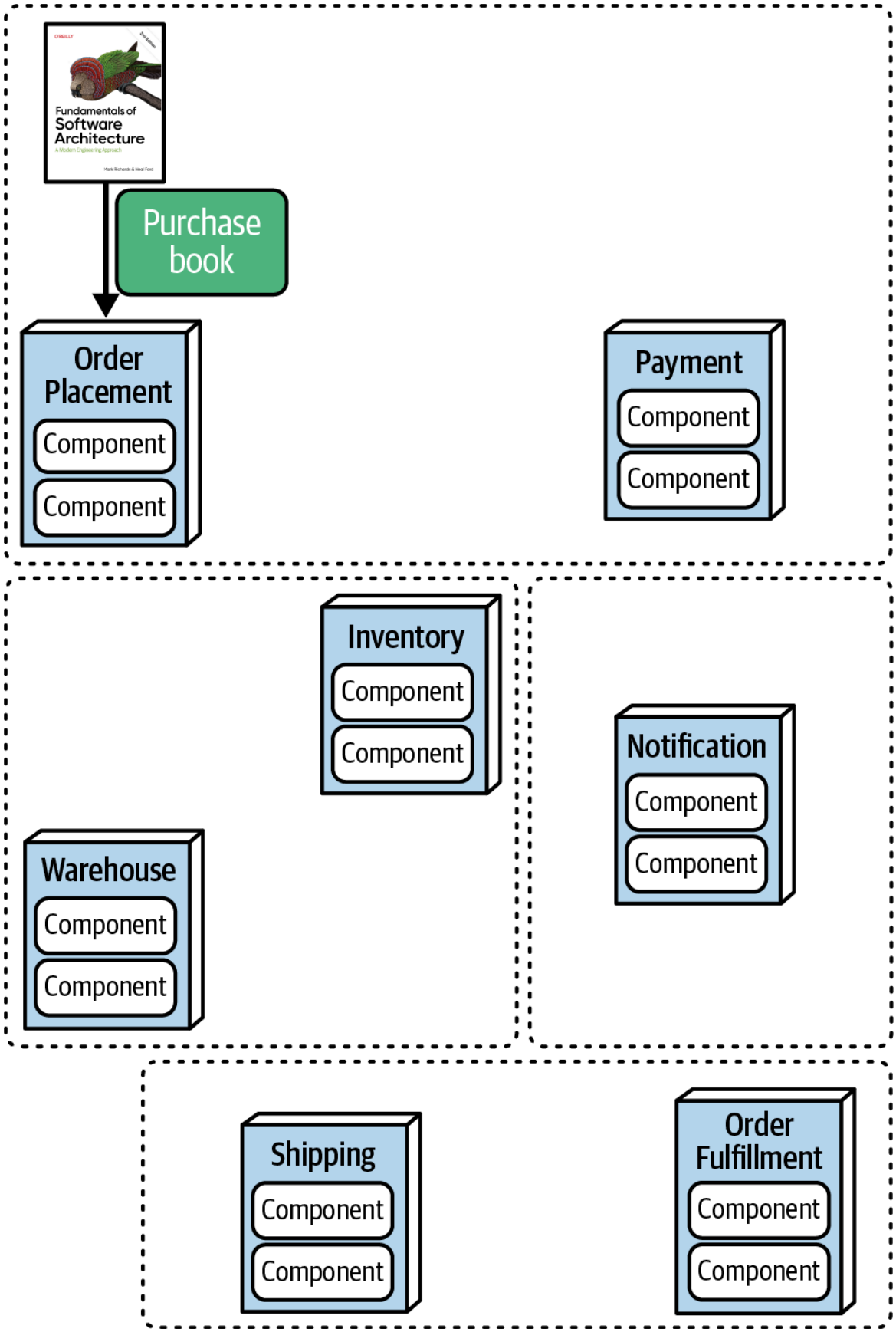

Fundamentals of Software Architecture
A Modern Engineering Approach
Purchase book
Order Placement
Component
Component
Payment
Component
Component
Inventory
Component
Component
Notification
Component
Component
Warehouse
Component
Component
Shipping
Component
Component
Order Fulfillment
Component
Component

Figure 15-2. Multiple domains revealed

With these domain boundaries in mind, the architect can now add a broker *per domain*, as demonstrated in Figure 15-3.

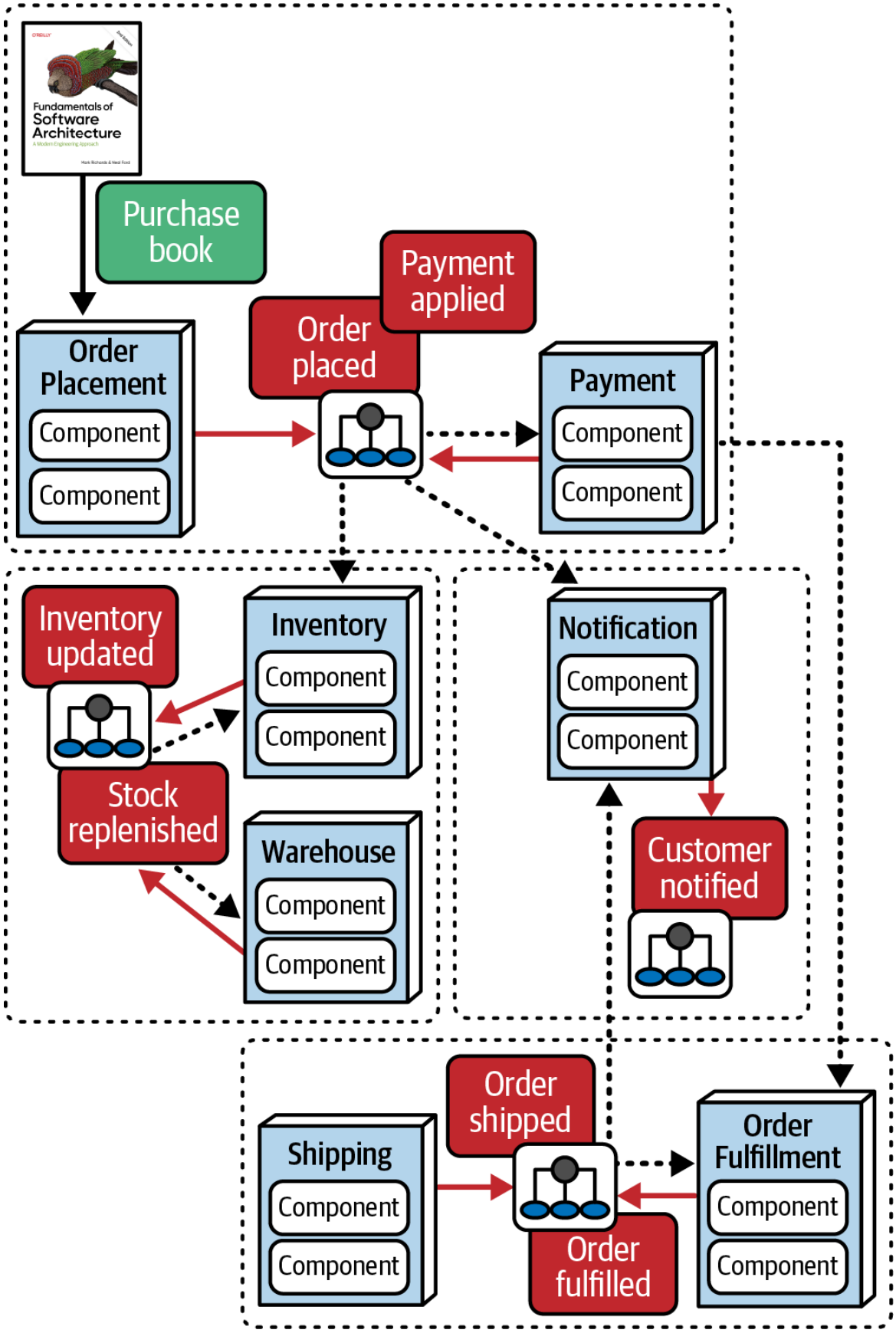

Fundamentals of Software Architecture
Purchase book
Order Placement
Component
Component
Order placed
Payment applied
Payment
Component
Component
Inventory updated
Inventory
Component
Component
Stock replenished
Warehouse
Component
Component
Notification
Component
Component
Customer notified
Shipping
Component
Component
Order shipped
Order fulfilled
Order Fulfillment
Component
Component

Figure 15-3. Broker per domain

Each domain, now served by its own broker, is an independent architecture quantum. This reduces the coupling between domains.

Implementing the Pattern

Moving from an existing single-broker setup to domain-based brokers requires architects to do significant analysis and planning. The first step in aligning brokers to specific domains is to identify the larger domains and subdomains at play and attempting to find their "edges." Domain-Driven Design (DDD) can be used to identify domains, but it requires a lot of organizational alignment and buy-in to be effective, so architects may have to reach for other tactics in some scenarios.

DDD approaches decomposition from a business perspective. However, as Table 15-1 shows, the Domain-Broker pattern also improves the architecture's technical and operational capabilities, which suggests alternative decomposition strategies. One is to look at traffic and data metric patterns, such as volume or payload size. Parts of the architecture may have certain retention and delivery guarantees that require a very specific kind of broker.

Operational service-level objectives (SLOs) are also a strong signal that it's a good idea to derive domains and separate the "money-making" parts of the system from its other parts. Consider fault tolerance needs. Customers can't place orders if both the `Order Placement` *and* `Payment` services are unavailable. Perhaps this reveals a "front-facing" or a "customer-facing domain." Separating these services into their own quantum ensures that even if other parts of the architecture are momentarily unavailable, customers can go about their business.

Architects can also use legal, compliance, and regulatory needs to identify certain boundaries. For example, if the system must comply with the Payment Card Industry Data Security Standard (PCI DSS), then using a single broker would put the entire architecture under audit. However, giving that piece of the architecture its own broker would keep the rest of the

architecture out of audit scope, simplifying things and saving time and money.

Once the architect identifies the domains, they should identify topics and event queues, assign them to specific domains, and assess their consumers. For example, in Figure 15-3, the broker serving `Order Placement` has multiple downstream consumers, both inside and outside its own domain.

At this point, the architect is ready to start planning the migration strategy, which will focus on "strangling" the singular broker out of the architecture. Figure 15-4 illustrates a common migration strategy.

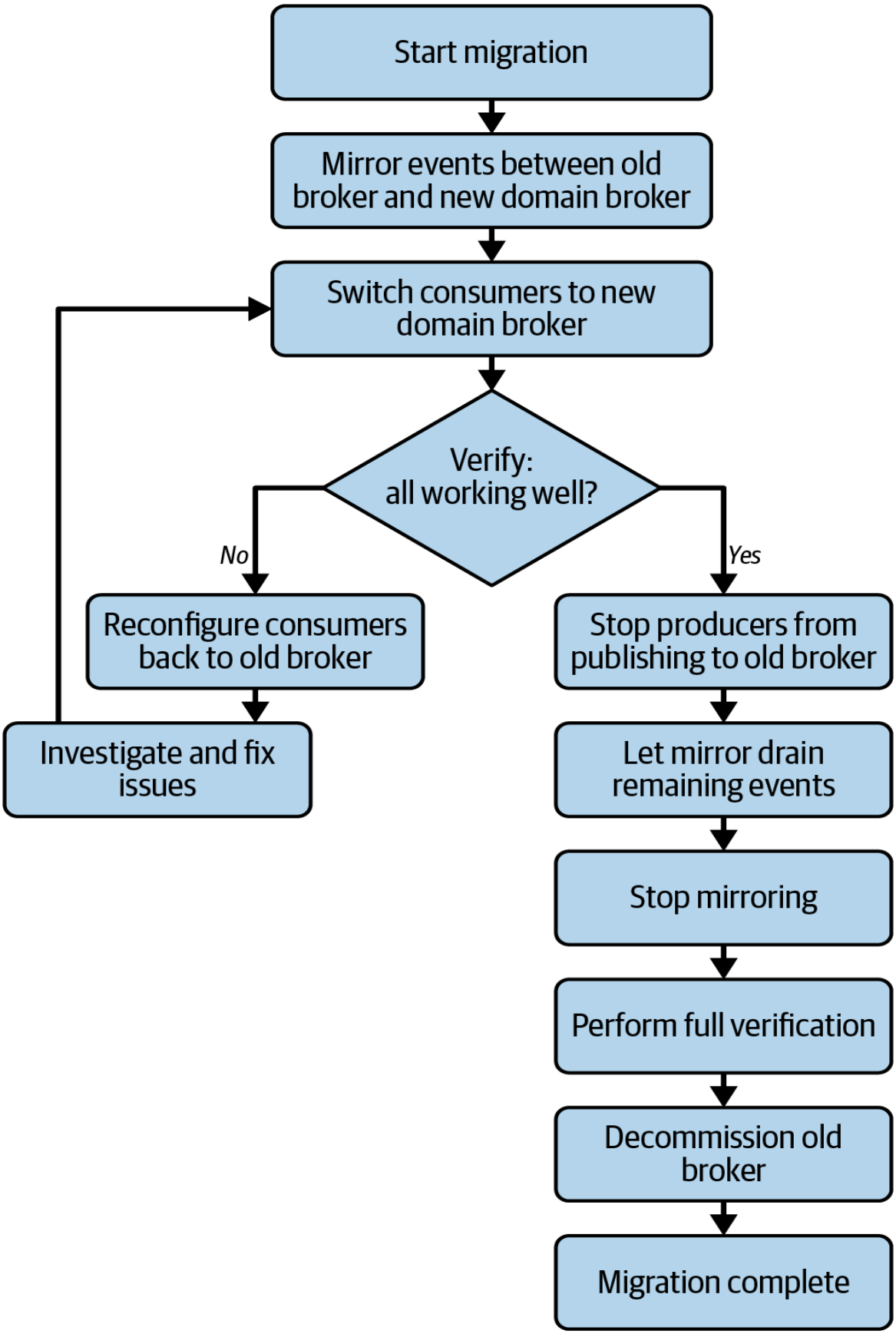

Start migration
Mirror events between old broker and new domain broker
Switch consumers to new domain broker
Verify: all working well?
No
Yes
Reconfigure consumers back to old broker
Investigate and fix issues
Stop producers from publishing to old broker
Let mirror drain remaining events
Stop mirroring
Perform full verification
Decommission old broker
Migration complete

Figure 15-4. Migration strategy to domain brokers

The first step here is to mirror events between the old broker and the new one. Then switch consumers over to the new domain broker. This is a good time to verify that all is still well. If something goes awry, teams can easily reconfigure the consumers to point back to the old broker while they figure out what happened. If all is well, then it's time to have producers stop publishing to the old broker and let the mirror drain any remaining events. Once the old broker's queue is empty, stop mirroring and perform a full verification. Once that's complete, the old broker can be decommissioned.

Trade-Offs

Recall that an *architecture quantum* is "the smallest part of the system that runs independently." To put it another way, by introducing domain brokers, the architect is introducing true *physical* boundaries between domains. Furthermore, introducing additional brokers helps to decouple the domains operationally, which brings additional benefits to the table.

Pros

The advantages of applying this pattern include:

Domain alignment

> The Domain-Broker pattern partitions the architecture by domain, as opposed to other patterns (like the Multi-Broker pattern, see Chapter 25) that partition it *technically*. Domain partitioning greatly helps with abstraction, for example, by keeping everything together in one place.

Availability

> Having just one broker means having a single point of failure, so decoupling multiple domains by supplying each one with its own broker counters that problem and limits the blast radius of potential failures. It also greatly increases the availability of the entire system, since other domains can continue to function even if one domain's broker is

unavailable. In the situation depicted in Figure 15-3, customers can still browse and buy books even if fulfillment isn't currently possible.

Scalability

With a single broker, one high-volume domain can cause the other domains to "starve." Using separate domain brokers, domains can scale independently, which improves scalability. Furthermore, if portions of the architecture are susceptible to bursty or spiky traffic patterns (perhaps `Order Placement` on the day a popular author launches a new book), that specific broker can scale as needed.

Security and compliance isolation

If a certain domain has a different security, compliance, or risk posture, separating messages from that domain to its own dedicated broker allows architects to adjust its security and access controls. This aids in ensuring compliance while keeping the rest of the architecture outside the scope of audits.

Cons

The disadvantages of applying this pattern include:

Cost

Licensing costs for message brokers will likely increase when adopting this pattern, since more broker instances are needed. More operational overhead results from running and managing multiple brokers. Architects must do a careful cost-benefit analysis to ensure that this trade-off is justified.

Simplicity

Adding message brokers tends to complicate the system, particularly on the operational side — and systems evolve. To which quanta, if any, do new processors belong? Adopting this pattern means thinking through

this recurring question over and over. Each processor may also have to subscribe to multiple brokers, depending on its role in the workflow.

Comparison Chart

The following chart (Table 15-2) summarizes the trade-offs associated with this pattern. "Without Pattern" refers to using a single broker, whereas "With Pattern" means using the Domain-Broker pattern.

Table 15-2. Comparison chart for the Domain-Broker pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Availability	★	★★★
Domain Alignment	★	★★★★★
Elasticity	★	★★★★
Fault Tolerance	★	★★★
Flexibility	★	★★★★★
Modularity	★	★★★★
Scalability	★	★★★★
DEGRADED		
Cost	$	$$$
Simplicity	★★★★	★★

The following sections describe each of the capabilities found in Table 15-2.

Improved

The capabilities that are improved when applying this pattern include:

Availability

Partitioning by domain introduces isolation and reduces the architecture's reliance on the availability of a single broker. This is simply a desirable side effect.

Domain Alignment

Assigning brokers by domain is essentially partitioning by domain, so it significantly improves the domain abstraction of the entire system. This is the Domain-Broker pattern's *primary* benefit.

Elasticity

Separate brokers can be tuned specifically to the needs of their domains. Breaking the architecture quantum into smaller units allows teams to better tune brokers for domains with bursty traffic to accommodate spikes without overwhelming the whole architecture.

Fault Tolerance

Having a single architecture quantum centered around a single broker introduces a single point of failure. A fault in the broker halts the entire workflow. Partitioning brokers by domain better contains failures, so unaffected domains can continue processing. Even though each individual domain broker is still a critical dependency for that domain, the system's fault tolerance is improved.

Flexibility

Having a broker per domain lets architects choose the solution that best supports their specific needs around throughput, retention, delivery guarantees, and operational characteristics. For example, for a mission-critical domain, an architect might choose a high-performance cluster like Kafka, as opposed to a much lighter-weight broker for, say, the Notification domain.

Modularity

This pattern improves modularity by reinforcing explicit domain boundaries in the messaging layer. Processors within a bounded context interact primarily with their domain broker, reducing cross-domain entanglement.

Scalability

Much like for elasticity, this pattern enables scaling at the domain level instead of treating the entire architecture as one scaling unit. High-throughput domains can scale their broker clusters and consumers independently, while low-volume domains remain right-sized.

Degraded

The capabilities that are degraded when applying this pattern include:

Cost

The licensing costs associated with the message broker will likely increase when adopting this pattern, because more instances are created. Architects should perform a full cost-benefit analysis.

Simplicity

As noted above, adding message brokers complicates the system, particularly as it changes and evolves. For every additional processor, architects must decide which quantum it belongs to and consider integrations for downstream processors that might need its notifications. For example, downstream processors like `Notification` (in Figure 15-3) must now connect with multiple upstream brokers, increasing the complexity of configuration management.

Influences

This pattern is influenced by the architecture quantum and influences workflow state.

Architecture Quantum

As discussed in "Architecture Quantum", architects must consider different aspects of coupling and their effects on architectural scope:

Static coupling

> *Static coupling* refers to the dependencies required for a processor or system to function independently. The Domain-Broker pattern can significantly reduce the architectural scope of a system by narrowing it for each domain. However, the static dependency of the domain-specific broker remains, since without the brokers, the individual processors in the system can't function.

Dynamic coupling

> *Dynamic coupling* describes the runtime communication between services. The Domain-Broker pattern doesn't change how services communicate dynamically; they still publish and subscribe to events. The only difference is which broker they connect to.

Temporal coupling

> *Temporal coupling*, which deals with time-based dependencies, doesn't really apply to the Domain-Broker pattern.

Contract coupling

> *Contract coupling* relates to the contracts shared between producers and consumers. This pattern does not influence contract coupling: contracts remain the same whether the system uses a single broker or domain-specific brokers.

Transactional coupling

Transactional coupling refers to data consistency and ACID capabilities across service boundaries. The Domain-Broker pattern doesn't introduce new transactional concerns — services still face the same eventual consistency challenges whether they use one broker or multiple. However, coordinating distributed transactions across domain boundaries may become slightly more complex when workflows span multiple brokers.

Workflow State

If the system needs to track which services have executed their step in a workflow, what the final result was, or if any errors were encountered, having multiple brokers in the mix makes this a lot harder. Consider Figure 15-3 once again, which illustrates a broker per domain across the customer, order, fulfillment, and inventory domains. Events are now scattered across the domains, so finding the status of a specific order would mean stitching together that particular workflow's path across the architecture by tracing events across each one of those brokers, using techniques like distributed tracing or correlation IDs.

Along the same lines, this pattern can also influence the implementation of other patterns, specifically Explicit Workflow State (see Chapter 17). This pattern introduces a *workflow-state owner*: a service or processor that acts as a place to query for the state of a workflow. With the introduction of multiple domain-specific brokers, this too becomes a bit more challenging. Not only does the workflow-state owner have to connect to multiple brokers, any updates to the architecture now need to account for this requirement.

How to Govern This Pattern

The primary focus of this pattern is domain partitioning, so the focus for governing it is preventing domain drift.

Governing domain boundaries within architectures is hard. It can be helpful to use fitness functions to test that each processor is only emitting events to its domain-specific broker. To further reinforce these boundaries, teams can set up observability around each broker to ensure that it only receives events originating from processors within its bounded context.

When new processors are introduced, their broker placement should be justified in terms of domain responsibility rather than convenience or short-term operational needs. This might require governance in the form of architecture decision records (ADRs) to codify the decision and get stakeholder sign-off before implementation.

Related Patterns, Antipatterns, and Pitfalls

The following pattern is related to the Domain-Broker pattern:

Multi-Broker pattern (see Chapter 25)

> The Multi-Broker pattern can improve message throughput and capacity through technical partitioning, whereas the Domain-Broker pattern partitions by business domain. It's not unusual to combine the Multi-Broker and Domain-Broker patterns when a domain-specific broker becomes a performance bottleneck. In this case, architects first discover the architecture quanta in place and introduce domain brokers. They can then use performance testing and observability to identify which domains need higher throughput, and apply the Multi-Broker pattern to technically partition those specific domain brokers.

---

1 We'll refer to "messages" and "events" interchangeably here and throughout the chapter.

Chapter 16. Event Sourcing Pattern

Traditionally, writing to a database is destructive — updates to the data destroy what was there previously. That is, we lose the journey that got us to a certain state. The Event Sourcing pattern captures significant "events" that represent domain or business constructs in an immutable event log. These events typically represent higher-level business concerns, rather than low-level actions that the system performs. This log can be replayed to reconstruct the state of the system at any point in time, preserving complete history. Table 16-1 summarizes the metadata for the Event Sourcing pattern.

Table 16-1. Metadata for the Event Sourcing pattern

CATEGORY	METADATA
CONTEXT:	Data, Monolithic Architectures, Microservices
IMPROVED:	Auditability, Debuggability, Testability
DEGRADED:	Cost, Data Consistency, Data Integrity, Simplicity

Should I Capture My Data as an Event Stream?

Consider a system that tracks customers and their shipping addresses. When a customer updates their shipping address, the system simply destroys the previous address, creates a new address, and associates it with the customer. In this workflow, the fact that the customer has moved is lost forever!

However, it may be important to track not just the current state but also the series of events that led to that state — perhaps for regulatory reasons, to make the system auditable, or simply for convenience of usage. In these cases, the destructive way of using datastores does not work. Simply recording the state of events can be beneficial for other use cases, too — such as trend analysis, understanding user journeys, machine learning, and debuggability.

One way to track how the system arrived at a certain state is to introduce a mechanism that records its "history" — for example, a log or a history table. The idea is simple — this table reflects the initial state of an entity and how it has changed over time. With this in hand, teams can see the system's state at any point in time. Take the example of the customer who moved. If the system retained their old *and* new shipping addresses, it would be clear that the customer has moved. Let's say the customer moved between states or even countries — and a previously shipped item is recalled due to safety issues. The customer is due a refund. Knowing where the customer lived when that item was shipped might help calculate what taxes or shipping costs should be withheld from the refund.

Another benefit of having a separate history table is that the operational aspects of the system — the "create, read, update, delete" (CRUD) actions — are not impacted by the responsiveness requirements of analytical queries, trend analysis, or machine-learning workloads that process historical data. This separation of concerns allows the primary operational datastore to remain optimized for transactional workloads, while the history table can be structured and indexed specifically for analytical access patterns.

However, this approach does have its own set of trade-offs. When the system stores the current state and *all* past states, there will be duplication within the datastore, and these always need to be in sync. Let's say that a single application is responsible for writing to both the log and the primary datastore. One write could fail while the other succeeds. So which one is the source of truth? Moreover, if it's possible to re-create the application's past

states by replaying the history table, why maintain a separate copy of the final state?

Another trade-off lies with the mechanics of capturing history. Most off-the-shelf database systems "automagically" capture and track history using triggers and temporal tables. While using these *may* work to capture the system's state as it transitions, they can't provide a clear picture of exactly what happened. For example, a customer might transfer money from one account to another. The trigger would capture this as two independent operations: one to deduct from the source account, and the other to add to the target account. However, this loses the fact these two operations are part of a larger business concern — that the customer wished to *transfer* money from one account to another! So, while audit logs capture the "what" and "who" of changes, it does not capture the "why."

Finally, consider fidelity. There might be events of interest that aren't always captured at the datastore level. For example, a customer might browse several lists of similar products and add all of them to the cart, but then delete all but one. While the item they actually purchase is recorded in the primary datastore, the business may well be interested in these multiple selections and the customer's vacillating behavior pattern — but that information is lost without another mechanism explicitly designed to capture it.

To summarize: while audit or history tables provide some level of "replayability" in the system, significant duplication occurs at the data-storage level, the tables don't explain the "why," and they often don't offer the right level of visibility.

The Event Sourcing pattern helps with all of these concerns by treating "events" as a source of truth that can be replayed to arrive at the current state.

Applying the Pattern

When a system is using the Event Sourcing pattern, it captures and publishes changes to the state of an object by using another object: an event object. The *event object* contains not just the next state of the entity, but the "what" and the "why." In other words, the event object attempts to provide the business context necessary to explain and justify what has happened.

This event object is stored in an immutable, append-only log that captures all the event objects related to that system in chronological order. This log becomes the source of record for every event emitted within the system, as well as the source of truth for the state of any particular object.

In order to re-create an entity's current state (or its state at any given point in time), the system simply takes all the event objects associated with that entity and replays them back from start to end.

Event Sourcing is a complex pattern with a lot of moving parts. To avoid confusion, Table 16-2 identifies the different players and their roles. This table assumes the object-oriented programming (OOP) paradigm and uses terminology common in OOP implementations, though the underlying concepts also apply to functional programming approaches. Feel free to refer back to Table 16-2 if a term seems unfamiliar.

Table 16-2. Event sourcing components and responsibilities

Term	What it is	Typically implemented by	Res
Aggregate	A domain model component (represented as a class) with business invariants	Application team	Rep valic rule ever
Command	A request to change state (data object)	Application team	Carr inter `ddLi`
Command Handler	Application-layer component (a class or method)	Application team	Loa beha aggr emit
Domain Event	Immutable record of a significant business occurrence (data object)	Application team	Cap happ serv of tr
Repository	Persistence abstraction (represented as a class)	Application team (or framework adapter)	Loa aggr strea

Term	What it is	Typically implemented by	Res
Event Store	Database/product/infrastructure capability	Platform team/vendor	Imm only with orde chec
Projection/Read Model	Query-optimized view/store	Application team and infrastructure	Serv effic asyn

Identifying Events and Aggregates

An architect's first concern when implementing the Event Sourcing pattern is identifying which entities' state changes need to be tracked. One approach to this task is to use Domain-Driven Design (DDD). DDD introduces the idea of an "aggregate" which, as listed in Table 16-2, is a class that represents a collection of other DDD constructs (entities and value objects) that "belong" together and should all be treated as a single unit. Aggregates are also responsible for generating *domain events*, which represent significant changes to the state of the aggregate that are of interest to the system.

DDD also imposes several constraints on how aggregates are designed and behave. First, the aggregate *is* the interface (also known as the *root entity*) that other aggregates use to change its state. Second, the aggregate as a whole includes codified business *invariants* — business rules that ensure that the state of the aggregate as a whole, as well as the states of all the other objects it contains, is always correct and consistent. To achieve consistency, architects must mandate that any change to an aggregate must

be made within a transaction boundary. This ensures that the overall state of that aggregate and any objects within it is always consistent.

To contrast this, perhaps a change in one aggregate causes another aggregate to change. These changes can be (and typically are) recorded in *separate* transactions. Consider the example of an `Invoice` aggregate containing a list of `LineItem` entities — all of which, summed up, represent the total cost of the invoice. However, aggregates *do not* create or delete a `LineItem` directly. Instead, they simply invoke `addLineItem` on the `Invoice` aggregate, which expects a description of rendered services, hours worked, and an hourly rate. The `Invoice` aggregate first applies its business invariants — perhaps the description needs to be filled in, and the number of hours worked and hourly rate can't be negative. Next, it creates a `LineItem` and associates it with the existing `LineItem` list.

The `Invoice` aggregate may also offer users the ability to `approve` or `reject` the whole invoice. Once again, certain invariants may be applied — for example, at least one `LineItem` must be associated with the invoice before it can be approved. When invoked, the `approve` method changes the state of the `Invoice` to `approved`. It also loops through every associated `LineItem`, changing each one's state to `approved` as well.

Domain Events

Domain events are the star of the Event Sourcing show. They're immutable records of the changes that an aggregate undergoes. They enable communication between aggregates and bounded contexts, while encouraging loose couplings. DDD expects events to be identified and emitted when an aggregate undergoes specific changes.

Continuing with the `Invoice` example, there might be several events that represent significant business changes — for example, `InvoiceCreated`, `InvoiceLineItemAdded`, `InvoicePaid`, and `InvoiceCancelled`. (Note that they're all named in the past tense.) Each domain event contains all the necessary data to sufficiently describe

what happened. For example, `InvoiceCreated` might have a unique `id`, the name of the vendor submitting the invoice, `issueDate`, `createdBy` and a `timeStamp`. For each `LineItem` added, there might be another event, say `InvoiceLineItemAdded`, describing the line item and the invoice to which it was added — the invoice `id`, `description`, `hoursWorked`, `hourlyRate`, and other metadata.

When implementing the Event Sourcing pattern, these events eventually go on to form the event log. Teams can re-create the state of any invoice by laying these events out chronologically and then replaying them on top of one another. Consider the following events generated by the `Invoice` aggregate, in chronological order: `InvoiceCreated` → `InvoiceLineItemAdded` → `InvoiceLineItemAdded` → `InvoicePaid`.

Event Store Structure

Once the events have been identified, the next step is to establish the *event store structure*. In the Event Sourcing pattern, the event log must be immutable and append-only. Recall that events are *historical* records — named for their immutable nature. The event store is simply providing the means to enforce that at rest.

At a minimum, each event must capture some standard fields, like aggregate ID and type, sequence number, event type, and event version (see "Schema Evolution"). These are all necessary to re-create the aggregate's state, but also help with other optimizations (see "Reading Events"). Some event stores even offer a global sequencing mechanism, which allows for total ordering across *all* events and event streams. This allows teams to stitch together everything that happened within a specific time window, which helps with understanding the system's behavior as a whole.

Typically, event stores are partitioned into separate streams by aggregate type, which can help with guaranteeing strict chronological order. Of course, architects may reach for other partitioning strategies, depending on the capabilities they're pursuing, like high scalability or operational ease.

Within each stream, the event store should guarantee *monotonicity*. Each event stored is assigned a stream-specific sequence number (like an event ID) in strictly increasing order. At scale, it's distinctly possible for the same aggregate to emit multiple events in succession that reach the event store at the same time. To counter this, the event store must offer concurrency controls (typically optimistic concurrency) to avoid persisting two events due for the same aggregate stream with the same ID (see "Handling Concurrency").

Reading Events

The event store must also exhibit several characteristics on the read side. (We discuss how events are written to the event store in "Writing Events".) Example 16-1 shows a sample event stream for the `Invoice` aggregate.

Example 16-1. Sample event stream for an `Invoice` aggregate

```
| aggregate_id | sequence | event_type     | payload
|
| ------------ | -------- | -------------- | ----------------------
-------- |
| INV-124      | 1        | InvoiceCreated | {vendor: "Wayne
Industries"}   |
| INV-124      | 2        | LineItemAdded  | { "description":
"Consulting", |
|              |          |                |   "hours": 10.00,
|
|              |          |                |   "hourlyRate": 200.00
}       |
| INV-124      | 3        | LineItemAdded  | { "description":
"Consulting", |
|              |          |                |   "hours": 8.00,
|
|              |          |                |   "hourlyRate": 200.00
}       |
| INV-124      | 4        | InvoicePaid    | {paid_date: ...}
|
| INV-125      | 1        | InvoiceCreated | {vendor: "Stark
Industries"}   |
```

As Example 16-1 shows, each entry for a particular instance of an aggregate (`Invoice`, in this case) can be identified by its `aggregate_id`. The

`event_type` identifies the specific domain event that occurred — what *type* of change happened to the aggregate. For example, `InvoiceCreated`, `LineItemAdded`, and `InvoicePaid` are all different event types that represent distinct business operations performed by the `Invoice` aggregate. The `sequence` number provides strict chronological ordering within each aggregate stream. To reconstruct the aggregate state, the system can simply replay this list of events sequentially.

This means the event store must have a mechanism to read all the events for a particular instance of a particular kind of aggregate. It should be able to isolate the aggregate-specific stream and provide the stored events in the right order so that the team can rehydrate the entity's state at that point.

Reading all the events associated with an entity all the time can get compute-intensive, time-consuming, and expensive. The architect's choice of event store should thus offer a way to create materialized views or projections, so that an entity's state need not be re-created from scratch all the time. To optimize querying, architects can create multiple projections from the same event stream. For example, reconsider the set of events in Example 16-1. From this table, multiple projections can be created, as shown in Example 16-2.

Example 16-2. Projections that optimize querying for invoices

```
# current invoice state
| invoice_id | vendor           | status | total | created_date |
| ---------- | ---------------- | ------ | ----- | ------------ |
| INV-124    | Wayne Industries | PAID   | 2000  | 2026-01-15   |
| INV-125    | Stark Industries | UNPAID | 1600  | 2026-01-20   |

# revenue by vendor
| vendor           | month   | total_revenue | invoice_count |
| ---------------- | ------- | ------------- | ------------- |
| Wayne Industries | 2026-01 | 12,500        | 42            |
| Stark Industries | 2026-01 | 8,300         | 28            |
```

Event stores might also support the notion of "snapshots" — a point-in-time representation of the state of an aggregate after replaying all events up to a certain sequence number, which is then persisted separately. In order to get the *current* state of the aggregate, only events *after* the snapshot was

created need to be replayed. This avoids having to read the entire event stream, significantly improving read throughput.

Snapshot management needs its own strategy and comes with its own set of trade-offs, and architects need to think this through. For example, should snapshots for aggregates be time-based, perhaps as a nightly job that takes a snapshot of all aggregates? While this is easy to do, it also increases the cost of storing snapshots. Or should snapshots for aggregates be on demand, with the system only taking snapshots of aggregate instances that are actually being read? This saves on costs, but the trade-off is read responsiveness.

Writing Events

Consider the invoicing application once again. A typical user journey would look something like this:

1. The user logs in to the system and pulls up an existing invoice — let's say it's invoice `INV-100`.
2. The user adds a new line item for some work done.
3. The system does some validation to ensure that no business invariants are being violated.
4. The system saves the new state of the `Invoice` aggregate and (perhaps) returns a success message to the user.

While this workflow may not be unusual for any CRUD-like application, it's significantly different for an application that leverages Event Sourcing.

To start, in the Event Sourcing pattern, the request for invoice ID `INV-100` would trigger a repository to fetch all events of aggregate type `Invoice` with the ID `INV-100` from the event store. The repository then creates a new invoice instance and hydrates the current state by replaying all the events in chronological order. This invoice is then sent back as the response. This sequence of actions is listed in Figure 16-1.

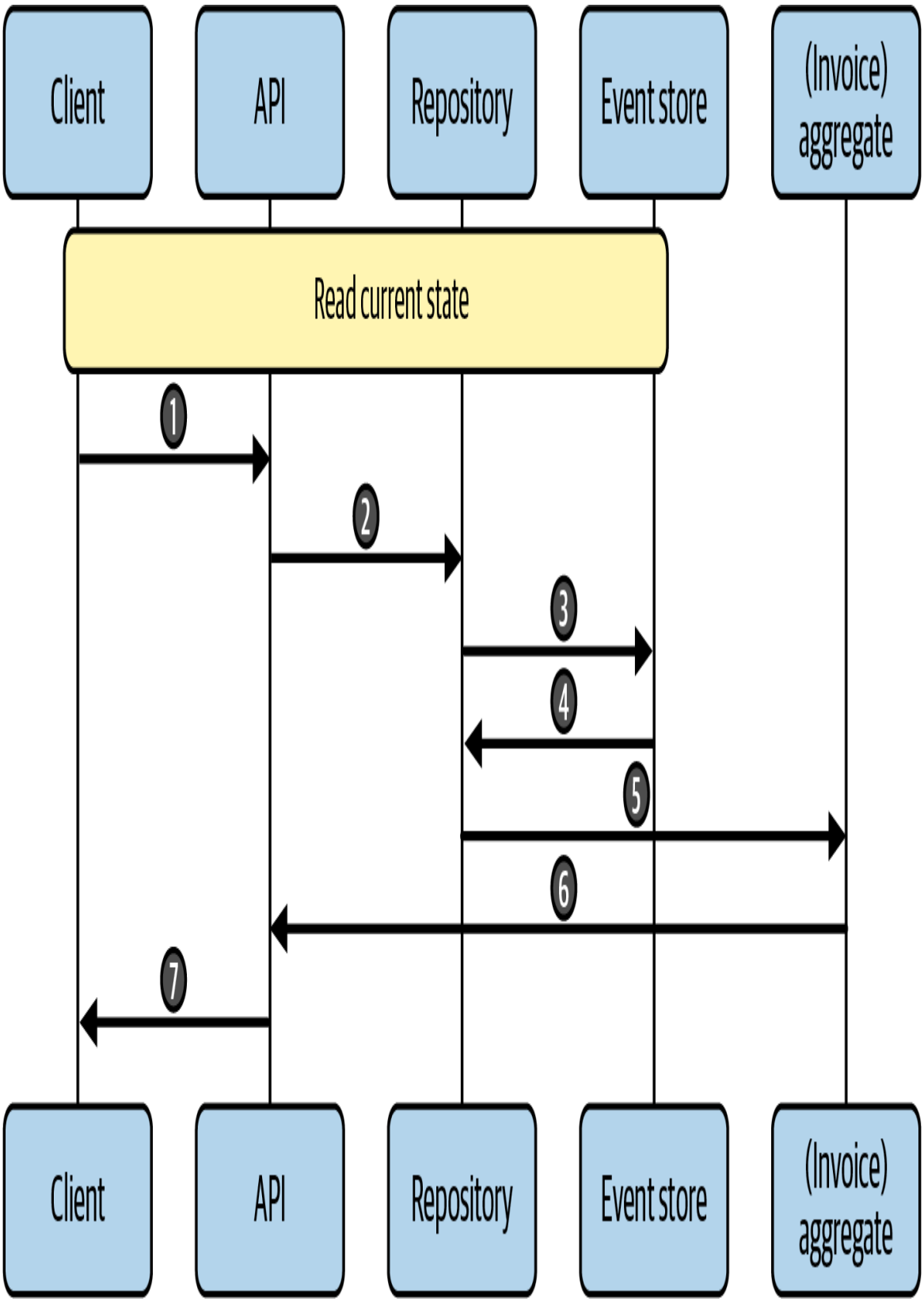


Figure 16-1. Sequence diagram for reading an `Invoice` aggregate

As depicted in Figure 16-1, the sequence of events for reading is as follows:

❶ `GET /invoices/INV-100`
❷ Load `Invoice INV-100`
❸ Fetch stream (`Invoice`, `INV-100`)
❹ Return events in order
❺ Replay events to hydrate state
❻ Current invoice state
❼ 200 OK + state + sequence

This returns a fully hydrated `Invoice` aggregate object to the client. Example 16-3 shows a sample response for `Invoice INV-100`.

Example 16-3. Sample response for fetching invoice `INV-100`

```
// GET /invoices/INV-100
{
  "aggregate_id": "INV-100",
  "status": "DRAFT",
  "items": [
    {
      "description": "Consulting",
      "hours": 10.00,
      "hourlyRate": 200.00
    }
  ],
  "total": 2000.00,
  "sequence": 5  // Current state of the stream
}
```

Note that the returned invoice contains the last "sequence" number — that is, the count of *facts* or events that were replayed to hydrate the aggregate.

The user's browser receives this response detailing the current invoice state. Now, suppose the user adds a new line item for 6 hours of consulting at $200 an hour. This is sent as a `POST` request with a body, as shown in Example 16-4.

Example 16-4. Sample request to add a new line item to invoice `INV-100`

```
// POST /commands/add-line-item
{
  "invoiceId": "INV-100",
  "description": "Consulting",
```

```
  "hours": 6.00,
  "hourlyRate": 200.00,
  "expectedVersion": 5 // "This is the version I think the invoice
is at"
}
```

This time, the response contains an `expectedVersion` number — the version the system *thinks* the `Invoice` ought to be at. This helps with concurrency issues, as we describe in "Handling Concurrency".

The JSON `POST`ed in Example 16-4 triggers the system's write sequence, which is illustrated in Figure 16-2.

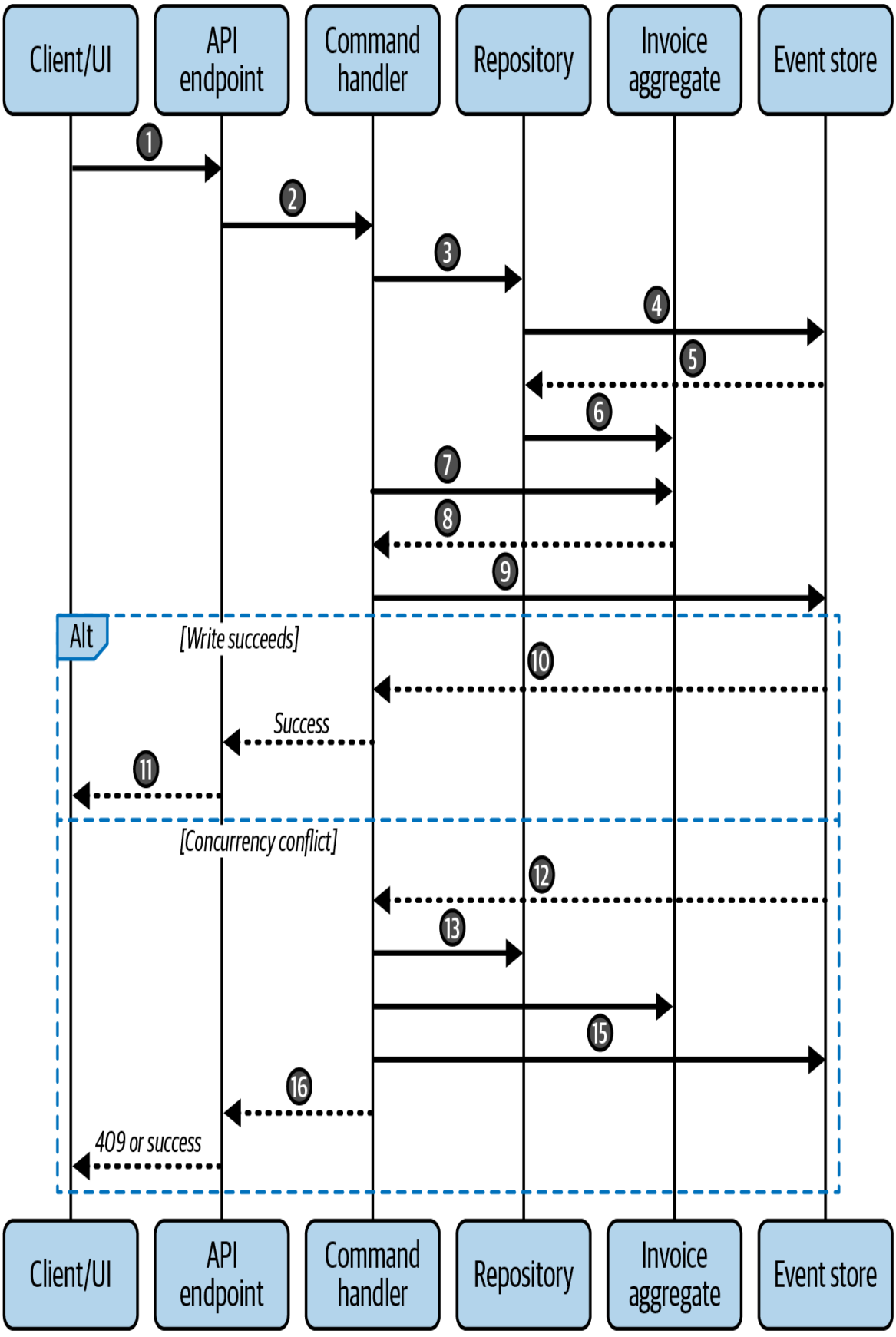

Client/UI
API endpoint
Command handler
Repository
Invoice aggregate
Event store
1
2
3
4
5
6
7
8
9
Alt
[Write succeeds]
10
Success
11
[Concurrency conflict]
12
13
15
16
409 or success
Client/UI
API endpoint
Command handler
Repository
Invoice aggregate
Event store

Figure 16-2. Sequence diagram for adding a line item

As depicted in Figure 16-2, the sequence of events for reading is as follows:

1. `POST /commands/add-line-item`
2. `handle(AddLineItemCommand)`
3. `load(invoiceId=INV-100)`
4. `readStream(Invoice, INV-100)`
5. Returns all the events for `Invoice INV-100`
6. Rehydrate aggregate from events
7. `Invoice.addLineItem(description, hours, hourlyRate)`
8. `LineItemAdded` (uncommitted)
9. `append(events, expectedVersion=5)` to event store
10. Append OK (sequence=6)
11. New version (`HTTP 200`)
12. `ConcurrencyException`
13. Reload + rehydrate
14. Reevaluate command
15. Retry append with new `expectedVersion`
16. Success OR conflict error

The application receives the `POST` request, typically in the form of a command like `AddLineItemCommand`. As Example 16-4 shows, the `AddLineItemCommand` command contains an `expectedVersion` field. This is the client's way of telling the system that, when it attempts to persist the *new* state of the invoice, it must ensure that no other changes have been made to the invoice since version 5. This is an optimistic concurrency-control mechanism to prevent updates from getting lost, in case multiple updates are made to the same invoice concurrently.

This request will eventually be handled by a "command handler" (see Table 16-2) that is specific to the `Invoice` aggregate, which performs command handling, validation, event generation, and persistence. Let's dive into the details of the workflow.

Command handler

The application receives commands from users or external systems that express the *intent* to change an aggregate's state. Unlike *events*, which describe what happened in the past, *commands* are requests for changes and may be rejected. For example, commands like `AddLineItemCommand`, `ApproveInvoiceCommand`, and `CancelInvoiceCommand` are processed by the command handler.

As shown in Figure 16-2, the command handler is central to the write event workflow. Specifically, it performs the following steps:

1. Loading the current state of the aggregate by reading and then replaying its event stream
2. Invoking the appropriate method on the aggregate, such as `invoice.addLineItem(...)`
3. Catching any business-rule violations thrown by the aggregate
4. Persisting the resulting events to the event store
5. Responding with an "OK" status back to the invocation API if successful, or attempting to retry in case of an error

Validation

Before an event is generated, the aggregate validates that the command respects all business invariants. As we suggested earlier, the `Invoice` aggregate might check to see whether the line item has a description and that the values for hours worked and hourly rate are not negative.

If this validation fails, the aggregate throws an exception and no event is generated or persisted.

Event generation

When a command successfully passes validation, the aggregate generates one or more domain events that describe what just happened. These events should contain:

- The aggregate ID

- A timestamp
- All data necessary to understand the change (the "what")
- Business context explaining the change (the "why")

The aggregate typically maintains an internal list of uncommitted events that were generated during the current operation.

Persistence

Finally, the command handler persists the aggregate and generated domain events to the event store in a single atomic operation. The event store assigns each event a monotonically increasing sequence number and appends it to the aggregate's event stream.

If the write succeeds, the events are considered committed. Optionally, they may be published to other parts of the system using a message broker. While this isn't strictly a part of the Event Sourcing pattern, publishing events allows downstream systems and services to react to state changes, enabling loosely coupled event-driven architectures. It should be handled carefully to ensure reliable delivery and idempotency of event consumers.

If the write fails due to a concurrency conflict, the command handler attempts a retry, which is detailed in the next section.

Handling Concurrency

"Event Store Structure" discussed the event store's mechanism for concurrency controls to ensure strict ordering of events. On the application side, architects may have to implement additional mechanisms to avoid and reconcile any conflicts. For example, what should happen if the event store receives two events for the same aggregate with the same sequence number? Perhaps the architect routes all events from the same kind of aggregate to the same processor to avoid conflicts with sequence numbers. This would affect scalability and throughput, but would simplify the concurrency concerns. Alternatively, if the event store were to throw an exception upon receiving these two conflicting events, the command

handler could attempt to reload the event stream and produce a *new* event that accurately captures the state of the aggregate. Consider Example 16-5, which shows the event stream for an `Invoice` aggregate with the `id` `INV-125`.

Example 16-5. Initial event stream for `Invoice INV-125`

```
| aggregate_id | sequence | event_type     | payload
|
| ------------ | -------- | -------------- | ----------------------
---------------- |
| INV-125      | 1        | InvoiceCreated | {vendor: "Stark
Industries"}            |
| INV-125      | 2        | LineItemAdded  | {item_id: "S1",
amount: 100}            |
| INV-125      | 3        | LineItemAdded  | {item_id: "S2",
amount: 200}            |
```

If we were to map the events listed in Example 16-5 in chronological order (that is, by sequence number), we'd arrive at a newly created invoice for Stark Industries with two line items — item S1 for $100, and item S2 for $200 — for a total of $300.

Now imagine that two *concurrent* operations occur:

- User Alice adds a new line item (item S3 for 150)
- User Bob deletes an existing line item (item S2)

Both operations load the current state (sequence 3), apply their changes, and attempt to write new events with sequence number 4, as depicted in Example 16-6.

Example 16-6. Concurrent events attempting to write with the same sequence number

```
// Alice's event (arrives first):
{
  "aggregate_id": "INV-125",
  "sequence": 4,
  "event_type": "LineItemAdded",
  "payload": {
    "item_id": "S3",
    "amount": 150
```

```
  }
}

// Bob's event (arrives second, will be rejected):
{
  "aggregate_id": "INV-125",
  "sequence": 4,
  "event_type": "LineItemDeleted",
  "payload": {
    "item_id": "S2",
    "amount": 200
  }
}
```

The event store successfully persists Alice's event with sequence 4 but rejects Bob's event with a `ConcurrencyException`, because sequence number 4 already exists for aggregate `INV-125`.

The command handler can catch this exception and implement a retry strategy:

1. *Reload the event stream* to get the latest state (now including Alice's event at sequence 4).
2. *Rehydrate the aggregate* by replaying all events, resulting in an invoice with items A1 ($100), A2 ($200), and A3 ($150), totaling $450.
3. *Reevaluate the business logic.* Bob was trying to delete item A2. Is this still valid? Yes, since item A2 still exists.
4. *Generate a new event* with the correct sequence number (5) and an updated payload reflecting the current state. This is shown in Example 16-7.

Example 16-7. Retry with updated sequence number

```
{
  "aggregate_id": "INV-125",
  "sequence": 5,
  "event_type": "LineItemDeleted",
  "payload": {
    "item_id": "A2",
    "amount": 200,
```

```
    "new_total": 250
  }
}
```

The final event stream now correctly captures both operations in the order in which they were successfully persisted, as Example 16-8 shows.

Example 16-8. Final consistent event stream

```
| aggregate_id | sequence | event_type      | payload
|
| ------------ | -------- | --------------- | ----------------------
--------------- |
| INV-125      | 1        | InvoiceCreated  | {vendor: "Stark
Industries"}          |
| INV-125      | 2        | LineItemAdded   | {item_id: "A1",
amount: 100}          |
| INV-125      | 3        | LineItemAdded   | {item_id: "A2",
amount: 200}          |
| INV-125      | 4        | LineItemAdded   | {item_id: "A3",
amount: 150}          |
| INV-125      | 5        | LineItemDeleted | {item_id: "A2",
amount: 200}          |
```

This optimistic concurrency control ensures that the event stream maintains strict ordering and consistency, while the retry mechanism allows the system to handle concurrent modifications gracefully. In some cases, the retry logic might determine that the operation is no longer valid — for example, if Bob tried to delete item A2 but another event had already deleted it. In this case the retry would fail and either return an application-level error to the user or log it for human intervention.

Schema Evolution

One of the biggest challenges with event sourcing is how to handle event schemas changing over time. As both business requirements and applications evolve, the events representing changes to the aggregates' states have to evolve as well. However, events stored in the event store are by definition historical records, so they can't be changed! Event stores themselves are immutable, append-only logs that should serve as a reliable foundation for downstream systems. Consumers should be able to trust that

the event history will not change retroactively. This allows them to build materialized views and projections confidently and eliminates data consistency issues caused by upstream modifications. This paradox means that managing event-schema evolution *cannot* be an afterthought when adopting event sourcing. Architects must consider this and make it part of the implementation from the get-go.

The first thing architects must do to address schema evolution is versioning the events themselves, as illustrated in Example 16-9.

Example 16-9. Tracking event schema versions via a specific attribute

```
{
  "aggregate_id": "INV-125",
  "event_type": "InvoiceCreated",
  "event_version": "v1", // track the schema version
  "payload": {...}
}
```

In the future, as the schema evolves, the `event_version` will need to be incremented. This will clearly signal to any processors or downstream systems that they need to handle multiple schemas for this event type.

As for evolving the schema, the easiest approach is probably ensuring that the changes are backward compatible. Changes can be *additive only* — deleting attributes can't be permitted. While this helps maintain backward compatibility, the trade-off is that cruft gathers in the data model. That can increase future storage costs and introduce confusion regarding whether an attribute is being used or not; there's always a risk that the field could be misused or misunderstood.

In some cases, another option is simply deleting unused or unnecessary attributes. Downstream systems can assume defaults for any attributes that they expect but don't see in the events themselves. The trade-off to this approach is that it introduces complexity — in the event of a misunderstanding or miscommunication about what the default ought to be, different systems and consumers could represent the data differently. Shifting from an additive-only approach to setting defaults simply shifts the

point where the architect makes assumptions from the data model to the logic processing the events.

Of course, these strategies may not always be sufficient. An architect might need to change an attribute's type or rename it in the event to better align with the business's terminology.

The aggregates themselves could also be refactored. If, say, one aggregate is broken into two or more, the events representing changes to these aggregates would also change, perhaps even significantly — and that typically breaks their backward compatibility.

In event evolutions where maintaining backward compatibility is not an option, architects can use a design pattern aptly named "Upcasting" that works similarly to upcasting or downcasting primitive types in code. An *event upcaster* inspects an event's version number and *casts*, or transforms, the event into the appropriate current version, as illustrated in Example 16-10.

Example 16-10. Upcasting events

```
public void on(InvoiceCreatedEvent event) {
  this.invoiceId = event.getInvoiceId();
  this.status = event.getStatus();

  // ensure catering to the right type and version
  if (rawEvent.eventType == "InvoiceCreated" && rawEvent.version ==
1) {
    // Transform v1 to v2 format
    return new InvoiceCreatedV2(
      rawEvent.id,
      rawEvent.vendor,
      rawEvent.issueDate,
      purchaseOrderNumber: null  // Add missing field with default
    );
  }
  return rawEvent;
}
```

When an upcaster transforms an event, it relieves downstream systems of the need to continuously adapt their logic to accommodate multiple versions.

Another option is to re-create the event stream as "V2." However, this would violate Event Sourcing's central tenet: that history is immutable.While it prevents the constant event-schema version checks, it technically *destroys* history, affecting the auditability of the system.

No matter what kind of change forces an event schema (or the mechanism used to accommodate multiple versions of it) to evolve, Event Sourcing attempts to record history via events. While immutable, append-only stores align well with this requirement, the nature of business is constant change. Aligning that notion with immutability introduces significant complexity.

Trade-Offs

Architects should be very deliberate about choosing a complex pattern like Event Sourcing, since it comes with significant trade-offs.

Pros

The advantages of applying this pattern include:

Auditability

> Event Sourcing absolutely shines here, offering unparalleled auditability. It provides complete audit trails with *entire* histories, in stark contrast to the traditional approach where updating the datastore simply erases history. Strong auditability is particularly useful for organizations that have strict compliance or record-keeping mandates. The ability to "time travel" by replaying the system up to any point in time isn't just a boon for debugging: it can be invaluable for forensics.

Cons

Complexity

> Event Sourcing is an extremely complex architectural pattern to implement correctly. Architecturally, as discussed earlier, Event

Sourcing leverages eventual consistency at the *system level* — bringing with it concerns like managing distributed transactions, using compensating actions, and handling failures. If the scale and volume of events require projections, which typically lag behind writes and are used on the "read" side of the system, architects must consider how this affects users when designing the UI.

Infrastructurally, Event Sourcing systems *may* require multiple datastores: the event store itself, a snapshot store, and potentially middleware if events will be published for consumption by downstream systems. Each of these datastores brings concerns like observability, backup and disaster recovery strategies, and security.

At scale, the need for storage can grow at an incredible rate, and architects must consider the associated costs and devise archival and retention strategies.

At the design level, it's crucial to understand DDD, aggregates, and domain events. It's hard to refactor the aggregates and domain events without significantly affecting the event stores and event processors, which is why domain events require a well-established versioning strategy.

Query patterns also need to be revisited, especially if third-party tools expect a relational datastore. Architects might need to think through new projections (and thus replays) to answer specific kinds of queries and new business intelligence questions. Architects must also think through governance issues before leveraging event stores, like the "right to be forgotten" (enshrined in the EU's General Data Protection Regulation, or GDPR, and other regulatory frameworks).

Cost

Not only is Event Sourcing operationally complex, it requires mature engineers and well-established architecture and design practices, all of which come at a price.

Comparison Chart

Table 16-3 summarizes the trade-offs associated with this pattern. It shows a comparison of using a service backed by a datastore with using the Event Sourcing pattern.

Table 16-3. Comparison chart for the Event Sourcing pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Auditability	★★	★★★★★
Debuggability	★★★	★★★★★
Testability	★★★	★★★★
DEGRADED		
Cost	$$	$$$$
Data Consistency	★★★★★	★★★
Data Integrity	★★★★★	★★★★
Simplicity	★★★★★	★

The following sections describe each of the capabilities found in Table 16-3.

Improved

The capabilities that are improved when applying this pattern include:

Auditability

Events, and thus the event store, capture every state change and its business context (the "why"). This helps provide comprehensive audit trails, which is especially important in highly regulated industries. Contrast that with a traditional datastore-backed CRUD system, wherein history is often stamped on by updates. Even with additional mechanisms like history tables, auditability still suffers with traditional stores.

Debuggability

Certain production bugs are often hard to reproduce — by the time the team does a postmortem, the state of the system has changed, and the exact set of conditions that manifested the bug is lost. Being able to replay the state of the system up to any point in time can help teams understand not just what the bug was and how it might be resolved, but *how* the system arrived at that point. This could prompt them to reevaluate the UI and controls, the user journey, or even the interactions and expectations between different parts of the system. For example, perhaps the UI permits a user to skip a mandatory field or a wizard allows the user to submit midway. If one part of the system makes assumptions about how and when another part completes its portion of the work, this can lead to inconsistencies.

Testability

Testability sees a slight improvement. Domain events in DDD dovetail well with *behavior-driven development* (BDD), which typically uses "given-when-then" scenarios. Architects can make testing easier by decoupling aggregates from handlers that affect them and from their own side effects (the events that aggregates themselves produce). This way, tests can focus solely on domain logic. On the other hand, this system *is* eventually consistent, which brings with it complexities in testing.

Degraded

The capabilities that are degraded when applying this pattern include:

Cost

> Finally, storing all of the events for all time can be expensive. Architects can mitigate some of these costs by leveraging different storage types (hot versus cold) for new versus older events — or perhaps devise a deletion strategy that meets regulatory or compliance requirements. This too demands strategic thought. How long should events stay in hot storage, and when can they be moved to cold storage or an archive? How will the system ensure that it can provide a usable snapshot when events leave hot storage?

Data Consistency

> Recall that when an aggregate's state or any events generated from it change, those changes happen *within a transaction*. This provides strong *write* consistency, especially when paired with checking the sequence number prior to persisting or rolling back the whole transaction (see "Handling Concurrency").
>
> However, in the Event Sourcing pattern, reads almost always lag behind writes, so business logic and UI components have to accommodate for the lag. The lag may come from having to re-create the new state of an aggregate by replaying all the events, or waiting on projections to update since the last write to the event stream.
>
> If downstream systems are reacting to new events being generated, then the system *as a whole* exhibits eventually consistent behavior, since different parts of the system process events at different rates. If subsequent operations can't be completed, the architect will need to design and install compensating controls.
>
> Contrast this with CRUD-based systems, which offer strong write, read, and system consistency and thus allow for significantly simpler system designs.

Data Integrity

Turns out, Event Sourcing strengthens data integrity, especially within a single aggregate (since all interactions to the event store are transactional). The event log also acts as an immutable audit trail, further improving integrity.

However, it's at the system level that data integrity is sufficiently degraded, to the extent that this capability falls squarely in the degraded list. The very nature of eventual consistency *across* the different aggregates, particularly in comparison with a traditional datastore that enforces referential integrity within ACID transactions, makes maintaining cross-aggregate data integrity constraints significantly more challenging.

Simplicity

Simplicity is the Event Sourcing pattern's Achilles' heel. While other patterns like the Command Query Responsibility Segregation (CQRS) pattern in Chapter 10 can help improve the complexity of querying for data, they bring their own set of architectural trade-offs. Event Sourcing systems are eventually consistent — so all the challenges of eventual consistency crop up.

From an operational perspective, the system could have multiple datastores, including the event store itself, such as a snapshot store or a read model datastore for projections. Furthermore, snapshot management, building (and rebuilding) projections all require a deep understanding of not just how to set up operational event stores, but also manage them.

Publishing events may require additional middleware, which introduces its own varying operational considerations. It may require the architect to design the downstream systems to be idempotent, or to design the entire system needs to allow for a deduplication strategy.

Querying patterns are also different in event-sourced systems. Different aggregates might appear at different points in the event log. Add snapshots and projections, and rebuilding the state of the entire system becomes a lot harder. Commercial and open source analytics and business intelligence tools often assume a system with a traditional datastore, so getting them to work with the event store requires additional considerations.

From a design perspective, Event Sourcing demands a good understanding of DDD concepts such as aggregates, domain events, repositories, and services, as well as good implementations of all these in code. Architects must think through event schema versioning strategically, considering what kind of versioning schema to use and how best to deal with multiple versions of the same event.

In a CRUD-based system, even one with a distributed architectural style like microservices, most teams will be familiar with the operational patterns, failure modes, and tooling. Event Sourcing introduces operational complexity that extends beyond coding concerns, requiring expertise in distributed systems, eventual consistency patterns, and polyglot persistence strategies that may be new to the organization.

Influences

Several factors can significantly affect the trade-offs and success of Event Sourcing implementations. Understanding these influences helps architects make informed decisions about when and how to apply this pattern.

Consistency Model and Atomicity

Event Sourcing has two completely opposite notions of transactional guarantees: strong consistency *within an aggregate*, but eventual consistency *across the system*. That is, all state changes *and* subsequently generated events are committed atomically. However, if those events affect the state of other aggregates, those aggregates can't participate in the same

ACID transaction as the originating aggregate. This significantly degrades data consistency and integrity *across* the system. To counter this, architects must be always keep this dichotomy in the back of their minds, and accommodate for it architecturally by using distributed coordinators or sagas to coordinate atomicity across the system.

Event Volume and Scale

The scale at which events are produced has a dramatic effect on the system's responsiveness, throughput, cost, and complexity. Systems that are expected to produce events at a low volume allow for easy and fast reconstruction of state. This avoids the burden of storing snapshots. This keeps the architecture simple, and costs manageable.

On the other hand, systems that produce large amounts of events make it extremely hard to rehydrate an aggregate's state. Replaying the event stream takes too long, so snapshots are needed. Any updates to the event schema or query pattern require rebuilding the projections, which can take a very long time with a long event stream. Finally, storage costs can balloon, so architects must build their strategies around tiered storage mechanisms, archiving and, if possible, deleting events after a certain period. All this tremendously complicates the system.

Architecture Quantum

As discussed in "Architecture Quantum", architects must consider how coupling affects architectural scope as it relates to persistence patterns like Event Sourcing. This section provides a quick rundown of the coupling types relevant to this pattern:

Static coupling

> *Static coupling* refers to the dependencies a service or system requires to function independently. The Event Sourcing pattern requires, at minimum, an event store that acts as an immutable event log. This allows the system to preserve its own bounded context, but expands the system's static quantum to include the event store itself. Without the

event store, the system can't function. Depending on the scale and volume of events, it may need read patterns, additional databases to store projections, and perhaps even a snapshot store. While the event store always presents itself as a static dependency in the Event Sourcing pattern, projection and snapshot store may still be required.

Dynamic coupling

Dynamic coupling does not apply to the Event Sourcing pattern, which focuses on asynchronous event persistence.

Contract coupling

Contract scope, in Event Sourcing, refers to the event schemas that form the contracts between the write model, read models, and potentially other downstream consumers. Events are the primary contract mechanism, and they affect quantum boundaries in several important ways. Any event, published in an event sourcing system, is consumable by multiple consumers:

- Internal projection builders within the same service
- External services in other quanta that depend on those events
- Audit systems, analytics platforms, and other cross-cutting concerns

This creates a challenging situation: event schemas become a shared contract that couples multiple quanta together. Changing an event schema could affect every consumer of that event across the entire architecture. This is why event versioning (see "Schema Evolution") is *not* optional in Event Sourcing — it's an architectural necessity to prevent breaking changes from rippling across quantum boundaries.

Events often carry more information than is strictly necessary for their immediate purpose, like additional context needed for workflow management, audit trails, or future projections that haven't been built yet. This can lead to a subtle form of the *Stamp Coupling* antipattern

(see Chapter 31), where event consumers depend on only a subset of the event's payload, but the full event defines the contract scope.

Architects can mitigate this by using different event types for different consumers, such as domain events (which are detailed and aggregate-specific) for internal projections, and integration events (which are coarser grained and more stable) for external consumers.

This adds complexity, but can help maintain clearer quantum boundaries by limiting the scope of contract coupling.

Temporal coupling

Temporal coupling doesn't really apply in the Event Sourcing pattern, which has no specific time-based dependencies.

Atomic coupling

Atomic coupling is the composite of data consistency and integrity, and occurs when architects must build pseudotransaction workflows to achieve both consistency and integrity. The Event Sourcing pattern makes it much harder to maintain atomicity *across* multiple aggregates. If that's needed, architects must introduce additional mechanisms to coordinate atomicity across the system, as described in "Consistency Model and Atomicity".

How to Govern This Pattern

Design frameworks like DDD are often necessary to implement Event Sourcing correctly — and these come with their own requirements that must be governed. For example, tests must ensure that aggregate boundaries are not violated, and that all save operations involving aggregates and generated domain events guarantee atomicity. The code needs mechanisms to ensure that all domain events have certain behaviors — for example, object-oriented techniques like inheritance (all domain events must inherit from the same base class) — or tests, such as to ensure that all domain events have schema versioning baked in.

Things are even more interesting on the architectural side. Given this pattern's complexity, governance is a must. Event Sourcing systems are eventually consistent at the system level. The question is: how eventual is "eventual"? Does it have to be submilliseconds, or can it be a few hours? Architects must carefully weigh the trade-offs and build fitness functions that appropriately define the service-level indicators (SLIs), monitor the service-level objectives (SLOs), and guard against any violations of the service-level agreements (SLAs), if any.

Given this pattern's tendency to produce topological sprawl, static coupling with multiple pieces of infrastructure means that reliability or uptime concerns with any one piece can affect the whole system. Architects must leverage observability to ensure that all parts of the architecture are behaving as expected at all times, and use chaos engineering to pressure-test both the system and their own assumptions to ensure resiliency.

Related Patterns, Antipatterns, and Pitfalls

The following patterns are related to the Event Sourcing pattern:

Command Query Responsibility Segregation (CQRS) pattern (see Chapter 10)

> The CQRS pattern is often used alongside Event Sourcing. It separates the write side (data modification) and the read side (data retrieval) into different modes. Event Sourcing has atomic guarantees around writes at the aggregate level, but reads lag behind since they require rehydrating the aggregate via event replay or rebuilding projections to facilitate reading. In other words, reads have a different modality than writes in terms of how they're performed (aggregate hydration) and stored (projection store versus event store).

Cache Aside pattern (see Chapter 8)

> Caching is fundamental to making Event Sourcing performant in production systems, particularly for read-heavy workloads. The Cache

Aside pattern aligns well with how projections and read models are typically managed in Event Sourcing architectures.

From an implementation perspective, the Cache Aside pattern affords a great degree of flexibility — which allows the application control over how and which projections to build, what data to include — and even how long to retain them.

Read-Through/Write-Through Cache pattern (see Chapter 29)

The Read-Through/Write-Through Cache pattern, and particularly its Write-Behind variant, relate to Event Sourcing, notably for snapshot management.

When it's time to capture a snapshot, the application can write the snapshot to an extremely fast write-behind cache, which can then update a persistence store in batches. Recall that snapshots aren't a source of truth — merely a source of record. Even if the cache, or a batch update to a persistent store, fails — the application can always reproduce the state of the aggregate by fetching an older snapshot and updating it with events that appeared since, or simply start from the beginning.

Chapter 17. Explicit Workflow State Pattern

When working with distributed architectures, tracking the state of any workflow can be challenging. Which services have executed their step in the workflow? What was the result? Did the system encounter any errors?

Some architectural patterns, such as Orchestration (see Chapter 27) have a built-in location for workflow state: the orchestrator itself. Other patterns are specialized, like Cross-Cutting State (Chapter 13) and Front Controller (Chapter 19) for choreographed workflows. The Explicit Workflow State pattern offers another solution to this problem.

Table 17-1 summarizes the metadata for the Explicit Workflow State pattern.

Table 17-1. Metadata for the Explicit Workflow State pattern

CATEGORY	METADATA
CONTEXT:	Event-Driven Architecture, Message-Based Architectures, Microservices, Workflow
IMPROVED:	Testability, Workflow
DEGRADED:	Adaptability, Domain Alignment, Evolvability, Simplicity

How Should I Manage My Workflow State?

Distributed architectures make it hard to track state. Consider a distributed architecture style like event-driven architecture (EDA): an initiating event (such as `order_placed`) kicks off the workflow. This is consumed by one or more downstream processors or services, which process it, then publish their own derived events for consumption by other processors. The pressing question about any particular workflow is often: "What is the *current* state?"

Consider the scenario illustrated in Figure 17-1. A customer places an order for a book, which is the initiating event that triggers the workflow. As different processors react to that event, they trigger derived events. For example, the `Order Placement` service emits an `order placed` event. Three downstream processors subscribe to that event: `Payment`, `Notification`, and `Inventory`.

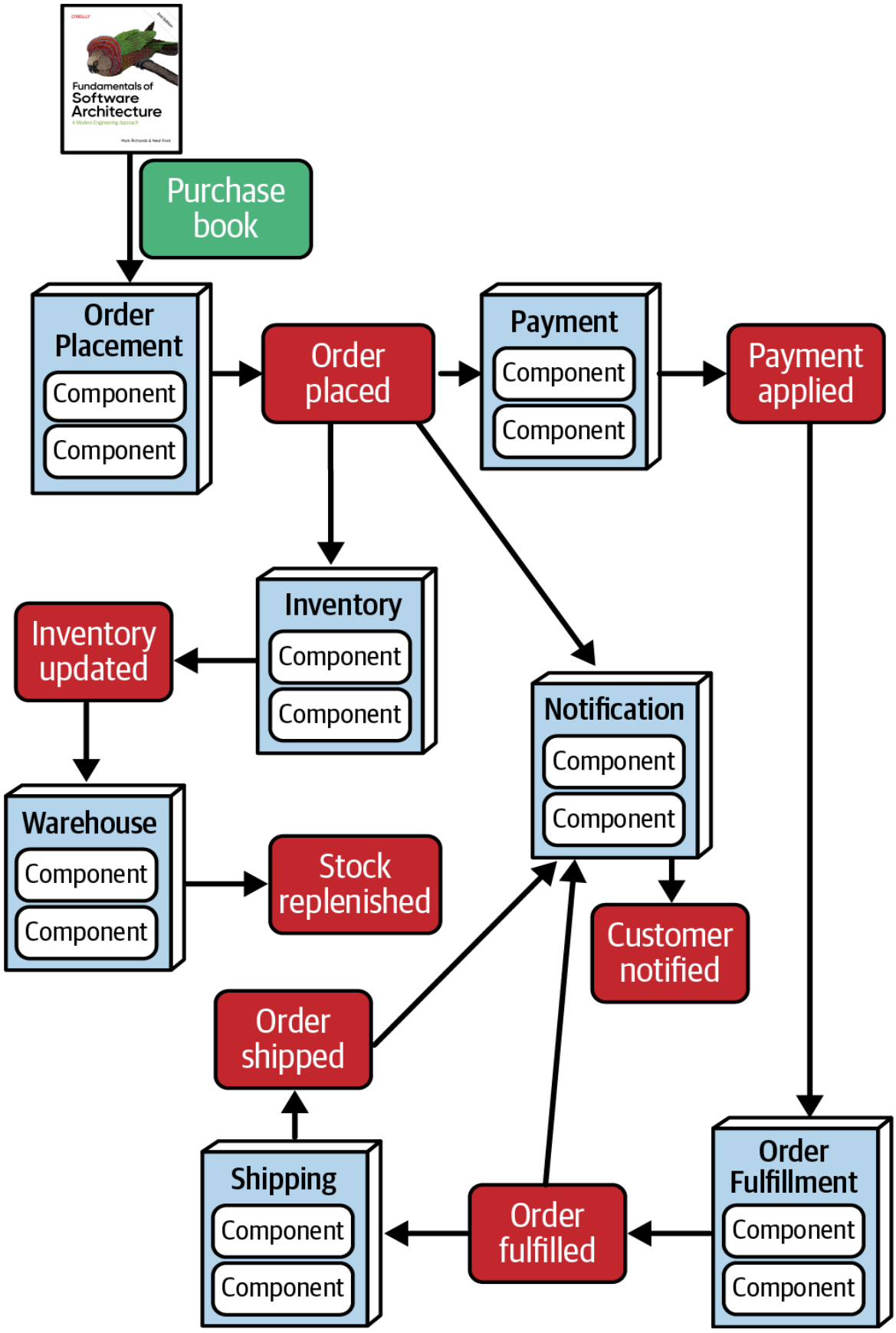

Fundamentals of Software Architecture
Purchase book
Order Placement
Component
Component
Order placed
Payment
Component
Component
Payment applied
Inventory
Component
Component
Inventory updated
Notification
Component
Component
Warehouse
Component
Component
Stock replenished
Customer notified
Order shipped
Shipping
Component
Component
Order fulfilled
Order Fulfillment
Component
Component

Figure 17-1. A typical processing flow for an event-driven system

Let's say the customer wishes to change or cancel their order. Now what is the state of their order? In the architecture depicted in Figure 17-1, it's hard to discern an answer, because no singular source captures state! In any distributed architecture, "Where does the state reside?" is often a pressing question. This is where the Explicit Workflow State pattern can help.

Applying the Pattern

To apply the Explicit Workflow State pattern to any message-based or event-driven topology, architects must ask three questions:

- What external state is missing?
- Where does the data reside that would allow us to hydrate state?
- What's the best location for the state to reside?

Consider Figure 17-1 once again. Where in this system could a particular customer's order be found? Look carefully at the workflow: it includes all the processors that act directly to affect customer orders, as shown in Figure 17-2.

The next step is to have a *workflow state owner*, a service or processor that acts as a one-stop shop for all things related to customer orders' state. This component can be queried any time to answer questions about the state of customer orders. Much like in the Front Controller pattern (see Chapter 19), it's common for this to be the first service in the workflow.

In the Explicit Workflow State pattern, each processor that modifies the state (in this case, a customer order), in addition to publishing its own event, also publishes an update to a "state-change" event channel, which in turn updates the state in the workflow state owner.

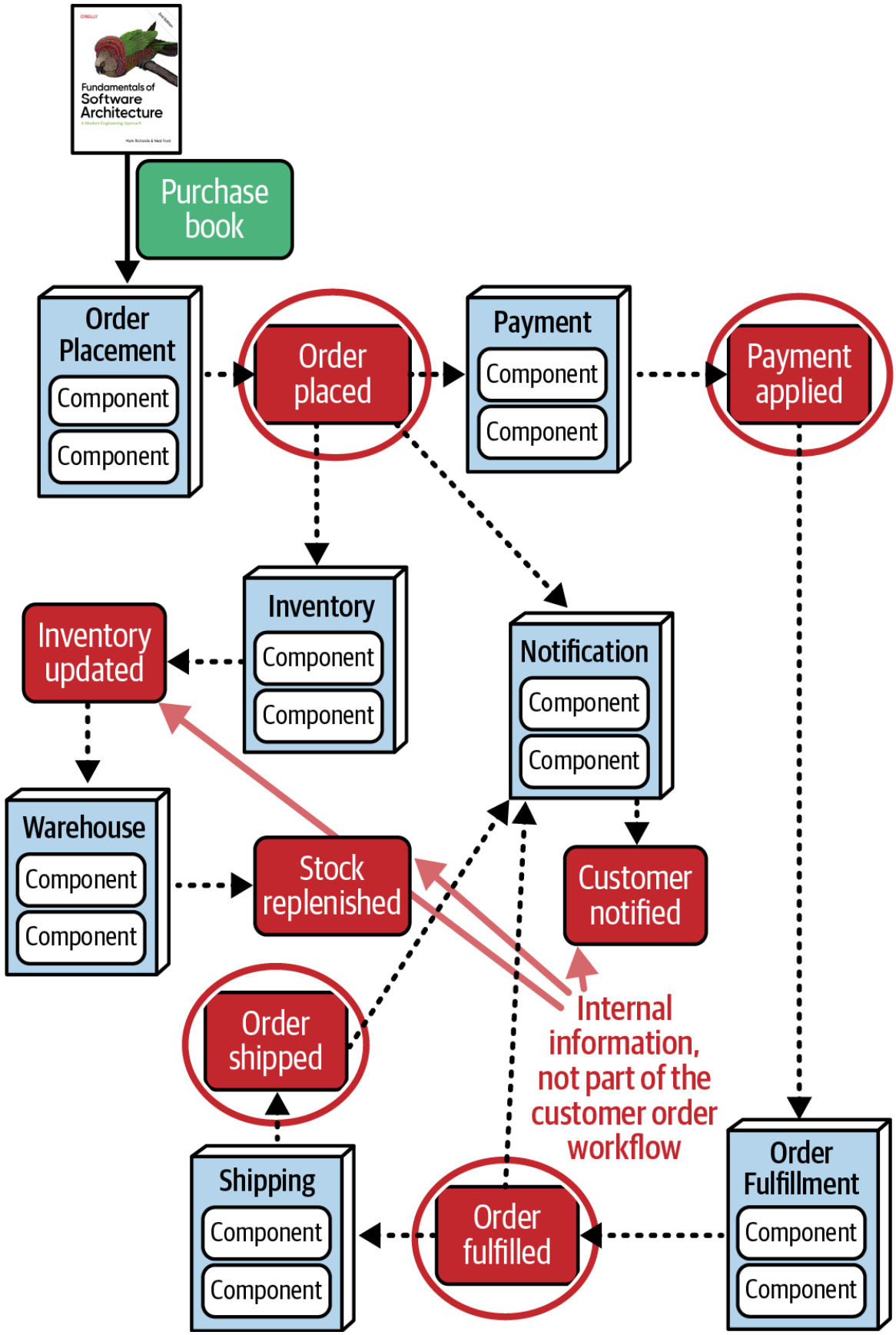

Fundamentals of Software Architecture
Purchase book
Order Placement
Component
Component
Order placed
Payment
Component
Component
Payment applied
Inventory
Component
Component
Inventory updated
Notification
Component
Component
Warehouse
Component
Component
Stock replenished
Customer notified
Order shipped
Internal information, not part of the customer order workflow
Shipping
Component
Component
Order fulfilled
Order Fulfillment
Component
Component

Figure 17-2. Events pertaining to customer orders

Figure 17-3 shows how each processor sends an update. For example, the `Payment` processor charges the customer, then publishes its regular `payment applied` event for consumption by other processors downstream. It sends a `PAID` update to the state-change event channel, which the workflow state uses to update that particular order's state. From that point on, answering the question, "What happened to this order?" is as easy as looking up the latest update for that particular order using the workflow state owner.

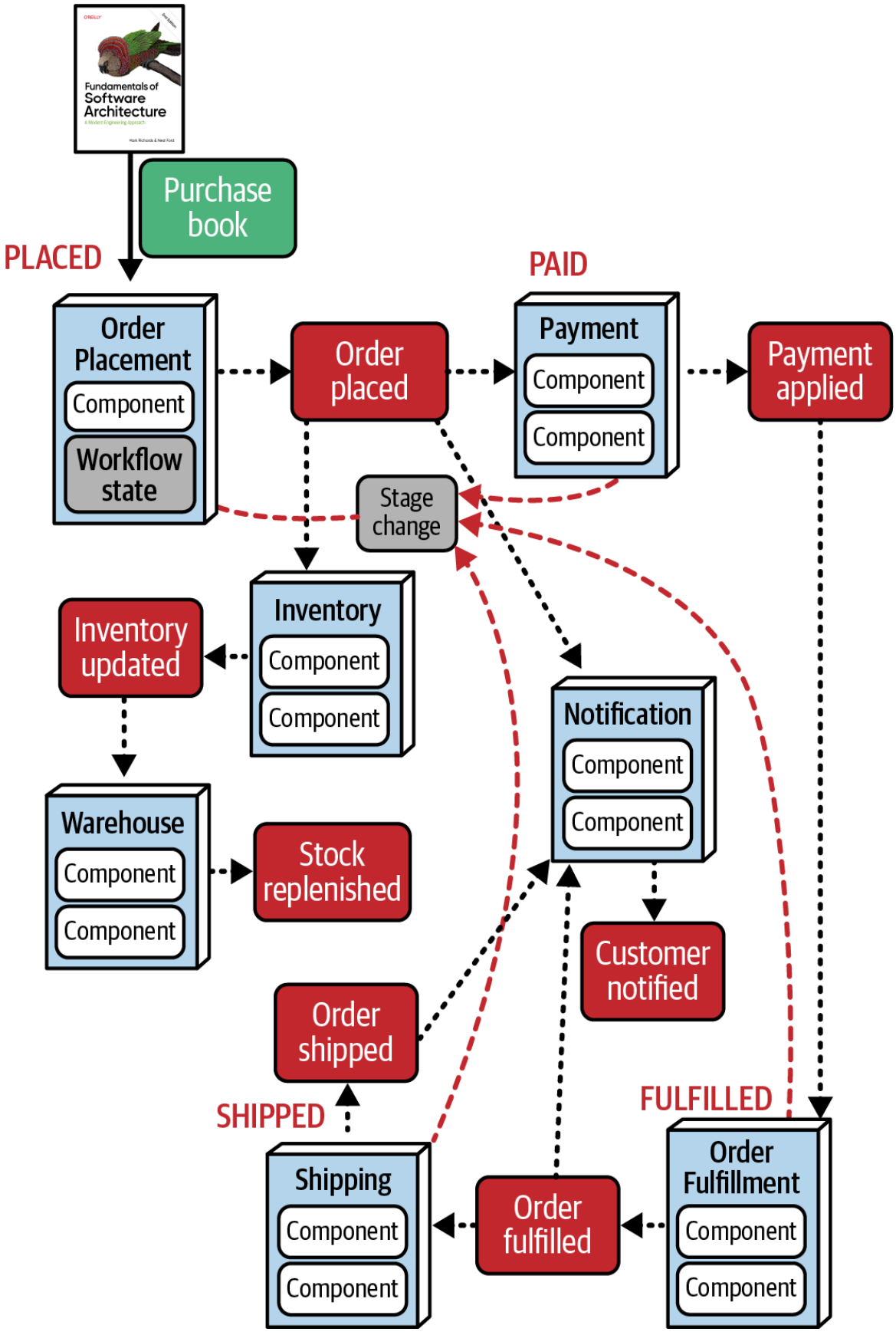

Fundamentals of Software Architecture
Purchase book
PLACED
PAID
Order Placement
Component
Workflow state
Order placed
Payment
Component
Component
Payment applied
Stage change
Inventory
Component
Component
Inventory updated
Notification
Component
Component
Warehouse
Component
Component
Stock replenished
Customer notified
Order shipped
SHIPPED
FULFILLED
Shipping
Component
Component
Order fulfilled
Order Fulfillment
Component
Component

Figure 17-3. Explicit Workflow State pattern

In effect, the Explicit Workflow State pattern creates a *distributed* state machine owned by the workflow state owner, with multiple players involved in transitioning it.

The Explicit Workflow State pattern forces every processor involved to be modified to know of the state-change event channel and explicitly publish that event. This also means that if one or more processors forgets or fails to publish to the state-change event, the system can no longer reliably depend on the workflow state to report the right status.

Trade-Offs

This section discusses the trade-offs of the Explicit Workflow State pattern.

Pros

The advantages of applying the Explicit Workflow State pattern include:

Workflow

> Tracking state is such a common problem in distributed architectures that architects created a named pattern for it. The Explicit Workflow State pattern solves a real problem: knowing the state of any workflow in an architectural style that does not intrinsically manage state.

Initiating event owner

> Typically, the state owner is the first-called service or processor in a workflow. This may be familiar, since it resembles other patterns, like Front Controller. From an implementation perspective, this pattern does not pose any surprising changes to the topology.

Separation of concerns

> Having a distinct flow of messages through the system explicitly to track state provides a clean separation of concerns. Business messages

continue to flow throughout the system as they always did. Architects can decide to emit more or fewer derived events based on requirements, and message schemas can change without affecting the workflow state owner. Updating the state via the state-change event channel becomes an out-of-band operation, with a set of messages specifically designed for the "internal plumbing" of state management.

Cons

The downsides of the Explicit Workflow State pattern include:

Broker coupling

This pattern does increase coupling within the system. Every service that participates in updating the state of the workflow in question is now aware of, and coupled to, the state-change event channel. Other patterns that introduce domain-specific brokers to narrow the architectural quanta, like Domain-Broker (see Chapter 15), are negatively affected by this pattern, since it can broaden the scope of coupling across the architecture.

Complexity

The Explicit Workflow State pattern adds complexity to every service, since every service involved in updating the workflow state has to be modified to update *and* push state. This complexity also introduces additional failure scenarios. At a minimum, every service now has to publish two events — except what happens if the publication of one event succeeds but the other fails? This introduces drift between the workflow's *actual* state and its *recorded* state. This is in effect, the "dual write problem" in which the system needs to update data across two independent destinations (like a database and a message broker) as part of a single logical operation. The Transactional Outbox pattern is a common solution.

Comparison Chart

The trade-off analysis chart for the capabilities affected by this pattern appears in Table 17-2. This chart evaluates both *workflow inspection* capabilities (how easy it is to track workflow state) and *service implementation* capabilities (the cost and complexity each approach imposes on individual services). "Without Pattern" refers to the case where workflow state is not universally managed or tracked and services maintain their own state, which makes discerning workflow state extremely hard. This chart also contrasts this pattern with the Implicit Workflow State pattern, an alternative pattern described in Chapter 22, in which the workflow state owner taps into events published by services and attempts to *infer* their state.

Table 17-2. Comparison chart for the Explicit Workflow State pattern

Capability	Without Pattern	Implicit Workflow State Pattern	Explicit Workflow State Pattern
IMPROVED			
Testability	★	★★★	★★★★★
Workflow	★	★★★★	★★★★
DEGRADED			
Adaptability	★★★★★	★★★★	★★★
Domain Alignment	★★★★★	★★★★	★★★
Evolvability	★★★★★	★★★★	★★★
Simplicity	★★★★	★★★	★★

All of the capabilities that the Explicit Workflow State pattern improves are related to tracking and testing workflow state, while all service-implementation capabilities suffer in this pattern.

Improved

The following capabilities are improved when using the Explicit Workflow State pattern:

Testability

Testing is significantly improved, as compared to both "Without Pattern" and "Implicit Workflow State Pattern," because it provides architects with well-known testing targets and outcomes. The Implicit Workflow State pattern also improves testability substantially over having no pattern, but requires more setup and coordination across multiple services for the test setup and verification stage. Since the Explicit Workflow State pattern requires every participating service to emit an explicit "state update" event, and centralizes state in the workflow state owner, this makes test verification straightforward and deterministic.

Workflow

Workflow management is the primary reason to utilize this pattern. Workflows often have shared state across services, and this pattern provides a well-defined way to manage it. Without a pattern, querying for workflow state involves querying each service in the workflow and stitching the state together. The column Implicit Workflow State Pattern loses a star because the state is inferred from domain events. Explicit Workflow State Pattern also loses a star because this pattern, as explained earlier, creates a distributed state machine, with multiple players involved in transitioning it. This means multiple points of failure, the potential for the dual write problem, and increased coupling across the architecture, all of which make drift between the actual and recorded workflows more likely.

Degraded

The following capabilities are typically degraded when teams choose this pattern, stemming from two key requirements: service implementations must emit an additional event to update state beyond their existing domain events, and the workflow state itself is managed in a separate component (the workflow state owner) rather than within each service. The Implicit Workflow State Pattern column does better than this pattern across all *degraded* capabilities because it does *not* require services to be modified to

aid in workflow state management. State is *inferred* from existing domain events that services publish — leaving services untouched.

Adaptability

Every service must know about and emit to the state-change channel, which must always be top of mind when modifying a service. Furthermore, changing state tracking requires coordinating across all participating services.

Domain Alignment

Consider Figure 17-3 again. An architect might look at this diagram and see multiple domains at play. The "customer-facing" domain involves the `Order Placement` and `Payment` services, while `Order Fulfillment` and `Shipping` are about getting the orders to the customers. In fact, certain patterns, like the Domain-Broker pattern (Chapter 15), leverage this to break up their architectural quanta into multiple smaller quanta.

The Explicit Workflow State pattern requires all participating services to be aware of the workflow state-change channel, which increases coupling and weakens domain abstraction.

Evolvability

Services have at least two concerns: publishing events for downstream consumption and emitting events that update the workflow state. This couples all the services participating in the workflow to the state-change channel, in terms of both static and contract coupling. That coupling makes evolving the architecture harder, since both new services and evolving existing services must always emit a state-update event alongside any domain-specific events.

Simplicity

Without workflow state tracking, each service is independent, but workflow inspection is a lot more difficult. Using this pattern makes it

significantly easier to track workflow state, but it also requires modifying every participating service to perform an additional action when updating state. This is an overloaded concern for every service. It increases implementation complexity and mandates some level of governance. The workflow state owner is also more complex because it addresses two orthogonal concerns: its own business concerns and tracking state. Finally, adding state means adding moving parts, which increases operational overhead and degrades simplicity.

Influences

Implementing the Explicit Workflow State pattern means that every service that updates workflow state must be aware of the state-change channel. That's coupling, and it affects the architecture quantum.

As defined in "Architecture Quantum", architecture quantum has five facets, some of which relate to the Explicit Workflow State pattern:

Static coupling

> *Static coupling* refers to the dependencies required for a service or system to function independently. When Explicit Workflow State is implemented, every service that affects workflow state is statically coupled to the state-change channel, increasing static coupling across the architecture.

Dynamic coupling

> *Dynamic coupling* does not apply to the Explicit Workflow State pattern.

Contract coupling

> This pattern *might* introduce some contract coupling, given that every service involved in updating the workflow now emits a specific message to update the state-change event channel. This means that going forward, architects may need to consider any changes to that

message schema, since it's specifically designed for one purpose, and one purpose alone.

Temporal coupling

Temporal coupling doesn't apply to the Explicit Workflow State pattern, which has no specifically time-based dependencies.

Atomic coupling

Atomic coupling and atomicity concerns are not influenced by this pattern.

How to Govern This Pattern

Architects choose this pattern when they need to capture some external state, which means that the state itself is a good candidate for governance. To bolster confidence in the overall implementation of this pattern, we recommend maintaining a registry of services that participate in updating state, and then installing some checks to ensure they're always emitting state updates.

Related Patterns, Antipatterns, and Pitfalls

The patterns related to the Explicit Workflow State pattern include:

Implicit Workflow State pattern (see Chapter 22)

The Implicit Workflow State pattern, specifically designed for event-driven architecture, is a complement to the Explicit Workflow State pattern with different trade-offs. For architects seeking a solution to state management in EDA, it's a worthy contender.

Cross-Cutting State pattern (see Chapter 13)

Cross-Cutting State is a generic pattern that helps manage and track state in many different kinds of distributed architectures.

Front Controller pattern (see Chapter 19)

This variant of the Cross-Cutting State pattern places workflow state among the first-called choreographed services.

Chapter 18. Extensible Events Pattern

In event-driven architectures (EDAs), services respond to events, process events, and then trigger corresponding derived events based on the processing they performed and state changes within that service. This event chain continues until no further events are triggered. When a service is done processing the event, the event chain terminates in one of two ways: not triggering an event or triggering a derived event, even if no other services are interested in it. The Extensible Events pattern helps architects and developers make that choice.

Table 18-1 summarizes the metadata for the Extensible Events pattern.

Table 18-1. Metadata for the Extensible Events pattern

CATEGORY	METADATA
CONTEXT:	Event-Driven Architecture
IMPROVED:	Adaptability, Agility, Deployability, Evolvability, Maintainability, Testability
DEGRADED:	Bandwidth Usage, Simplicity

Should I Trigger an Event If No One Is Listening for It (Yet)?

When processing an event, a responding service can either terminate the event chain or trigger a corresponding derived event that advertises what it

has just done to the rest of the system. Most often, responding services will trigger a derived event. In Figure 18-1, a `Payment` service responds to an `order placed` event, applies the payment for the order, and triggers a `payment applied` event to the rest of the system, letting other services know that the order has been paid for. In this example, the `Order Fulfillment` service is interested in that event because it can't fulfill an order until payment is applied. Such events are referred to as *significant events* because they're part of the major workflow for an order.

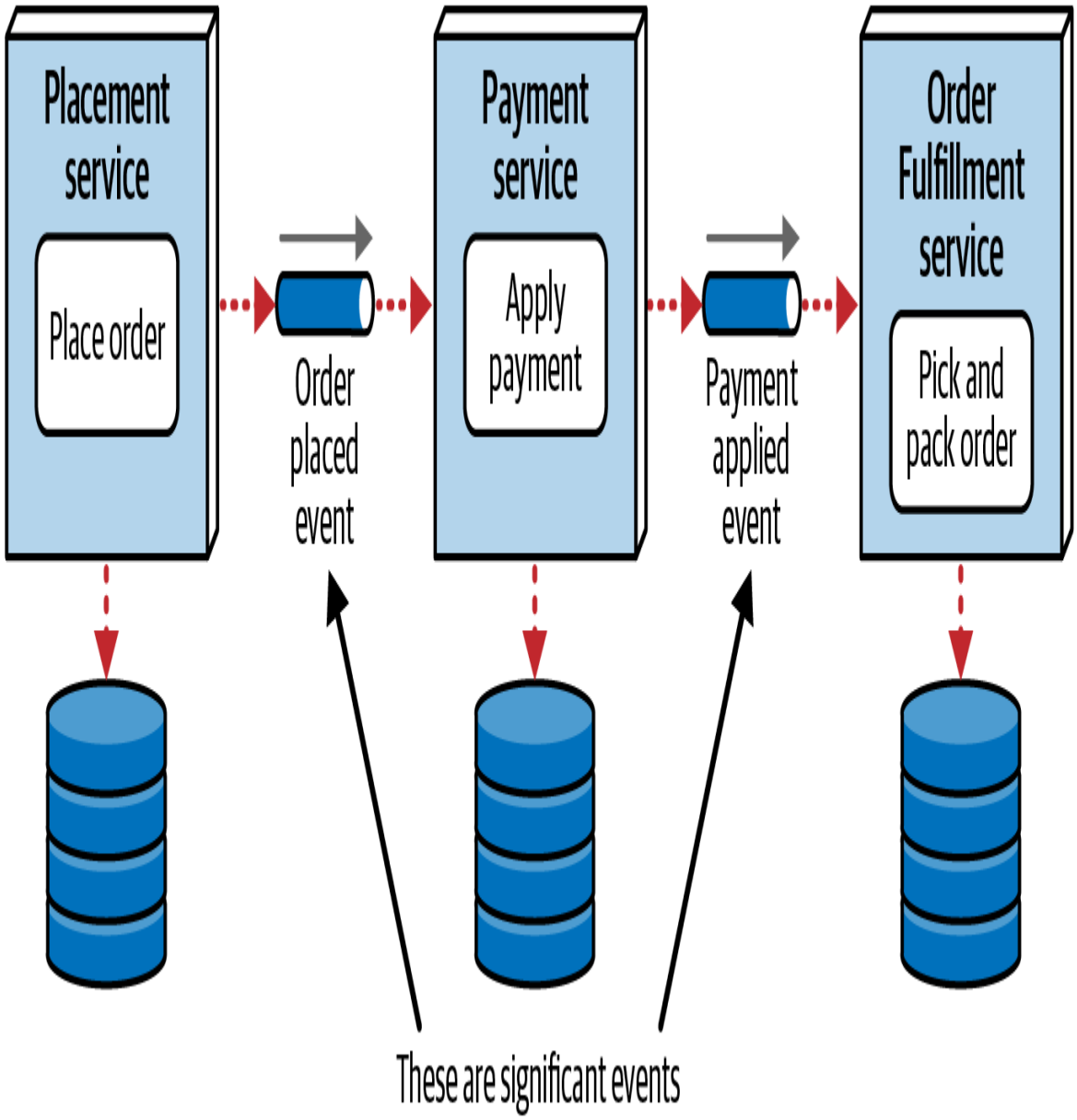


Figure 18-1. Significant derived events are part of the major workflow of a system

Now, suppose the `Customer Notification` service responds to an `order shipped` event to let the customer know their order is on its way, as illustrated in Figure 18-2. Here, the `order shipped` event is a significant event in the order processing workflow. Once the `Customer Notification` service emails the customer, thus completing its task, it can terminate the event chain or trigger a corresponding `customer notified` event that broadcasts to the rest of the system that it sent an email notification to the customer. This is a difficult choice: `customer notified` is not overly significant or part of the major customer workflow, and no other services in the system would respond to the event. Thus, it would be a wasted effort.

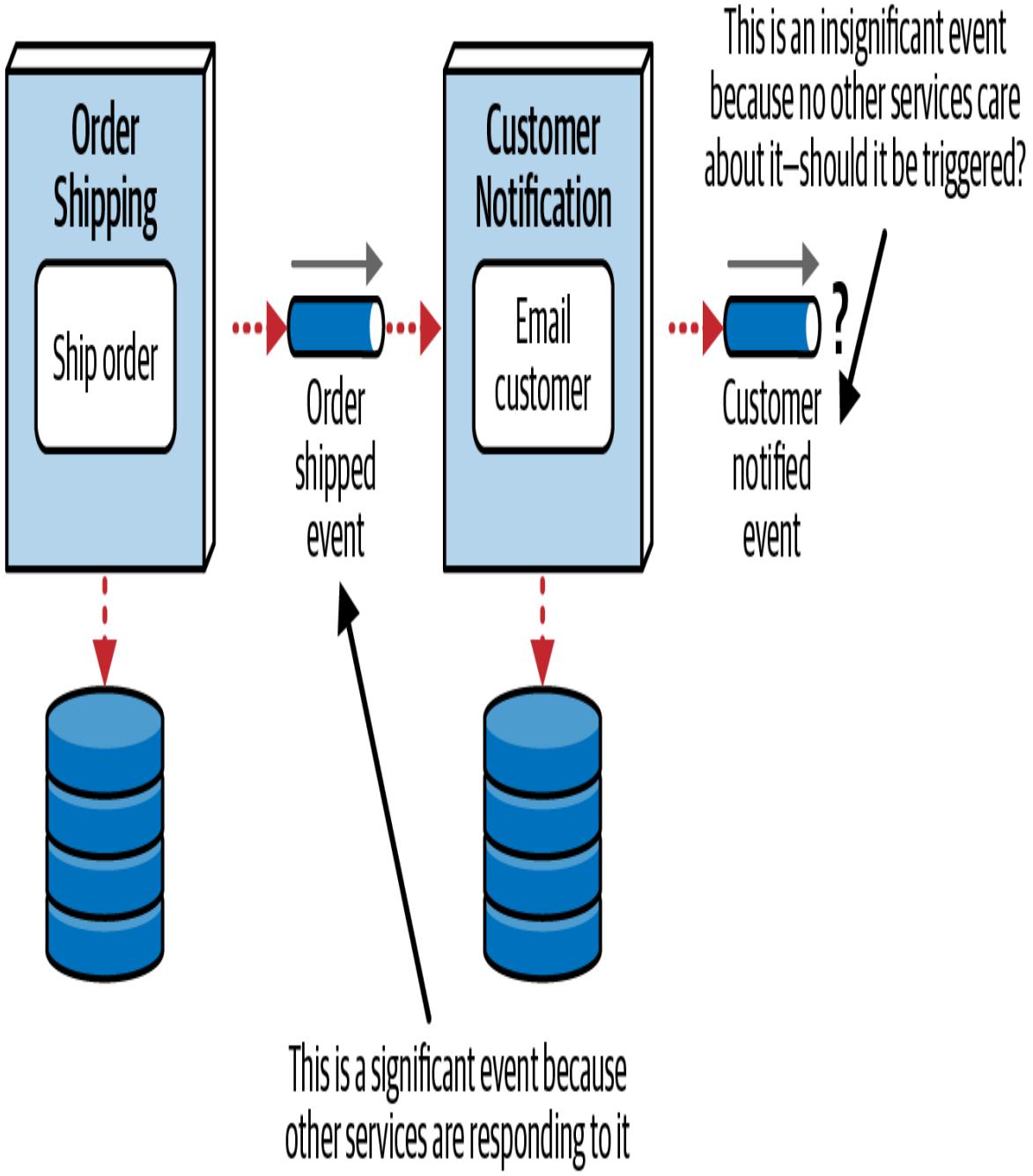


Figure 18-2. Insignificant derived events may or may not be triggered

In the Extensible Events pattern, a responding service advertising what it has done *always* triggers one or more derived events. Using this pattern promotes *architectural evolvability*, an architectural capability that describes how easy it is to extend a system's architecture with new behavior. Teams applying this pattern don't have to decide whether or not to end the event chain: it just keeps going.

Applying the Pattern

Applying this pattern means always triggering derived events when a service completes its processing. This practice allows the system to incorporate future services more easily, providing a good level of architectural evolvability. Future services need only subscribe to existing events; they don't need to modify and redeploy other services.

In the previous example, the `Customer Notification` service responded to an `order shipped` event by emailing the customer to let them know their order has been shipped. This action marks the end of the event chain for processing that order. However, if a successful email triggers a `customer notified` derived event and a failed email sends a `notification failed` derived event, future services can easily subscribe to those events.

For example, suppose the company wants to start collecting analytical data on the emails it sends to customers, logging how many fail due to bad email addresses or email send errors. With this pattern, a new `Email Analytics` service would only need to subscribe to the already published `customer notified` and `notification failed` events, requiring no other changes in the system, as illustrated in Figure 18-3.

> **CONFLICTING PATTERNS**
>
> Be careful when using this pattern in conjunction with the Async Chain of Persistence pattern. Because *all* events are persisted in that pattern, if no services are interested in the extensible event, it will remain in the queue forever and could cause issues with storage and memory in the broker. See "Conflicting Patterns" in Chapter 6 for some tips on how to coordinate these conflicting patterns.

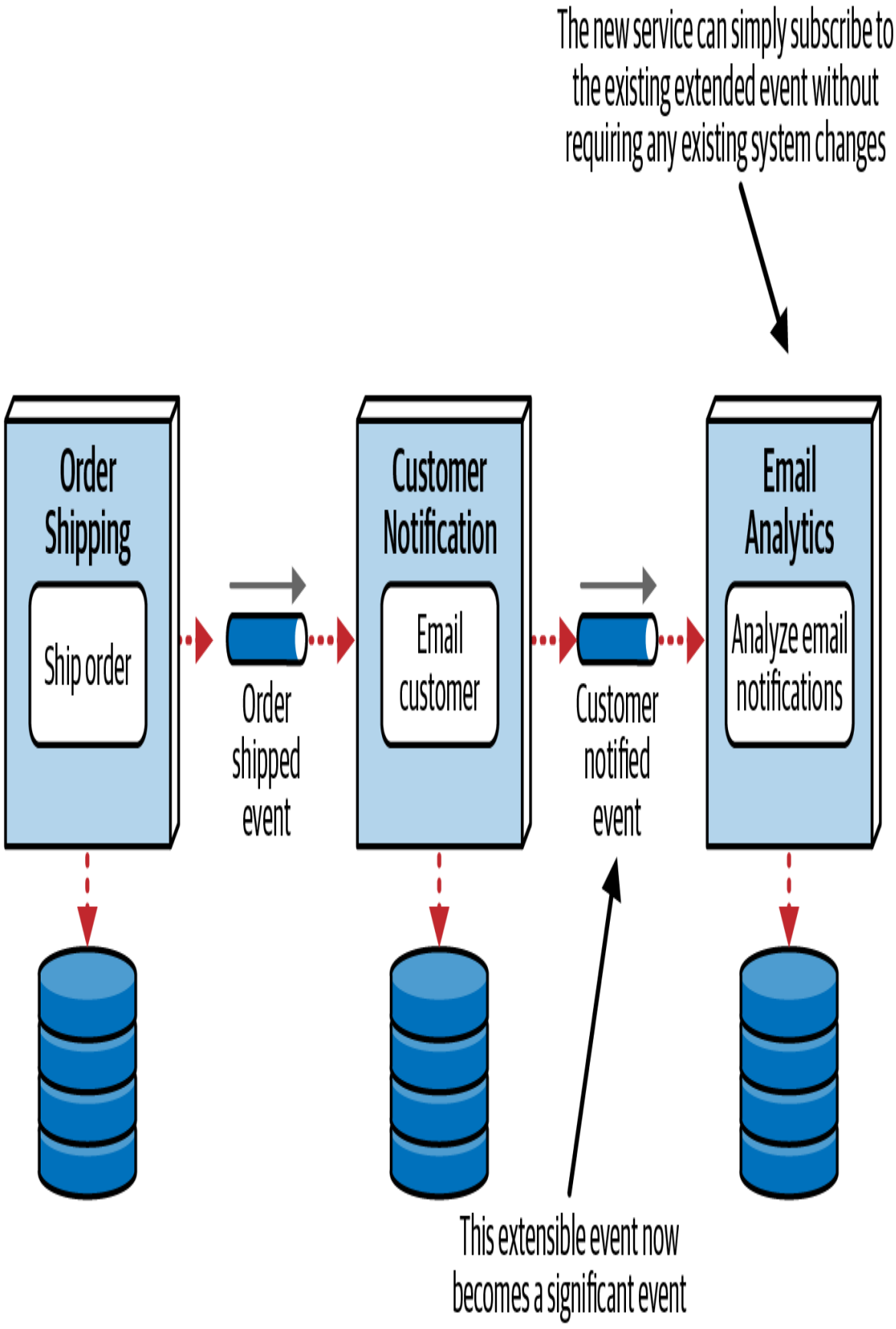

The new service can simply subscribe to the existing extended event without requiring any existing system changes
Order Shipping
Ship order
Order shipped event
Customer Notification
Email customer
Customer notified event
Email Analytics
Analyze email notifications
This extensible event now becomes a significant event

Figure 18-3. New services can easily be added to the system when using this pattern

Trade-Offs

Many architects using EDA follow the principle that no matter what, any state change in a service should be advertised. That's what this pattern is all about. However, architects should be aware of the trade-offs, however insignificant. We've outlined the pros and cons of this pattern in the following sections.

Pros

The primary advantage of this pattern is to provide evolvability by creating "hooks" into the system so future services can react to those events. This makes it easier for teams to add new services to react to those events.

Cons

Perhaps the biggest drawback of adopting this pattern is an increase in *event noise*, or the number of events triggered in the system, particularly when working with complex systems or systems with high volumes and lots of derived events.

Another drawback is the difficulty of predicting the data needs of future services and behavior. A service that publishes extensible events may need to be modified to send additional derived events or modified event-payload data based on a new behavior. For example, the `customer notified` extensible event might currently be sending only the customer ID and email address, but a new `Customer Analytics` service might need to know the order ID as well. This new behavior would require a change to the event payload contract in the `Customer Notification` service, which might negate the advantages of this pattern.

Comparison Chart

Table 18-2 summarizes the star ratings associated with applying this pattern. The Without Pattern column refers to not triggering events when no one is listening.

Table 18-2. Comparison chart for the Extensible Events pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Adaptability	★★★	★★★★★
Agility	★★★	★★★★★
Deployability	★★★	★★★★★
Evolvability	★★★	★★★★★
Maintainability	★★★	★★★★★
Testability	★★★	★★★★★
DEGRADED		
Bandwidth Usage	★★★★★	★★★★
Simplicity	★★★★★	★★★★

The following sections describe each of the capabilities found in Table 18-2.

Improved

These capabilities are improved when applying this pattern. These advantages start to disappear if the contract of the extensible event must change when adding new services.

Adaptability and Evolvability

> This pattern provides "hooks" in the system by triggering derived extensible events. New behavior can easily be attached to these hooks without requiring changes to the existing system, making it easier to extend the architecture.

Agility

> *Agility* is the ability to respond quickly to change. Applying this pattern reduces the changes needed in the existing system when adding new behavior, making it easier and faster to deploy new services that react to extensible events.

Deployability

> When using this pattern, only new services reacting to the extensible events need to be deployed, reducing the overall risk associated with deployment.

Maintainability

> Because services triggering extensible events generally don't need to be modified, systems are easier to maintain when adding new services that react to the extensible events.

Testability

> Because extensible events provide a "hook" into the system, fewer services need to change when adding new services, requiring less testing.

Degraded

The capabilities that are degraded when applying this pattern include:

Bandwidth Usage

> This pattern can slightly increase bandwidth usage by triggering events unnecessarily. This capability becomes more significant as the number of extensible events increases, particularly under high-volume loads.

Simplicity

> Applying this pattern adds complexity, due to having to create a new topic and event contract.

Influences

The only influence this pattern has is when teams also adopt the Async Chain of Persistence pattern (see Chapter 6). Teams should make sure extensible events are not persisted until they know that services will eventually react to them.

How to Govern This Pattern

Architects can use fitness functions to ensure that all services trigger at least one derived event after processing. These fitness functions could use static code analysis or logging and tracing. This governance technique is most effective if the code provides a standard way to publish events.

Using logging and tracing is most effective when the system uses a standard logging or tracing library. Not only does this ensure that derived events are always triggered from a service, it indicates *which ones* are triggered. This technique is also effective for knowing which derived events are extensible (and ignored by the system) and which are currently being consumed by other services.

Related Patterns, Antipatterns, and Pitfalls

Because this pattern is applied within an EDA context, it's related to several other EDA patterns described in this book:

Async Chain of Persistence pattern (see Chapter 6)

> When using the Extensible Events pattern, make sure events that aren't being processed are not persisted, or at least have a time-to-live value set to manage broker memory and storage. See "Conflicting Patterns" in Chapter 6 for tips.

Contextual Events pattern (see Chapter 12)

> A service might want to trigger *multiple* extensible events, depending on the context of the processing it just did. Because the Contextual Events pattern generally increases the number of events triggered in a system, using these patterns together could increase event noise.

Data-Based Event Contract pattern (see Chapter 14)

> Architects must decide how much data to include in the extensible event payload (its size). Using the Data-Based Event Contract pattern increases the data size of the events. Large payloads could accentuate some of the Extensible Events pattern's negative trade-offs, potentially impacting scalability, throughput, and responsiveness.

Key-Based Event Contract pattern (see Chapter 23)

> The Key-Based Event Contract pattern only contains the keys of the data entities in message payloads. This pattern is related in that architects must decide how much data to include in the extensible event payload.

Chapter 19. Front Controller Pattern

The Front Controller pattern (alternatively known as "Front Orchestrator") provides a way to add workflow state to choreographed workflows by adding it to the first-called service or event processor. This is a specialized version of the Cross-Cutting State pattern (see Chapter 13).

The Front Controller metadata appears in Table 19-1.

Table 19-1. Metadata for the Front Controller pattern

CATEGORY	METADATA
ALSO KNOWN AS:	Front Orchestrator
CONTEXT:	Event-Driven Architecture, Message-Based Architectures, Microservices, Workflow
IMPROVED:	Data Consistency, Data Integrity, Debuggability, Error Handling, Workflow
DEGRADED:	Adaptability, Availability, Domain Alignment, Elasticity, Evolvability, Fault Tolerance, Responsiveness, Reusability, Scalability, Throughput

How Do I Handle Mediation Without an Explicit Orchestrator?

Many architects like the Choreography pattern for distributed architectures (see Chapter 9) because it maps nicely to event-driven workflows. However, a common need architects have in those situations is to query the *state* of the workflow. For example, given a particular identifier, what is the state of the workflow concerning this item? In other words, in a complex, multistage workflow, how many event processors have already processed the item and how many are left?

The Front Controller pattern allows architects to implement a hybrid between orchestration and choreography, providing choreographed architectures with one of the benefits of having an orchestrator to maintain the state of the workflow (among other responsibilities) without implementing full orchestration. However, because each domain service must update the state, this pattern adds coupling between services that nominally would not be coupled.

Applying the Pattern

This pattern exists within the context of the Choreography pattern for event-driven architectures (EDAs). Its general topology appears in Figure 19-1.

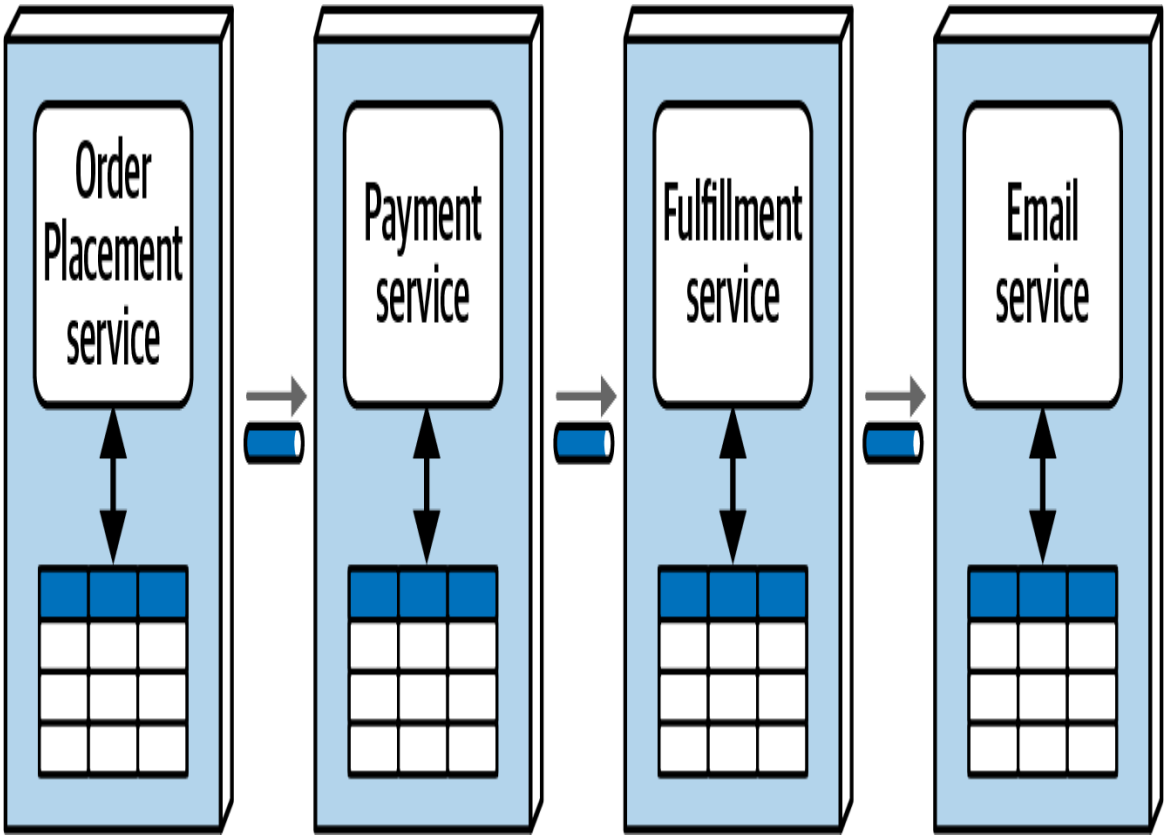


Figure 19-1. The general topology of Choreography, the context for the Front Controller pattern

The services shown in Figure 19-1 model a simple order processing workflow. The user interface generates an event, which is handled by the `OrderPlacement` service. `OrderPlacement` updates its internal state and issues an event destined for the `Payment` service. The workflow follows a similar pattern for the `Fulfillment` and `Email` services.

In a standard choreographed workflow, each service emits an event for which the subsequent event listens, as shown in Figure 19-2.

In this "happy path" scenario, each service responds to events within the system. However, the global state of the workflow isn't captured anywhere. If someone wanted to discover the current status of an in-process order, the only way to discover it would be by forensically querying each service or looking at log files.

While this is inconvenient in a workflow that's functioning normally, it's an even bigger problem when errors occur. Thus, many architects decide to keep state in the first-called service. This is a specialized version of the

Cross-Cutting State pattern, discussed in Chapter 13. The "front" in Front Controller is the first-called service in the workflow.

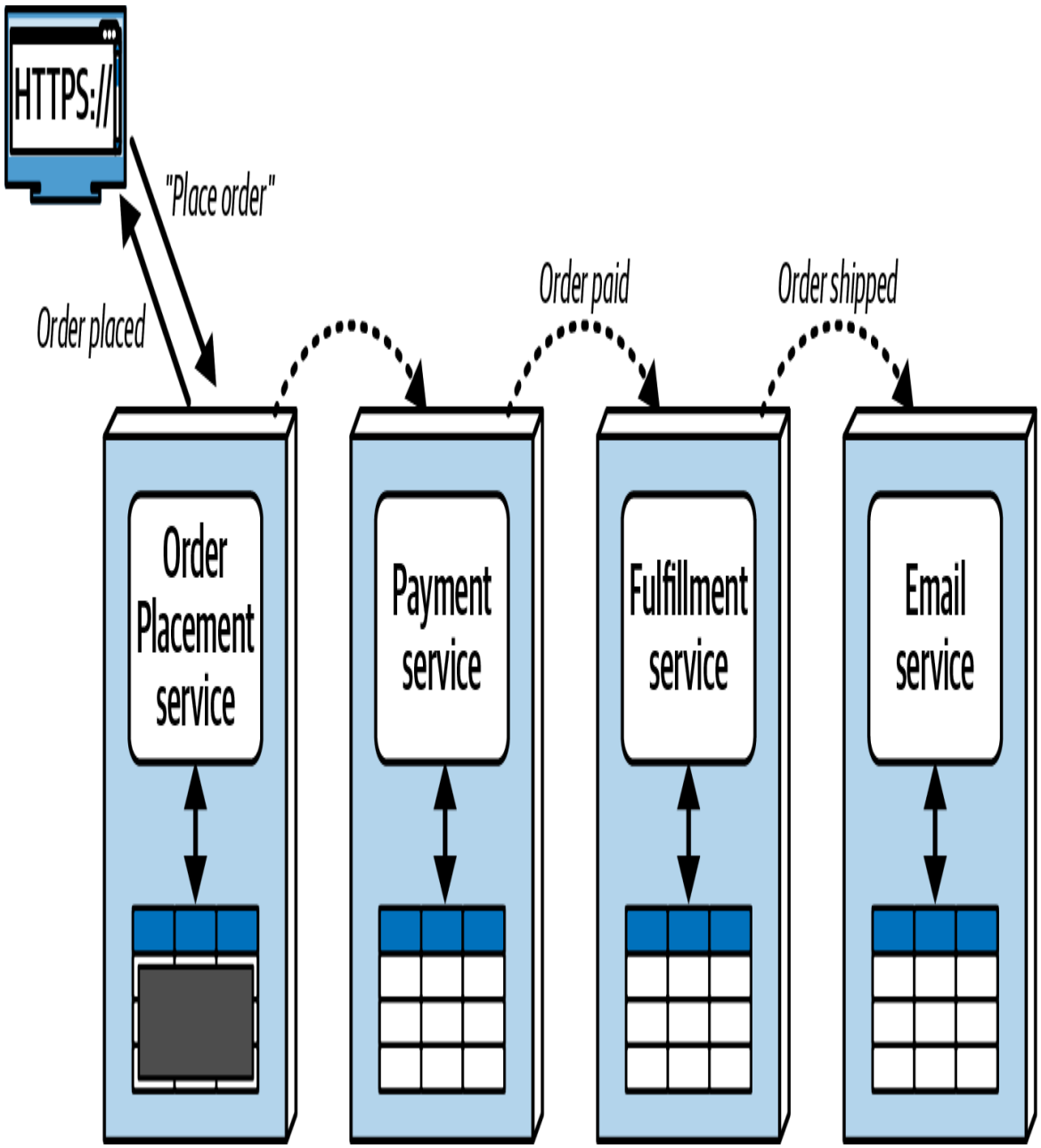


Figure 19-2. A standard workflow in Choreography

In Figure 19-3, each domain service that is triggered by events in turn generates a message to the `OrderPlacement` service to update its state. That service now contains two types of persistent information: the domain schema for orders *and* the state of the workflow. However, anyone can now

query the `OrderPlacement` service to find out the current state of any order.

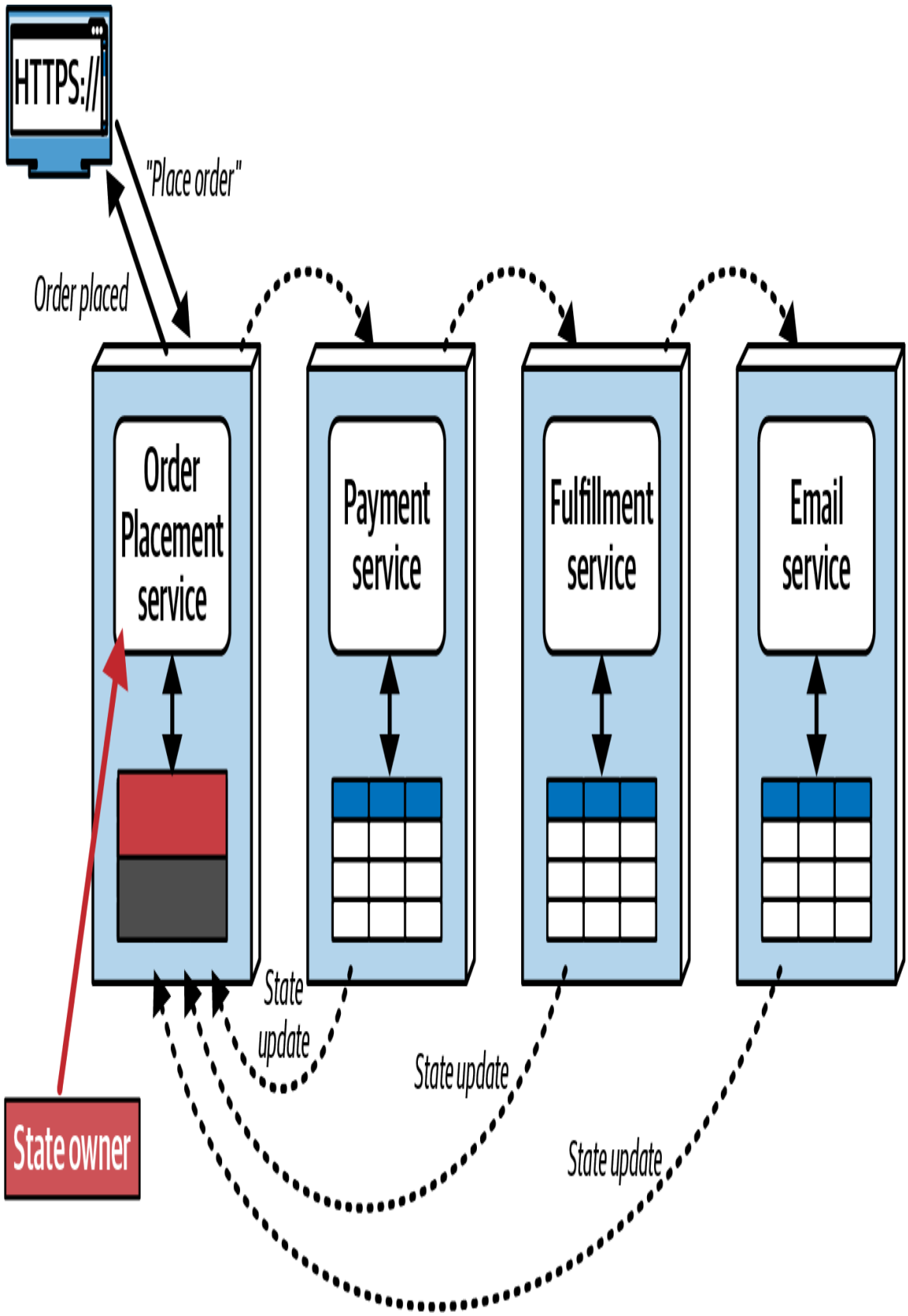


Figure 19-3. Using messages to update the state in the Front Controller

This pattern is a common mash-up of two patterns: Choreography (see Chapter 9) and Cross-Cutting State (see Chapter 13).

The Front Controller pattern basically adds behavior but no structure to an existing choreographed workflow, which changes the destination for error messages and other coordination, as illustrated in Example 19-1.

Example 19-1. Adding a Front Controller to a choreographed workflow

```
define event_processor controller
define event_processor domain1
define event_processor domain2
define event_processor domain3
define error_processor error
define system requester
message_queue = {SUCCESS, ERROR}

controller.listen_for INITIATING_EVENT
domain1.listen_for controller.SUCCESS, output STATE to controller
domain2.listen_for domain1.output via domain1.SUCCESS, output STATE
to controller
domain3.listen_for domain2.output via domain2.SUCCESS, output STATE
to controller
error.listen_for domain1.ERROR, domain2.ERROR, domain3.ERROR
   send ERROR state to controller
end error

requester.listen_for domain3.SUCCESS
requester.listen_for controller.ERROR
send requester.INITIATING_EVENT to controller
```

The code in Example 19-1 adds the `Controller` service to designate the first domain service. As in the normal Choreography pattern, the first domain service receives the initiating event, but in this case it's the `Controller` service. The other services are wired up as before, with the exception that each of them is designed to output its state back to the controller.

In typical Choreography, when the last-called service returns `SUCCESS`, each domain service sends any errors to a dedicated queue. In the Front Controller pattern, however, errors are returned to the controller so that it can restore state for the workflow and then send the error message back to

the requester. In general, events that used to go directly back to the requester now route back through the Front Controller.

Trade-Offs

This section discusses the trade-offs of the Front Controller pattern.

Pros

This pattern offers a clear advantage:

Maintains workflow state

> This is such a common problem in standard choreography that architects created a named pattern.

Cons

This pattern also carries a notable drawback:

Degrades coupling

> One of the advantages of Choreography is the lack of coupling between services, which allows for better scalability and related capabilities. However, state updates represent additional coupling, detracting from this advantage: when an architect changes a domain service, they might also have to change the Front Controller, and vice versa.

Comparison Chart

The trade-off analysis chart for the capabilities affected by this pattern appears in Table 19-2. Note that the comparison is between this pattern and "regular" Choreography, as this is a hybrid Orchestration pattern.

Table 19-2. Comparison chart for the Front Controller pattern

Capability	Front Controller	"Pure" Choreography
IMPROVED		
Data Consistency	★★★★★	★★
Data Integrity	★★★★★	★★
Debuggability	★★★★★	★★★
Error Handling	★★★★★	★★★
Workflow.	★★★★★	★★★
DEGRADED		
Adaptability	★★	★★★★★
Availability	★★	★★★★★
Domain Alignment	★★★	★★★★★
Elasticity	★★	★★★★★
Evolvability	★★★★	★★★★★
Fault Tolerance	★★	★★★★★
Responsiveness	★★	★★★★★

Capability	Front Controller	"Pure" Choreography
Reusability	★★★	★★★★★
Scalability	★★	★★★★★
Throughput	★★	★★★★★

The following sections describe each of the capabilities found in Table 19-2.

Improved

The following capabilities are generally improved when using this pattern:

Data Consistency

One of the primary purposes of this pattern is to provide a workflow to help data consistency, which ensures that data across the system is in sync and consistent across databases and tables. This pattern provides ways to restore just enough state to follow the semantics of the workflow without unnecessary coupling.

Data Integrity

Like the Cross-Cutting State pattern, one of the Front Controller pattern's primary purposes is to preserve data integrity: does each datum correctly reflect what it's supposed to?

Debuggability

Having a known state owner allows architects an easy and obvious place to aid in debugging, as the Front Controller "knows" the state of the workflow and other critical cross-service details.

Error Handling

A common source of errors in workflows derives from the lack of workflow state, making this pattern attractive for architects who need to guard against such errors.

Workflow

Workflows often have shared state, and this pattern provides a well-defined way to manage cross-service state.

Degraded

The following capabilities are generally degraded when using this pattern.

Adaptability

Adaptability requires more effort in an orchestrated workflow because the essence of adaptability requires adding new behavior while preserving the existing behavior. This requires that the architect change both domain services and the orchestrator, which in this case also has domain responsibilities as the first-called service.

Availability

Availability is degraded because the state holder (the Front Controller) must be available in addition to the domain services, increasing both the number of services that must be available for success and the workflow's reliance on the Front Controller.

Domain Alignment

From an individual service standpoint, externalizing state in the Front Controller helps domain alignment: when architects don't use this pattern, incidental state related to cross-service data interactions must reside within the domain services, making them more complex. However, from a workflow standpoint, this pattern degrades domain alignment because the first-called service takes on a second, nondomain responsibility — holding workflow state — and every other service must now coordinate with it. That cross-service coupling outweighs the

per-service benefit, so we mark this pattern as slightly degrading domain alignment.

Elasticity

Elasticity is degraded in this instance of an orchestrator as the Front Controller because even more startup agility relies on the Front Controller, both from a domain standpoint and workflow standpoint. Mean Time To Start will be highly critical for the Front Controller component.

Evolvability

Keeping workflow state in a Front Controller increases coupling over Choreography in that each service must provide state updates, which in turn degrades the general evolvability of the services.

Fault Tolerance

Fault tolerance is degraded with the Front Controller because the workflow becomes sensitive to a single service (the Front Controller) to be available for workflows to function.

Responsiveness

Because workflows using a Front Controller utilize an orchestrator to send and receive messages, it's inherently a bottleneck. Even if architects build redundant Front Controllers (which improves reliability), the fundamental communication within this pattern flows through the orchestrator. Thus, communication overhead limits responsiveness.

Reusability

Like evolvability, reusability generally degrades with Orchestration for two reasons: the increased coupling represented by the orchestrator and the typical use of messages (rather than events) in Orchestration, which

hinders reusability. The Front Controller is a hybrid of Choreography and Orchestration, taking on some of the disadvantages of each.

Scalability

The funnel-like topology of this architecture limits its overall scalability; all workflow must go through the Front Controller.

Throughput

Similar to scalability, the requirement that all workflows must coordinate with the Front Controller limits the throughput of the workflow to the throughput of the Front Controller.

This pattern is common when teams mostly want the high degree of decoupling inherent in Choreography but need a consistent place to handle state; the first-called service makes a convenient place for it.

Influences

The influences for this pattern are the same as for Orchestration (in Chapter 27), as this is a specialized version of that pattern.

How to Govern This Pattern

Governing this pattern depends on that important feedback an architect may need about malfunction. For example, if the architect is using Front Controller to manage state, they might build fitness functions to ensure that behavior.

An easy way to handle this problem in microservice architectures is to create a container-level fitness function that ensures that each service emits state information over a particular port, to which the Front Controller listens. This means that all services generate a consistent level of log information. To validate that, the architect could write an architectural

fitness function that aggregates the log information to ensure state. This fitness function can be part of a *service mesh*, a common way that microservice architectures share operational coupling like monitoring, logging, and so on.

Related Patterns, Antipatterns, and Pitfalls

This pattern is a specialized variant of two primary patterns:

Cross-Cutting State pattern (see Chapter 13)

> Front Controller is a specialized version of the Cross-Cutting State pattern, where the first-called service holds the external state (in this case, the state of the workflow).

Choreography pattern (see Chapter 9)

> Front Controller is a common addition to the Choreography pattern, augmenting it with one aspect of orchestration.

This pattern is also related to:

Compensating Updates pattern (see Chapter 11)

> If the workflow in question also contains transactional state, then the Compensating Updates pattern provides a way to mimic the atomicity of ACID transactions (see Chapter 37 for a full definition). Normally, the Compensating Updates pattern is more difficult to use in choreographed architectures. The Front Controller pattern makes it easier by specifying a clear state owner to handle restoring state.

Orchestration pattern (see Chapter 27)

> In many ways, Orchestration is the opposite of Choreography; however, they're both ways to implement coordination between services or event processors.

Chapter 20. Frozen Caveman Pitfall

Architects perform diverse analytical, strategic, and collaborative tasks — from making technology decisions and conducting trade-off analyses, to evaluating requirements and assessing risks. The Frozen Caveman pitfall emerges when architects rely exclusively on their past experiences to inform future decisions. The consequences range from implementing an architecture that's misaligned with the problem domain, to introducing business-critical risks that damage brand reputation and customer trust.

Table 20-1 summarizes the metadata for the Frozen Caveman pitfall. While adopting this pitfall still brings trade-offs, no architectural capabilities are improved. Many are degraded.

Table 20-1. Metadata for the Frozen Caveman pitfall

CATEGORY	METADATA
CONTEXT:	Decision Making
IMPROVED:	None
DEGRADED:	Adaptability, Agility, Availability, Cost, Debuggability, Deployability, Elasticity, Error Handling, Evolvability, Fault Tolerance, Flexibility, Maintainability, Modularity, Responsiveness, Scalability, Simplicity, Testability, Throughput, Workflow

Should I Only Base My Decisions on Past Experiences?

Architects are often key players in the decisions that software development teams make during the lifecycle of any project. Many of these decisions have long-lasting effects and a large blast radius, and thus need to be made very deliberately. In short, architects must be able to gauge (and often dismiss) their own biases, and discern between genuine warnings and ghosts in the closet.

The Frozen Caveman pitfall describes architects who let their past experiences — and *only* their past experiences — dictate every decision they make. The name comes from a trope best captured in the 1992 film *Encino Man*, in which Brendan Fraser plays a caveman who is frozen alive in ice until two high-school students find him, thaw him out, and introduce him to modern California. Much like a frozen caveman, these architects are stuck in the past.

So how does this happen? One reason is that architects, like all other human beings on the planet, have had both good and bad experiences that color their view of the world and affect future decisions — and architects make a lot of decisions. Experiences teach us many lessons, some of which are well worth carrying forward — for example, looking both ways before crossing a street is wise. However, a child who is burned once might learn not simply to be cautious about fire, but to dread it so much that they avoid even using it for cooking or campfires.

Imagine an architect who has a bad experience using asynchronous processing with events and messages, and vows to only use REST from now on. While REST *may* fit some scenarios, this architect is ignoring the benefits of asynchronous processing and will probably end up delivering a system that doesn't meet all of the architectural capabilities. Conversely, imagine another architect who has successfully delivered multiple projects using Java Server Pages (JSP). When starting a new project, they "play it safe" and stick with JSP, simply ignoring modern templating engines like Thymeleaf or even client-side frameworks like Angular or Vue.js.

It's not just technology choices, either. Consider a third architect working for a retail corporation who got burned one December when stores lost network connectivity and all point-of-sale systems stopped working at the height of holiday shopping. Ever since then, this architect prioritizes network connectivity above all else — even though the technology landscape has shifted significantly, as have the business's concerns.

Traditionally, the "floor above" the role of senior engineer has always led to the role of architect.[1] For many software engineers, becoming an architect is the obvious next rung in the corporate ladder. Outside of the corporate structure, it's not unusual for architects to have a background working as engineers in value-aligned, domain-specific teams — a role that rewards specialization. By contrast, the role of architect is much broader and requires understanding a wide breadth of technologies, techniques, and processes. Architects who have trouble switching from specialist to generalist, and from tactical to strategic, often fall prey to this pitfall.

Why Is This a Pitfall?

The effects of architectural decisions can ripple out very widely, and if those decisions are deemed unfit once implemented, undoing them tends to be very expensive. When architects base their decisions solely on past experience, they bypass the critical discipline of trade-off analysis. With their eyes glued to the rearview mirror, it's simply impossible for them to consider the current context in which they're *supposed to* be making decisions. This could mean failing to evaluate modern tools, techniques, or processes against the requirements and assess their pros and cons, or failing to evaluate the current landscape for risks and pain points. Regardless of the kind of decision at hand, relying only on the past introduces architectural and technical debt. It ends with building a system that doesn't behave as expected or exhibit the capabilities it needs — leading to revenue loss, dampened customer expectations, brand erosion, poor developer experience, and high levels of attrition.

How to Avoid This Pitfall

While it's important not to give prior experiences too much weight, it's not a good idea to dismiss them entirely, either. Experiences are simply distilled lessons and are often valuable guides. Their role is to *inform* architects' decisions without clouding their judgment or replacing rigorous analysis of the current context and requirements.

While technical *depth* is about knowing everything about a particular language, platform, framework, or a product, *breadth* is all about knowing a little bit about a lot of things. Great architects have significant technical breadth, and this helps them maintain their objectivity.

The knowledge pyramid (shown in Figure 20-1) illustrates the difference. All knowledge can be broken down into three categories — *stuff you know*, *stuff you know you don't know*, and *stuff you don't know you don't know*.

At the top of the pyramid is the architect's knowledge base — namely, the "stuff you know." These are the tools, technologies, frameworks, runtimes, platforms, and processes they've learned and in which they're competent or even expert. Naturally, this category is represented by the smallest area of the pyramid, since no one can be an expert at everything.

The area in the middle, "stuff you know you don't know," includes things the architect might have heard of, maybe discussed at a conference or a user group, or perhaps even used sparingly in a hobby project. This area is much larger and includes things in their peripheral vision, but which they don't claim to understand or leverage properly, let alone know well.

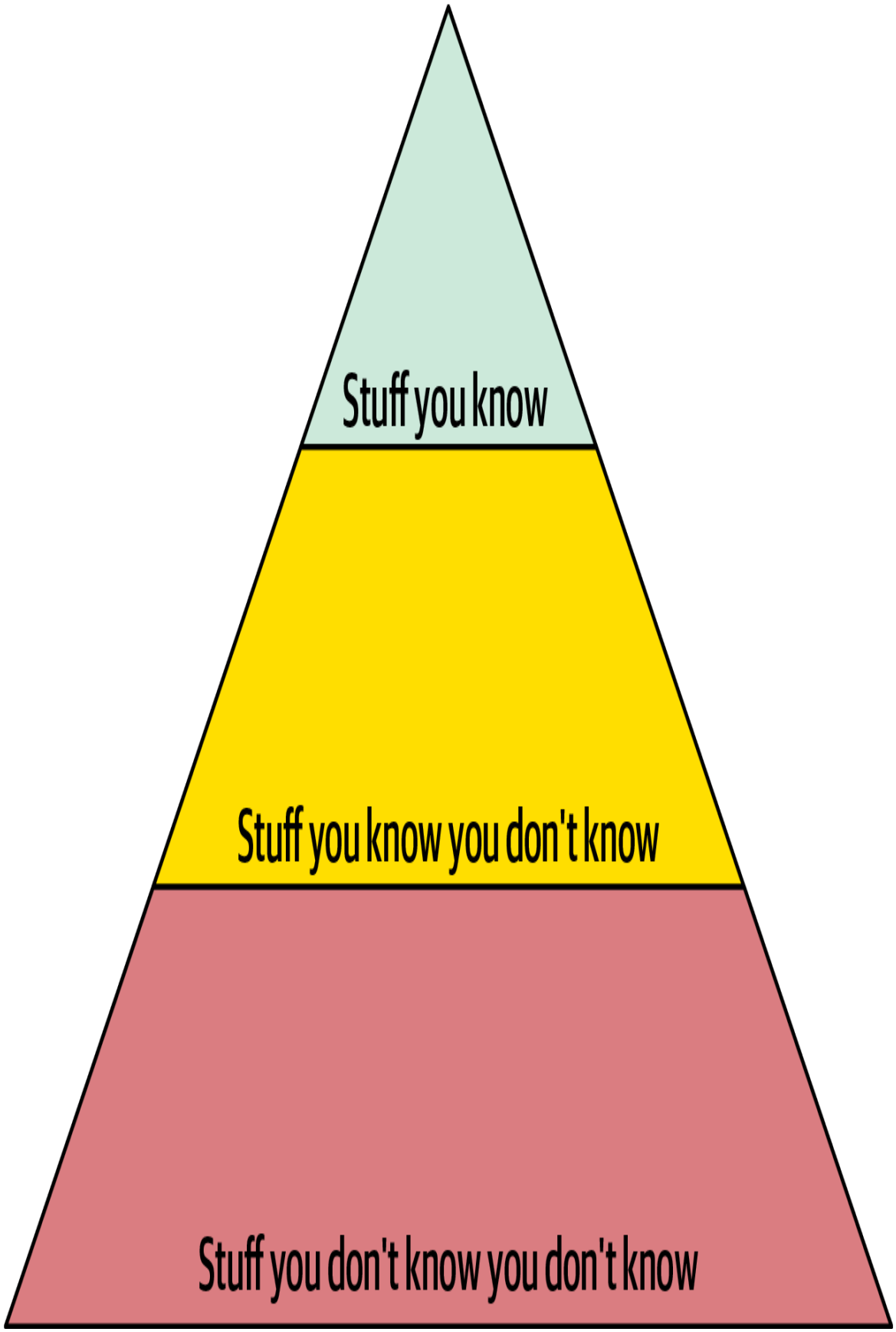

Stuff you know
Stuff you know you don't know
Stuff you don't know you don't know

Figure 20-1. The knowledge pyramid

The bottom of the pyramid — the "unknown unknowns" — is the largest and reflects true gaps in the architect's knowledge. These technologies often create "I wish I had known!" moments: they could have been *the* perfect solution for a problem — if only the architect had been aware of them. This area is also the fastest growing, given the "hockey stick" rates of recent technological growth.

It's important to understand how to work with the knowledge pyramid at different career stages. Early on, a junior developer's primary objective should be to expand the top area — the "stuff they know" — learning some things really well and gaining technical depth.

But as they progress, with an eye toward becoming an architect, they must transition to expanding the middle section of the knowledge pyramid: they should work on knowing a little about a lot of things rather than knowing everything about a few things, thereby expanding their technical breadth, as illustrated in Figure 20-2. It's better for an architect to know the basics of five different datastore solutions and their trade-offs than to know all the working details of just one. This can be a real struggle for those who've spent years honing their technical skills, because it asks them to sacrifice some of that hard-won expertise in favor of broadening their knowledge base.

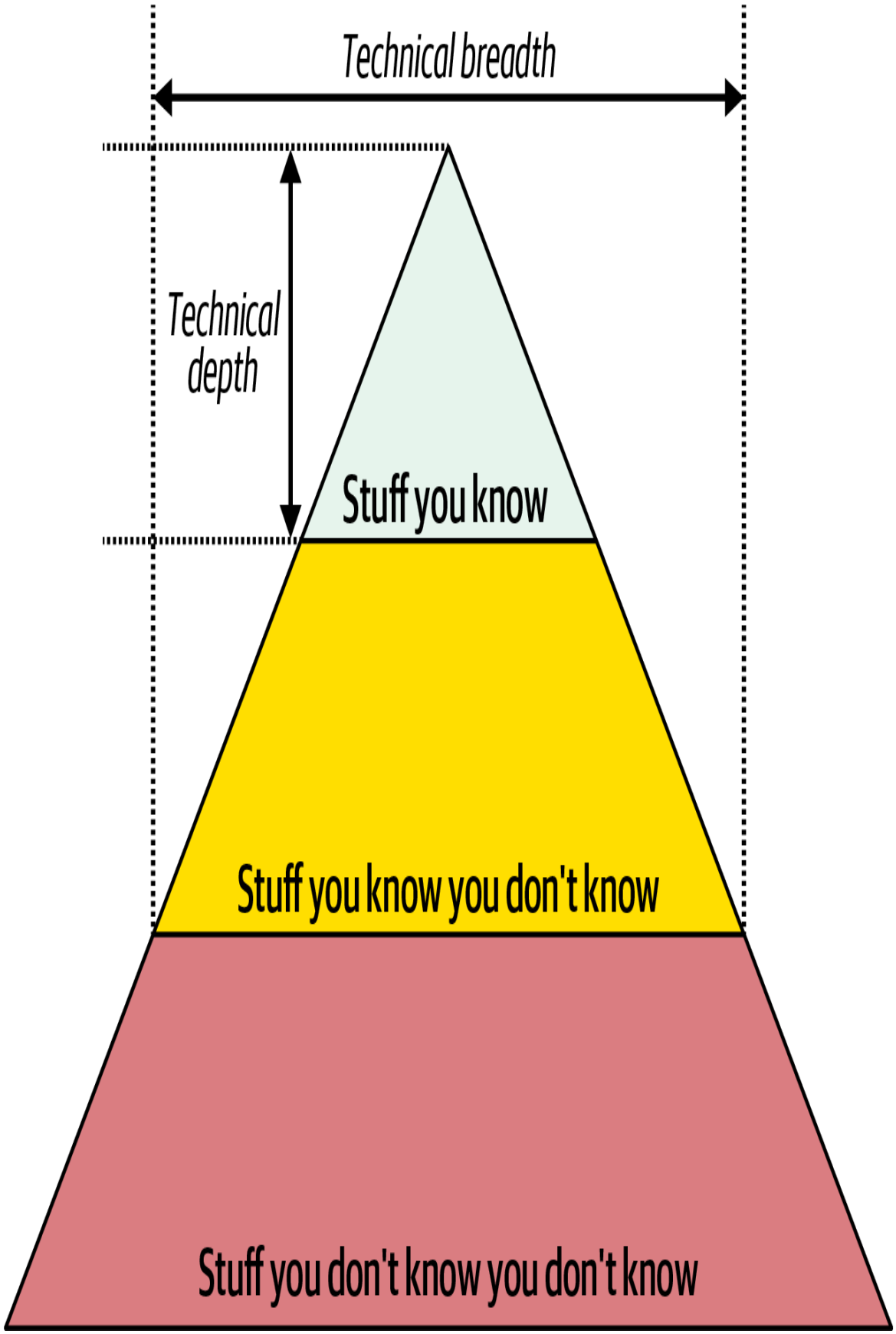

Technical breadth
Technical depth
Stuff you know
Stuff you know you don't know
Stuff you don't know you don't know

Figure 20-2. Technical breadth versus depth

Architects fall victim to the Frozen Caveman pitfall when they believe they *can* maintain the same technical depth they did as developers, but across a wide variety of areas. They rarely succeed, but since they never really did broaden their horizons, this can give them a false sense of assurance — and they often end up reaching for technologies that aren't really suited for the problem at hand.

Some architects believe that outdated technologies they spent years mastering are still relevant, simply because they've never put in the effort to broaden *and* modernize their portfolios.

The antidote to this pattern is to appreciate the knowledge pyramid and what it represents. As developers transition into the role of architect, they should embrace the notion that getting down to the wire, so to speak, is a luxury architects can no longer afford. With passing knowledge across diverse technologies, an architect must shift from tactical execution to strategic thinking — focusing on trade-offs, long-term implications, and alignment with business goals rather than just implementation details.

So how can architects go about expanding their technical breadth? Getting out of the weeds is a good start. Let senior developers and other experienced folks deal with daily technical issues, pair with other team members, and do the more critical code reviews. This frees up some time for the architect to broaden their horizons.

Staying current on new trends, technologies, and buzzwords can be demanding. One technique is the 20-minute rule: every day, devote at least 20 minutes to learning something new or probing deeper into a specific topic. Days get busy quickly, so it's best to do this early in the morning or later in the evening, when the world has gone quiet. Read sources like InfoQ and Thoughtworks Technology Radar, and follow the blogs, Substacks, and social media of notable technology companies and technologists.

Developing a Personal Radar

The software landscape is always evolving, and it's not impervious to "bubbles" — where a particular technology arrives, sucks up all the oxygen in the room, and then is never heard of again. At the same time, other, less shiny technologies forever change how software is developed. It's easy to get caught up in the hype. Architects often find themselves flitting from one technology to the next — barely registering an understanding of what each technology might be able to do for their problem domains.

One technique to help keep track of technologies to phase out, better understand, or keep an eye on (with the intent of digging deeper as their adoption grows) is to build a personal "technology radar," based on the Thoughtworks Technology Radar, a sample of which is shown in Figure 20-3.

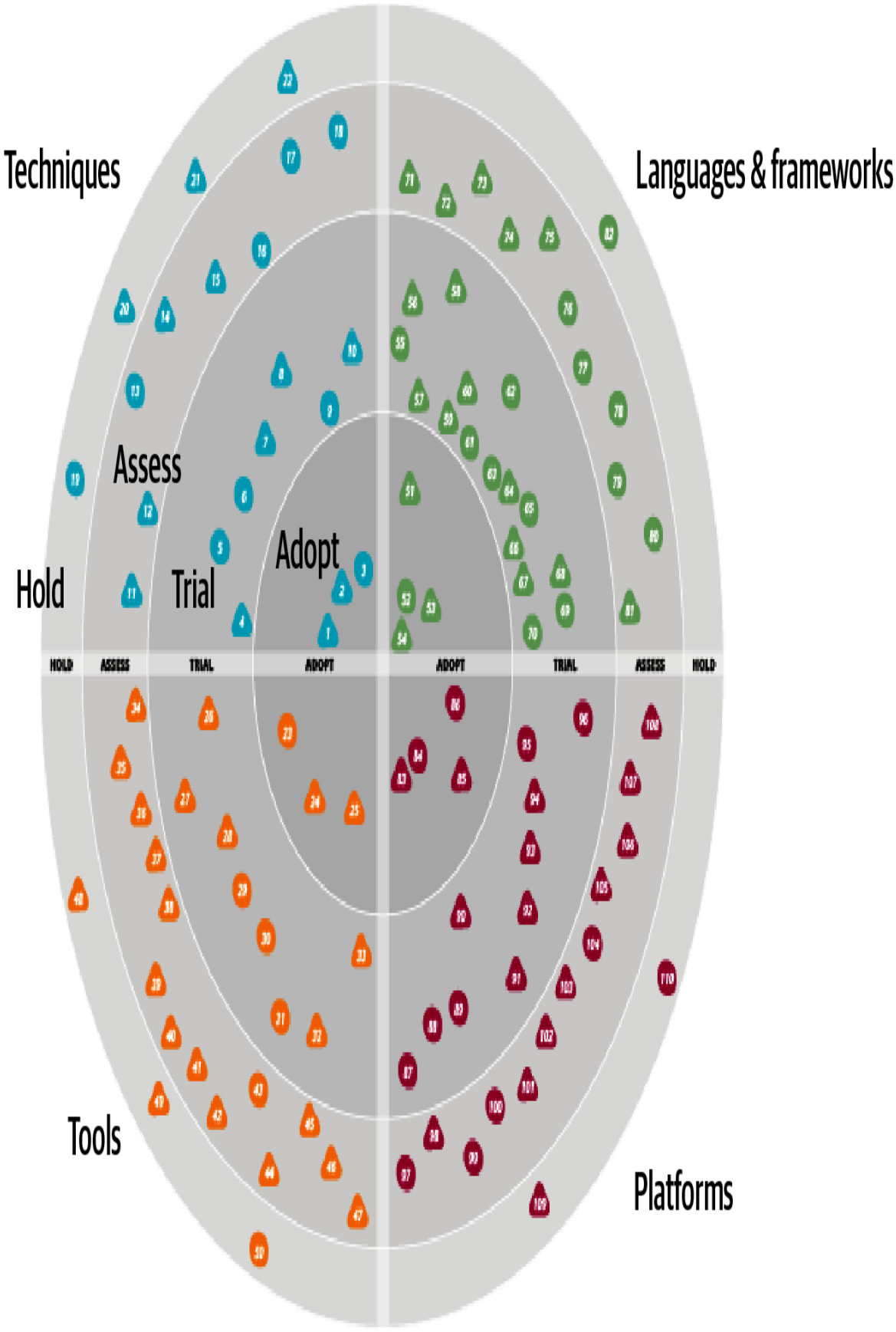

Techniques
Languages & frameworks
Assess
Adopt
Hold
Trial
Tools
Platforms
HOLD
ASSESS
TRIAL
ADOPT
ADOPT
TRIAL
ASSESS
HOLD

Figure 20-3. A sample Thoughtworks Technology Radar

The Thoughtworks Technology Radar comprises four quadrants: Tools, Languages and Frameworks, Techniques, and Platforms. Each quadrant has four rings:

Hold

> This represents things that might be at their end of life (EOL), or close to it. It's a list of items that should *not* be leveraged for new projects.

Assess

> This ring represents things that an architect might want to explore in the future. This ring acts as a gentle reminder of areas for further research and helps avoid "shiny object" syndrome.

Trial

> This ring represents tools and techniques that definitely seem to be getting traction and are consequently worth pursuing if they fit the problem domain. Assuming they are a good fit, the architect might consider planning a pilot or proof of concept, perhaps targeting a low-risk project, to perform a serious trade-off analysis.

Adopt

> This ring represents things that have proven useful and are being adopted widely.

Build your own Radar is an open source project released by Thoughtworks that lets architects build their very own radar — specific to their interests — or map out the existing stack within their organization, to help them have serious conversations about what needs investment and what needs to retire. Thoughtworks releases a new version every six months, which is a good cadence to revisit a personalized version — if for no other reason than to ensure that what was captured the last time around is still relevant.

Another benefit of building a custom radar is that it helps tweak one's intuition. Say an architect got really excited about an emerging technology and invested time and effort in it, but was disappointed when the hype dissipated quickly and the technology failed. The radar can help the architect do an informal retrospective — a look in the mirror, so to speak — to highlight how their intuition led them to a dead end. Over time, intuition becomes an architect's best friend. It prevents them from being swayed by every shift in the technological landscape or chasing trends that don't align with the organization's goals and problem domains.

Trade-Offs

The Frozen Caveman pitfall, while it is indeed a pitfall, can in some cases bring positives. But over time, these benefits fade away, since the central tenet here is the tendency of an architect (or a team) to *reject* new technologies, tools, and processes because of their prior, often successful experiences in the past.

Pros

If an architect has had a lot of success with a particular technology or tool in the past, they can often demonstrate deep expertise with it, meaning that not only do they understand how to wield it well, they also know the edge cases and the gotchas. This experience can help increase the predictability — be that in terms of how the architecture will look, better time/cost estimation, and having some patterns they can reuse.

A secondary, often tangential benefit is that the architect demonstrates a certain level of conservatism — and their unwillingness to move on *may* end up saving them from jumping onto every technology hype train that arrives.

Of course, all these benefits are time boxed — because technology, tools, processes, and consequently software architecture move on, so while this

pitfall *may* offer some upsides in the short term, they're rather detrimental to all aspects of software architecture in the long term.

Cons

The Frozen Caveman *is* a pitfall, and we have categorized it so because it comes with rather serious downsides. Right off the bat, this pitfall introduces technical debt, which in the true nature of debt, must eventually be paid off. This can stifle innovation in the present, since the architect could find better solutions in reach today, and it worsens in the long term as this debt compounds. Legacy technology can be harder to integrate with modern systems, often has known common vulnerabilities and exposures (CVEs) with no patching support, and may lack the performance characteristics and scalability needed for modern workloads, all of which leads to increasing maintenance cost, and exposing the firm to security and potential compliance/regulatory risks, not to mention brand erosion.

On the personnel side, few want to work on legacy technologies, and projects like these not only see a brain drain, but also find it hard to onboard the kind of talent that an organization would like to have on hand. Those who continue to remain on the project simply feel stuck, leading to a lack of morale and motivation.

Comparison Chart

Table 20-2 summarizes the capabilities trade-offs associated with applying this pitfall, as compared to when an architect is willing to do all that's necessary to expand their knowledge, stay current with technology, and perform rigorous trade-off analysis.

Table 20-2. Comparison chart for the Frozen Caveman pitfall

Capability	Adopting Pitfall	Avoiding Pitfall
IMPROVED		
None	---	---
DEGRADED		
Adaptability	★	★★★★★
Agility	★	★★★★★
Availability	★	★★★★★
Cost	$$$	$
Debuggability	★	★★★★★
Deployability	★	★★★★★
Elasticity	★	★★★★★
Error Handling	★	★★★★★
Evolvability	★	★★★★★
Fault Tolerance	★	★★★★★

Capability	Adopting Pitfall	Avoiding Pitfall
Flexibility	★	★★★★★
Maintainability	★	★★★★★
Modularity	★	★★★★★
Responsiveness	★	★★★★★
Scalability	★	★★★★★
Simplicity	★	★★★★★
Testability	★	★★★★★
Throughput	★	★★★★★
Workflow	★	★★★★★

The following sections describe each of the capabilities found in Table 20-2.

Improved

This particular pitfall improves no capabilities — it only degrades them, with severity increasing as time goes by.

Degraded

When architects fall into the Frozen Caveman pitfall, nearly every architectural capability suffers degradation. The following sections explain how and why these capabilities are impacted, organized by category.

Process capabilities

The Frozen Caveman pitfall degrades the following process capabilities:

Adaptability, Agility, Debuggability, Deployability, Error Handling, Evolvability, Flexibility, Maintainability, Modularity, Simplicity, Testability, and Workflow

> Older technologies may enforce tighter coupling, lack testing and mocking support, as well as may lack support for observability tooling — in effect, are resistant to change and integration with modern tooling — all affecting capabilities like adaptability, debuggability, error handling, evolvability, flexibility, maintainability, modularity, simplicity, and testability. They often are not compatible with modern engineering practices like focus on build tooling (which are often focused on fast build times) and CI/CD integration, affecting deployability. Agility is a composite of deployability, modularity, and testability — if those three capabilities suffer, agility will suffer as well.
>
> Finally, legacy technologies may also be at EOL, with limited support in the form of patching, which decreases security.

Operational capabilities

The Frozen Caveman pitfall degrades the following operational capabilities:

Availability, Elasticity, Fault Tolerance, Responsiveness, Throughput, and Scalability

> Older technologies were often designed with different scale and operational requirements. Consequently, they may lack support for built-in retry logic, circuit breakers, or modern resilience patterns that can affect availability and fault tolerance. Furthermore, these technologies may not support capabilities like auto-scaling, limiting their ability to support responsiveness, throughput, and scalability needs. Lastly, older technologies did not focus heavily on metrics like Mean Time To Start — affecting capabilities like scalability. Cloud-native capabilities like auto-scaling and elastic resource management

are typically absent, limiting the system's ability to handle varying loads efficiently.

Business impact

The Frozen Caveman pitfall also impacts business in the following ways:

Cost, Workflow

Over time, the subsequent brain-drain that efforts with legacy technologies are often confronted with, alongside the difficulty in hiring, compound the lack of expertise, which means that everything gets more difficult. Integrating legacy into modern workflows and integrations starts to require workarounds. Velocity starts to slow and debt continues to grow, all meaning that cost starts to grow exponentially.

Influences

No external factors influence the trade-offs discussed in the prior section.

Related Patterns, Antipatterns, and Pitfalls

This pitfall does not relate to any other patterns, antipatterns, or pitfalls listed in this book. It's more about architect behavior and process as opposed to software architecture, though it does influence software architecture.

---

[1] This is changing in many organizations with the introduction of individual contributor roles like staff engineer.

Chapter 21. Grains of Sand Pitfall

Picture walking barefoot on an ocean beach on a hot summer day, the soft, warm sand glistening in the sun. Sounds like paradise — except this isn't a vacation, just another tough day at the office dealing with thousands of fine-grained microservices. The Grains of Sand pitfall is about making microservices *too small*, like grains of sand on a beach.

Table 21-1 summarizes the metadata for the Grains of Sand pitfall. While adopting this pitfall does involve trade-offs, no architectural capabilities are improved. Many are degraded.

Table 21-1. Metadata for the Grains of Sand pitfall

CATEGORY	METADATA
CONTEXT:	Microservices, Modularity, Service Granularity
IMPROVED:	None
DEGRADED:	Adaptability, Cost, Data Consistency, Data Integrity, Initial Time to Market, Simplicity

Are My Services Too Small?

Microservices, by definition, are meant to be fine-grained *single-purpose* services. Unfortunately, what constitutes "single-purpose" is rather subjective. For example, assume that a system can notify customers through email, SMS texting, or postal letter. Some teams might create a single

microservice to handle notifying a customer (regardless of the notification type); other teams might create three separate microservices, one for each notification type. Knowing which of these is the correct choice is one of the biggest challenges of microservices.

The Grains of Sand pitfall occurs when teams make all their microservices *too small*. This chapter shows why this is a pitfall, and how to avoid this common trap.

Why Is This a Pitfall?

The term *service granularity* describes the size of a service. That's not necessarily its *physical* size, but rather its *behavior* — what the service *does*. One of the first steps in getting microservice architectures to work correctly is getting service granularity right.

To better understand why making microservices too small is a pitfall, it's necessary to first understand why the concept of *bounded contexts* is so critical for the microservice architectural style. This concept, which borrows from Domain-Driven Design, primarily defines ownership and access to data (as well as other artifacts, such as message brokers and distributed caches). In microservices, each service owns its own data. That means it's the only service that can access that data, regardless of whether the data resides in a separate physical database or a grouping of tables in a separate schema. For example, in the system depicted in Figure 21-1, the `Placement` service can't directly access the `Inventory` service's data because it's contained within a bounded context (represented by the surrounding dotted line). Rather, the `Placement` service must ask the `Inventory` service for the data it needs (in this case, the current inventory for the items being ordered).

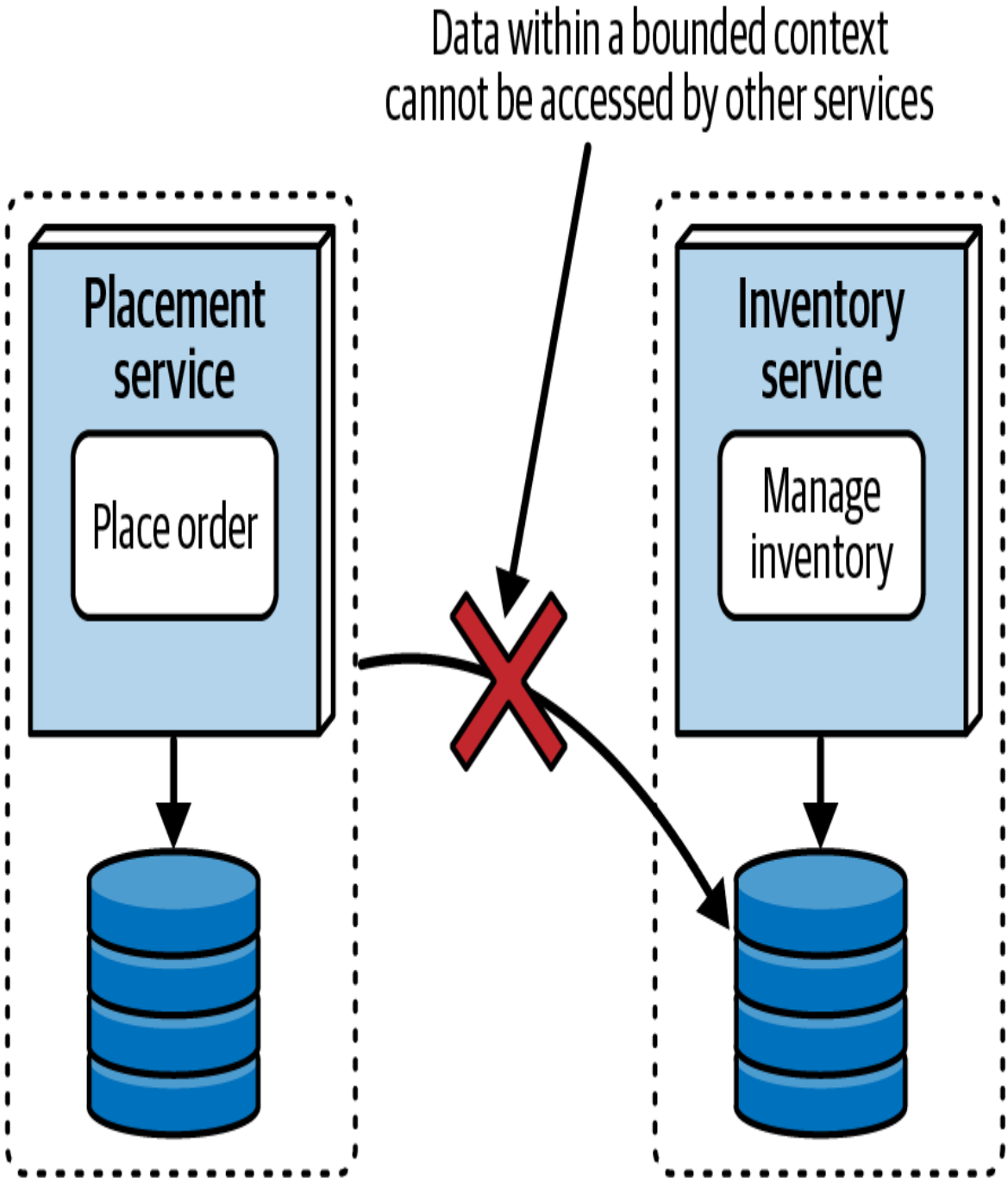


Figure 21-1. Services outside of a bounded context can't access data within that bounded context directly

A microservice's bounded context hides data from other services outside of that context, which means those data structures can change without impacting other services in the system. This is particularly important in a microservices architecture, due to the services' fine-grained nature and sheer numbers (typically hundreds or even thousands). Imagine a

microservices architecture without bounded contexts: dropping a column in a customer table might break dozens, if not hundreds, of microservices. It would be an unmanageable mess, creating the most brittle architecture in the world.

Making microservices too small forces architects to break the data into small pieces to form bounded contexts with those services. Getting service granularity right means getting *data granularity* right — a challenging task, particularly since other services might need that data as well.

Modeling fine-grained data is challenging because data usually has relationships, such as foreign-key constraints and *semantic relationships*, or groups of tables that form a single domain concept or entity. For example, consider the Sysops Squad trouble ticket system described in Chapter 4, where customers report issues with electronic equipment they've purchased. When a customer enters a ticket into the system, the customer-facing experts in the field — that's the Sysops Squad — come to their home or office to fix the problem. A data architect might choose to model the domain concept of a `Ticket Entry` through four separate database tables, as shown in Figure 21-2:

`ticket_entry`

: Stores basic information such as the ticket ID, date entered, customer information, and product information.

`ticket_issue`

: Stores a text-based description of the issue the customer is facing, as well as error codes and other relevant information.

`ticket_status`

: Stores the current status of the ticket, such as *entered*, *pending*, *assigned*, *in progress*, and *completed*.

`ticket_skills`

Stores the specific expert skills required, based on the product and nature of the reported issue.

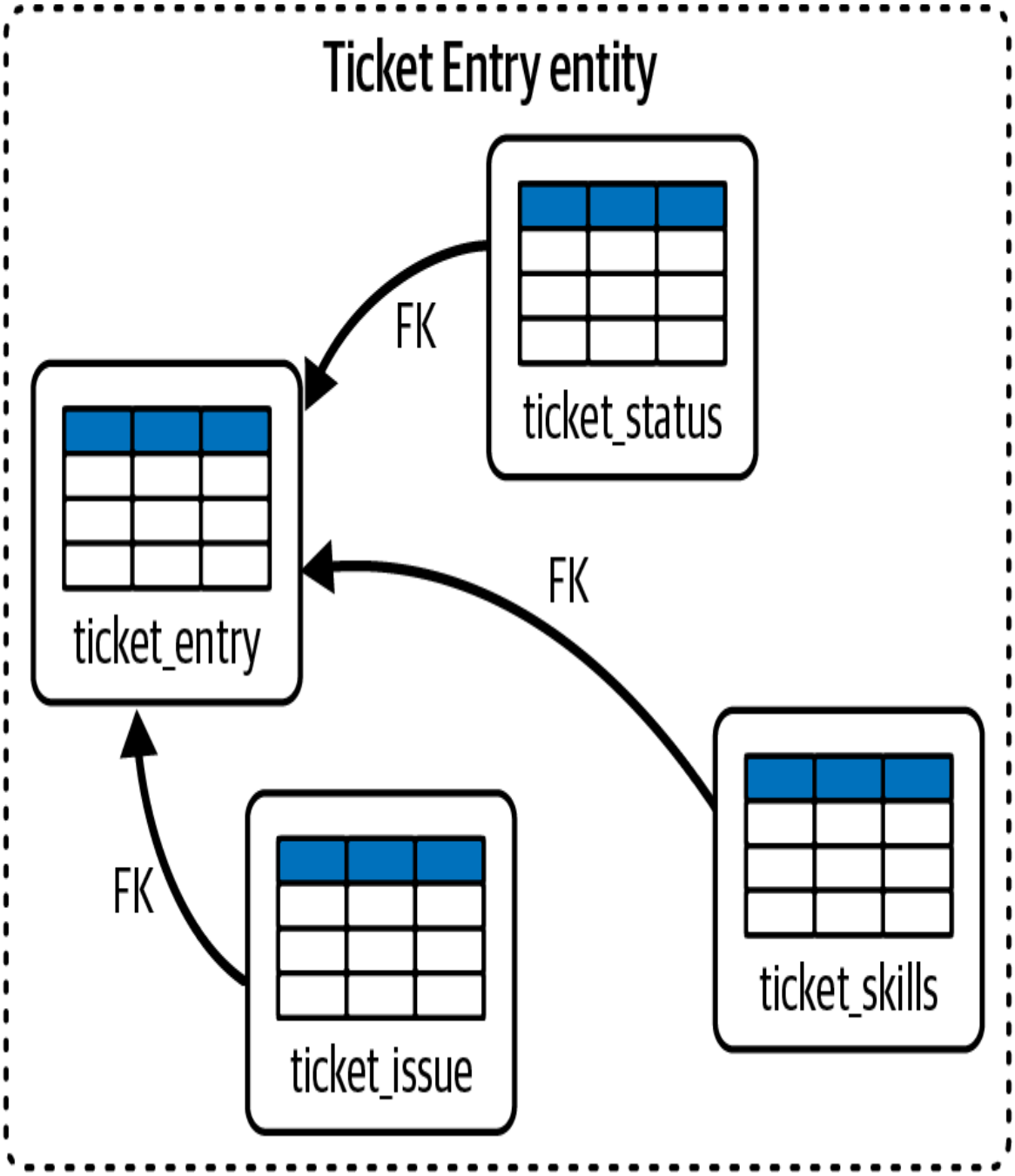


Figure 21-2. The `Ticket Entry` entity modeled as four separate tables

These four ticketing tables are considered *semantically coupled* because, collectively, they describe the `Ticket Entry` entity created by the

customer. They're also coupled from an *implementation* standpoint because the `ticket_issue`, `ticket_status`, and `ticket_skills` tables have a foreign-key relationship to the `ticket_entry` table. The `ticket_entry` table owns the primary key (`ticket_id`), which the other tables reference as a foreign key. Both forms of coupling are necessary to preserve the semantic meaning and data integrity of each ticket.

From a data perspective, making microservices too small is a pitfall because it causes data to lose its semantic and implementation coupling, and thus its meaning and integrity. To illustrate this point, consider the following microservices, which were created by the Sysops Squad application team and are used when a customer creates a ticket. Each microservice performs a single function:

`Ticket Entry` *service*

Captures the basic information about the ticket, such as customer and date information, and generates a unique ticket ID.

`Ticket Status` *service*

Tracks and updates the ticket's status as it changes during processing.

`Problem Description` *service*

Captures the customer's description of the problem and any additional supporting information (such as error codes).

`Skills Determination` *service*

Determines the expert skills needed to fix the problem. Other downstream services use these skills to locate an expert suited to fix the problem.

The first problem with making microservices too small is about forming bounded contexts and mapping data to each of the services. Each of the four database tables described earlier conveniently maps well to these four ticket

entry microservices. However, creating bounded contexts for these services destroys the `Ticket Entry` data's semantic meaning and its referential integrity, because each table is now in a *different* bounded context contained in a *separate* database or schema.

In Figure 21-3, the foreign-key relationships between the tables no longer exist. Semantically, these tables are no longer logically grouped together. Not only does this degrade data integrity, but remodeling the ticket entry data structure would significantly change which microservices are used when creating a ticket.

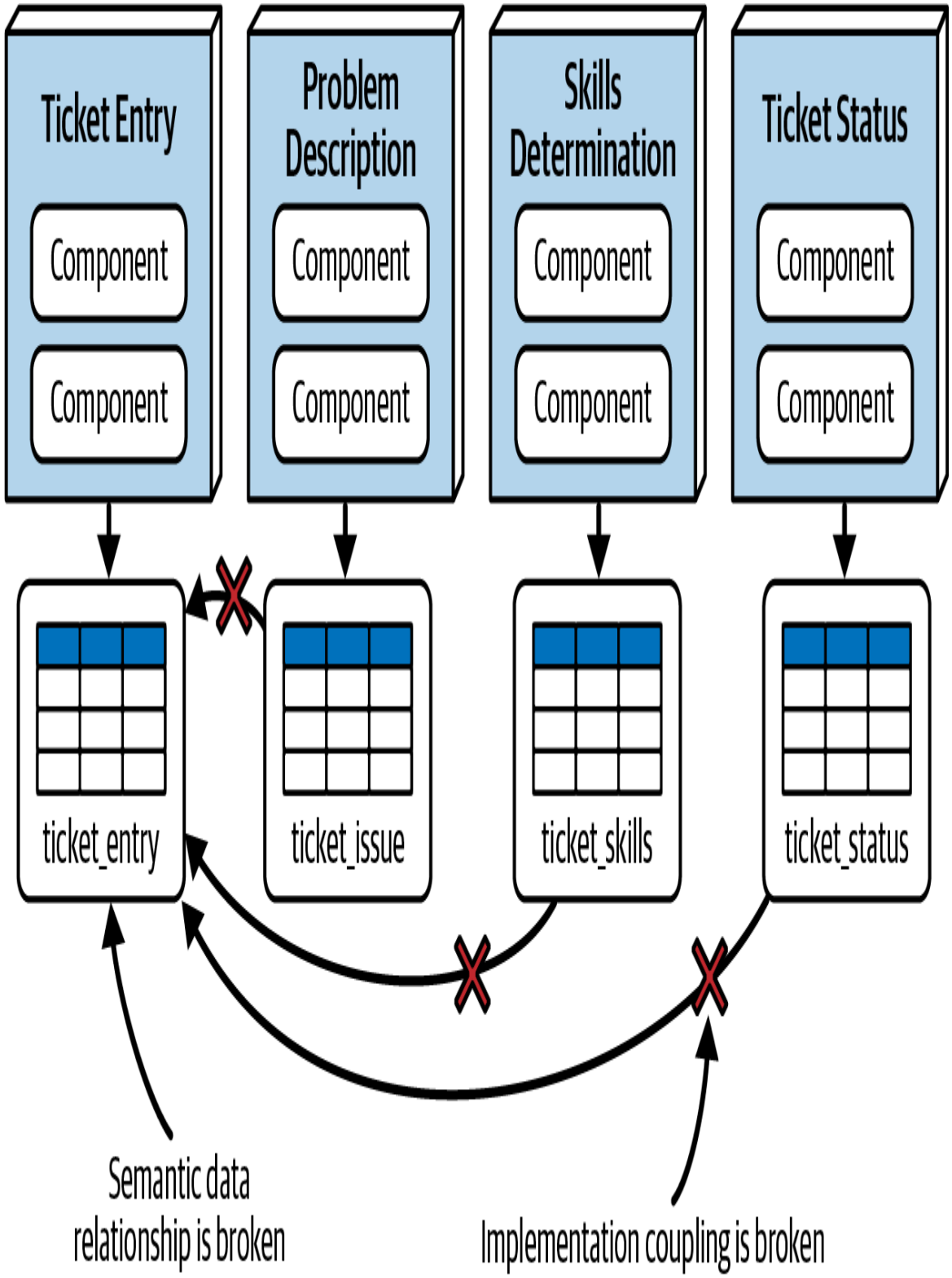


Figure 21-3. Because the services are too small, the Ticket Entry data model no longer has semantic meaning and referential integrity

Another reason why making microservices too small is a pitfall is that it can increase service communication and coordination, and thus the overall coupling between services. This isn't always the case when making microservices too small, but it's more likely with smaller, more fine-grained services.

Architects can use either Choreography (described in Chapter 9) or Orchestration (described in Chapter 27) to coordinate services. For example, all four ticket creation microservices shown in Figure 21-3 are required to complete the ticket creation process. Since the UI calls a single endpoint, architects can use an orchestrator to coordinate these services, as shown in Figure 21-4.

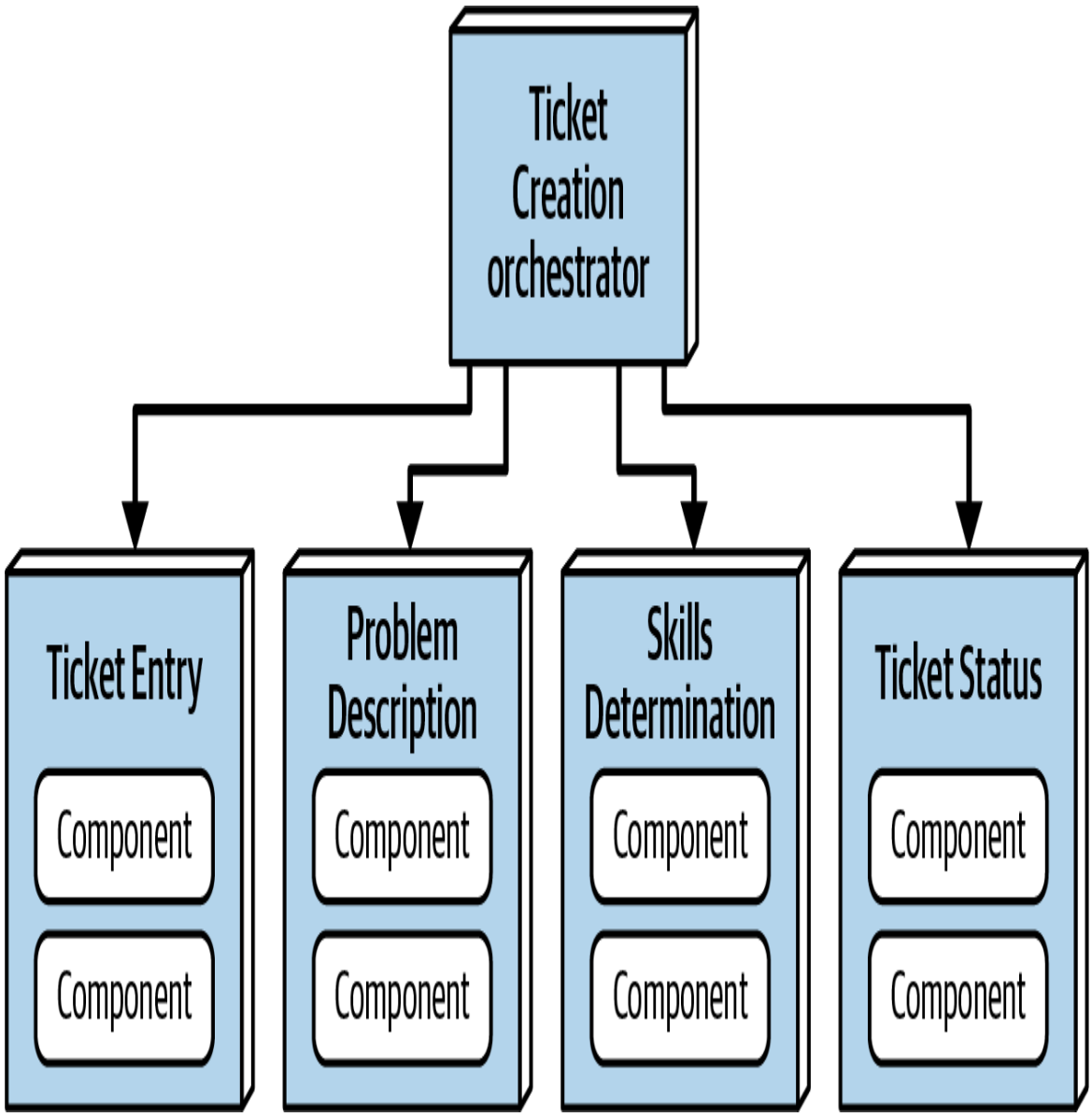


Figure 21-4. Creating a trouble ticket uses an orchestration service to coordinate the separate microservices

However, using orchestration in this manner can significantly degrade data consistency. Because the ticket creation process requires behavior from each of the four microservices, the single action of creating a ticket requires a *distributed transaction*, where each service independently commits its data updates within its own bounded context. The orchestration service manages each distributed transaction as an atomic transaction (see Chapter 37). If one service (say, `Ticket Status`) were to fail or throw an error while updating its data, all other previously committed updates

would have to be rolled back. This would require using *compensating updates*, or requests from the transaction coordinator (in this case, the orchestration service) to reverse any updates made during the course of this user request. That's when the problems start. See the fallacies of distributed computing for a new fallacy written by Mark and Neal that applies here: "Compensating updates always work."

Whether using Choreography or Orchestration, an architect must manage an atomic unit of work (that would be the customer's ticket request) through distributed transactions by leveraging compensating updates to deal with any errors. However, it's impossible to guarantee that those compensating updates will always work. While architects can sometimes use state management or eventual consistency (see Chapter 37) to help address these issues, the fact remains that making microservices too small compromises the overall consistency and integrity of the data. That's why this is a pitfall.

How to Avoid This Pitfall

Avoiding this pitfall is a matter of arriving at the most appropriate level of granularity for a microservice, resisting the temptation to make every microservice as fine-grained as possible.

Author and software architect Sam Newman offers some good advice in his book *Building Microservices*, 2nd edition (O'Reilly, 2021) about how to avoid this pitfall. Newman recommends starting out with coarse-grained microservices and moving to finer-grained ones as teams learn more about them. This is sound advice. It's generally easier to break a service apart than it is to put services together, and following this advice helps avoid the temptation to start out making everything too fine-grained.

In our book *Software Architecture: The Hard Parts*, we outline several drivers architects can use to learn more about a microservice and arrive at the right level of granularity. We categorize these drivers into *granularity disintegrators* — factors that drive architects to create smaller, more fine-grained services — and *granularity integrators* — factors that drive architects to create larger, coarser-grained services. Architects can use these

drivers to learn more about a service and decide whether to break it up or keep it coarse-grained.

Granularity Disintegrators

The following subsections describe granularity disintegrators, which drive breaking behaviors apart into smaller services.

Service scope and function

This driver relates to *cohesion*: how, and how much, the behavior or operations of a service are interrelated. In the customer notification service example we started the chapter with, notifying a customer via email, SMS texting, and postal letter could all be considered cohesive because they all relate to notifying a customer, and hence might suggest (per Sam Newman's advice) creating a single coarse-grained service containing all the notification logic. Now consider the app's customer-related behavior: customers can create and update a customer profile, post reviews on the website, and maintain their preferences. Although all these actions are all related to a customer, they aren't closely related to each other — in other words, they aren't highly cohesive. As such, it would likely be best to associate these actions with separate, finer-grained services rather than a single coarse-grained service.

Sometimes, after breaking up a service into smaller services, parts of the service's behavior are left over. For example, suppose the team has a justifiable reason to remove the email logic from a consolidated `Customer Notification` service and form a new, smaller `Email Notification` service. The team must now come up with a descriptive name for the resulting service that contains the leftover SMS texting and postal letter notification logic. Names like `Non-Email Notification` and `Other Notification` don't actually describe what the service *does*. As a matter of fact, we challenge the reader to come up with a good descriptive name for the resulting service containing the SMS texting and postal letter logic!

The reason this is so hard is that the leftover behavior is *no longer cohesive*. Service names like `SMS Postal Letter Notification` that simply combine the leftover behaviors are a good indication that behavior is no longer cohesive and should therefore be broken up into separate smaller services, as shown in Figure 21-5.

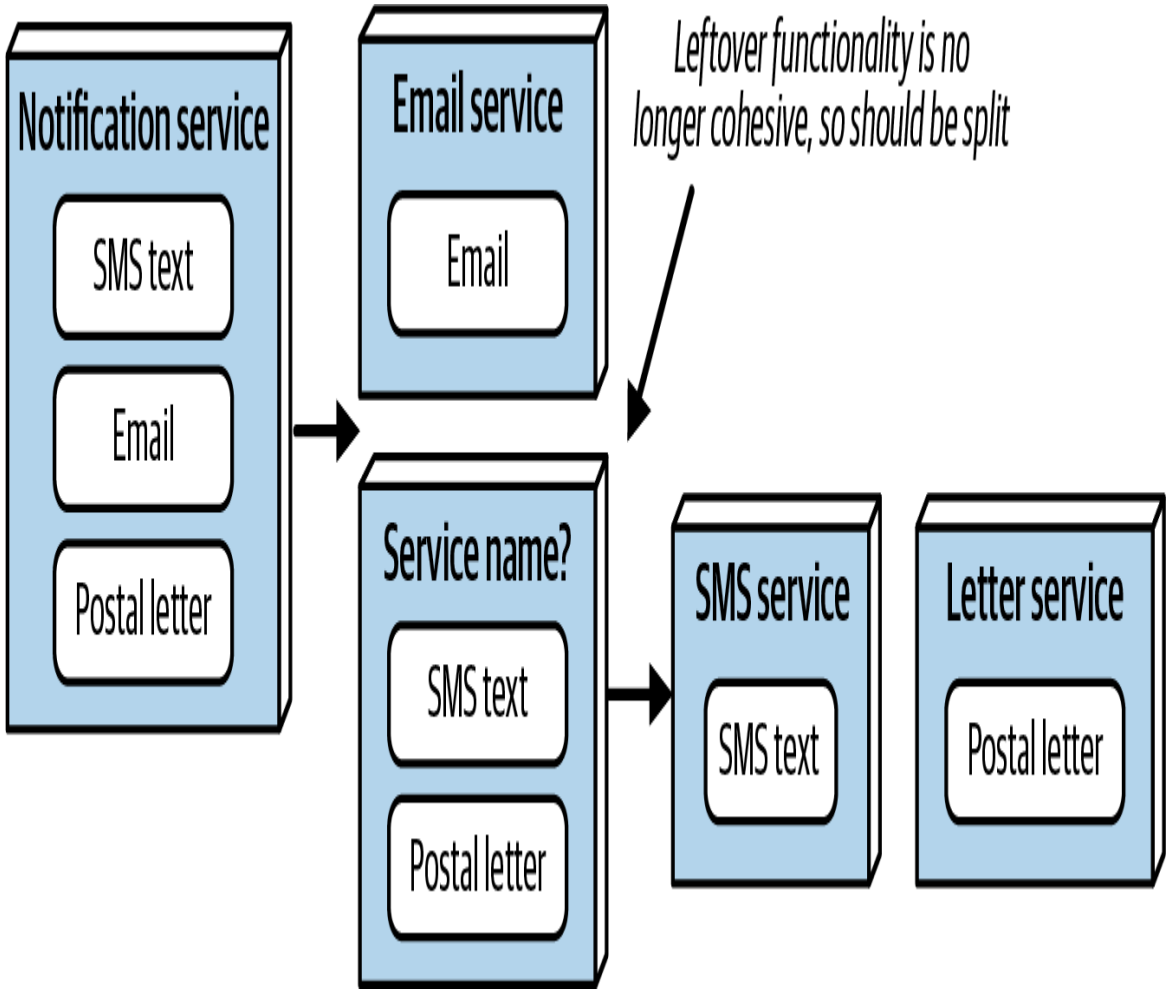


Figure 21-5. Leftover behavior that is no longer cohesive should be broken apart

Conversely, suppose the team moves the postal-letter logic into a separate, smaller service called `Letter Notification`, leaving the SMS texting and email logic as the leftover parts. Here, the team could create a consolidated `Electronic Notification` service, as shown in Figure 21-6, containing that behavior. Their shared nature as electronic communications means these leftovers are cohesive, so naming the resulting service is not overly difficult.

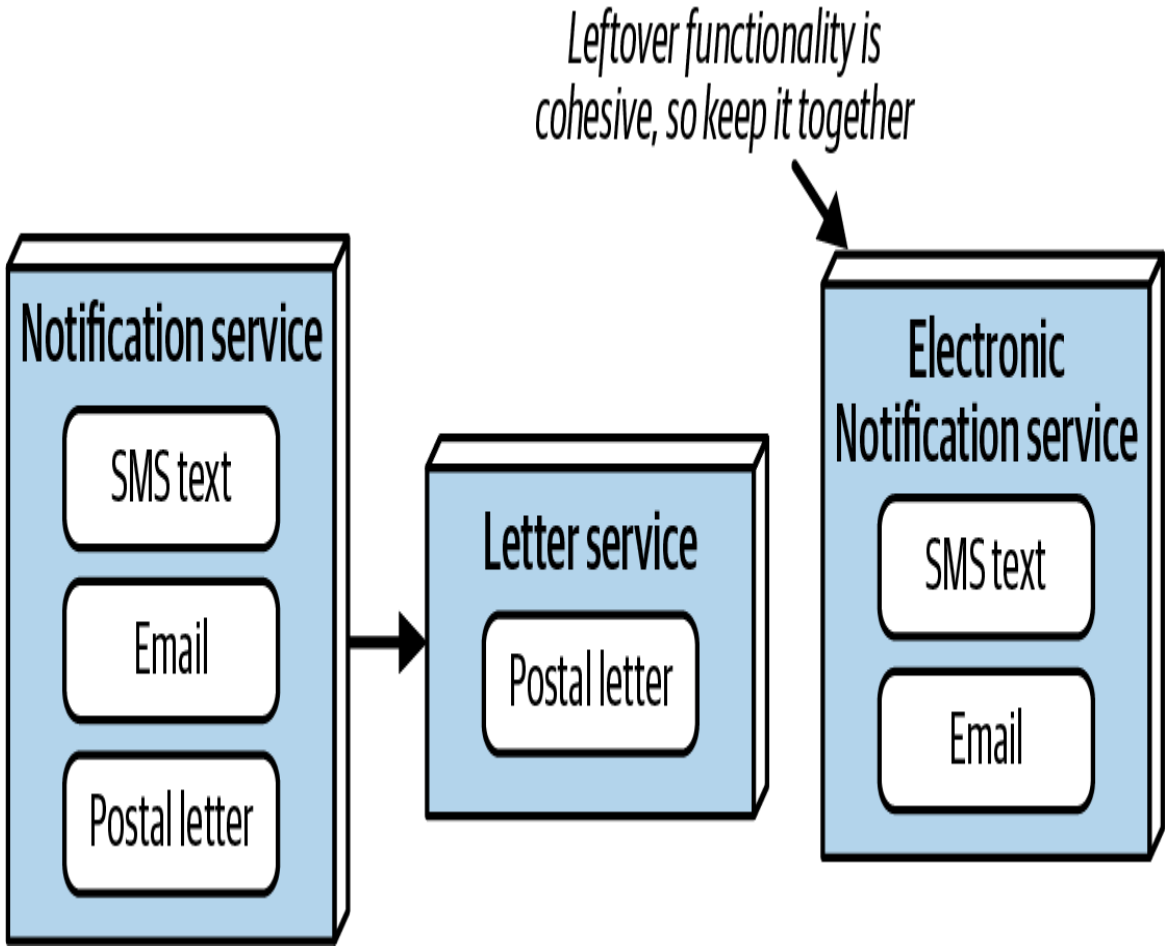


Figure 21-6. Leftover behavior that remains cohesive can reside in the same service

Code volatility

This driver is about how often the behavior within a service changes. Bigger, coarser-grained services are generally harder to maintain, have a larger testing scope, and carry a higher deployment risk than smaller, finer-grained services. If the code in one area or piece of behavior changes more often than other parts of the service — such as the postal-letter logic in the larger `Customer Notification` service — teams can extract that behavior and move it to a smaller, finer-grained service.

Teams can measure code volatility by using version control to track how often parts of a service change and which parts change more often. Gathering volatility metrics can help teams provide an objective justification for making services smaller.

Scalability and throughput

Teams can objectively measure and analyze the scalability and throughput needs of a particular service to determine if *all* parts of that service need to scale as much as others, and if needed, break the service apart to optimize scalability. For example, let's say the consolidated `Customer Notification` service sends 220,000 SMS texts per minute, 500 emails per minute, and, on average, 1 postal letter a minute. Because these notification methods have vastly different scalability needs, teams have a good justification for breaking these behaviors into smaller, finer-grained services to optimize each method's scalability needs.

Fault tolerance

Fault tolerance is how well other functions in the system continue to operate when a fault occurs. Teams can learn more about a service's fault tolerance by measuring, tracking, and analyzing its fatal errors. If one part of a service continuously fails, it might make sense for the team to extract the error-prone logic into separate services, isolating the faulty function. If the team for our consolidated `Customer Notification` service finds through gathering metrics that the postal-letter logic continues to experience fatal errors due to memory leaks within the printer connection pools, they can extract that logic into a separate `Letter Notification` service, renaming the remaining SMS and email logic to be an `Electronic Notification` service.

Another consideration related to fault tolerance is *Mean Time To Start* (MTTS) — the time it takes to start up a new instance of a service. Suppose the SMS texting part of our consolidated `Customer Notification` service takes 200 ms to start up (that's pretty fast). However, the email part of the service takes 1,800 ms to start up, because it has to establish an SMTP connection pool. The postal-letter part of the service takes a whopping 8,000 ms to start up, because it has to establish connections to each printer to check its health. When a fatal error occurs in the email part of the consolidated `Customer Notification` service, it takes 10,000 ms to start up a new instance to reestablish SMS texting logic (that would

be its MTTS). Considering that this consolidated service sends 220,000 SMS texts per minute, that's way too much time. Breaking up this service into three separate smaller services reduces the MTTS for SMS logic to 200 ms, which also satisfies the scalability and throughput requirements.

Security

In some cases, parts of a larger service may have different data security needs than other parts. Consider a coarse-grained `Customer Profile` service that manages customers' profiles and credit card information. Access to the credit card information clearly needs to be more secure than the customer's demographic information. However, because the service itself controls access to this information, someone with ill intent could steal sensitive credit card information by entering into that service through any of its public API endpoints.

Security reasons are a good justification for creating smaller, more fine-grained microservices. Breaking the larger `Customer Profile` service into smaller `Customer Demographics` and `Wallet` services allows teams to provide access-control checks at the API gateway *prior* to anyone entering the service, providing much better security for sensitive data.

Evolvability

Evolvability is the ease with which teams can add new features or functions to the architecture. Teams must sometimes extend the behavior of a system, such as adding a new payment type or shipping method. In these situations, they're often better off using fine-grained services to represent each type. Consider a coarse-grained `Payment` service. Customers can pay for their orders using a credit card, Apple Pay, or Google Pay. As the company continues to offer additional payment types, such as reward points or gift cards, the team faces the same maintainability, testability, and deployability issues as we described with code volatility. Complicated code bases, large testing scopes, and high deployment risk make adding payment types more difficult than it should be.

Making each payment type its own fine-grained service means that adding a new payment type is only a matter of creating a new microservice and adding it to the ecosystem — with no need to change, test, or redeploy any existing payment logic. When these situations occur, starting out finer-grained might actually be the right choice.

The astute reader might be confused at this point: we just listed six reasons teams might *want* to make their microservices smaller. Isn't that exactly what this pitfall is about? Yes and no. The Grains of Sand pitfall occurs when architects don't start with coarser-grained microservices or take *granularity integrators* into account.

Granularity integrators

Analyzing granularity trade-offs is all about balancing granularity disintegrators and integrators. *Granularity integrators* — the factors that drive architects to put multiple behaviors together into larger services — are described in the following sections.

Database transactions

To process a user request, a single service uses a transaction managed by the database (called a *database transaction* or *ACID transaction*). Most teams are familiar with the drill: start the transaction using a `begin` statement, perform multiple database updates, then issue a `commit` statement to complete the database processing. If an error occurs while processing the request, issue a `rollback` statement to reverse all updates performed during processing. Here, the database handles all of the transactional complexity of coordinating multiple updates.

However, when a single user request requires multiple services, those services are involved in a distributed transaction — one that needs to be coordinated outside of the service context. A microservice involved in the multiservice user request commits its own work, then request processing moves onto the next microservice to perform its updates, and so on. If an error occurs anywhere along this chain, a coordinator — usually an

orchestration service — must now tell the services to reverse the work they've done through compensating updates (see Chapter 11 for more information about this pattern).

If it's important that database updates made by multiple services be *atomic* — always guaranteed to be consistent — then consolidating their logic into a single, more coarse-grained service might be the more appropriate choice.

Data relationships

As we discussed at the beginning of this chapter, when breaking apart behavior into small, fine-grained microservices, it's also necessary to break the data apart to create well-formed bounded contexts. However, as illustrated in Figure 21-3, this breaks the data's semantic and implementation coupling. In many cases, teams that have good justifications for breaking *behaviors* apart into smaller, finer-grained services, fail to account for the *data* used by those behaviors. To get granularity right, teams must consider both of these factors.

Shared code

Shared code is a natural — and necessary — part of software development. String manipulation, date functions, logging, and metrics gathering are all examples of shared behaviors. If small, fine-grained services all share lots of common code, the team should consider the *change rate* — the code volatility — of that shared code when determining service granularity. If frequent changes to a large amount of shared code require changes in all the services using that code, it might make more sense to consolidate that behavior into a single service to avoid dealing with shared libraries, versioning, service change coordination, and deployment churn.

Workflow and choreography

Smaller, finer-grained services sometimes require workflows to tie them together, but this practice degrades responsiveness, scalability, fault tolerance, and data consistency. Teams that need to connect small microservices to perform a single business request should balance those

needs against any granularity disintegration drivers and consider consolidating the smaller services into a single, coarser-grained service.

The secret to getting service granularity right is balancing the trade-offs between disintegration and integration drivers. We recommend taking Newman's advice: start out coarse-grained with microservices and only break them apart when it's justified. Remember to account for the granularity integration drivers.

Trade-Offs

We've labeled Grains of Sand a pitfall because any advantages of making all microservices too small quickly fall away as services become interconnected, leaving only disadvantages. The following sections outline the reasons teams might be tempted to adopt this pitfall and what usually results when they do.

Pros

While a few systems could perhaps take advantage of every microservice being a fine-grained, single-purpose function, most systems that use this practice suffer (that's why it's a pitfall). However, teams start out creating systems this way for many reasons.

First, microservice architectures provide the opportunity for *function-level* scalability, meaning that only the functions that *need* to scale do so. Some teams make *all* of their microservices small to take advantage of this capability. However, when those small microservices need to interconnect with others to perform a business function or don't need the level of scalability offered by fine-grained behavior, this advantage goes away.

Second, microservice architectures are highly fault-tolerant. By creating fine-grained microservices, teams can leverage this superpower to have the highest level of fault tolerance possible at the function level. However, this superpower quickly disappears when services that are *too* fine-grained must

communicate with other fine-grained services to complete a single business request.

Cons

Most teams that adopt the Grains of Sand pitfall experience few or no benefits — just the pain of managing lots of fine-grained services unnecessarily. Interconnectivity is the key here. Making microservices smaller increases the likelihood that they'll need to communicate with one another to complete a single request. Again, while we acknowledge that this rule has a few exceptions, we urge architects to start out coarse-grained and move to finer-grained as necessary — and in some cases it might be necessary.

Having too many fine-grained microservices means a large number of "moving parts" in the system, which makes it more difficult to debug and analyze root causes of issues at a system level. Also, spreading cohesive behavior across multiple services makes it harder to change existing logic and data, and manage distributed transactions, creating data consistency issues throughout the system.

From a cost perspective, fine-grained microservices multiply the business's maintenance and licensing costs. Small microservices also take longer to design, lengthening the overall initial time to market for changes and new features.

Comparison Chart

Table 21-2 summarizes the capabilities trade-offs associated with this pitfall. In the comparison chart, the Adopting Pitfall column refers to teams naively making all their microservices too fine-grained from the start (as described in this chapter), whereas Avoiding Pitfall refers to teams starting out more coarse-grained with their services, and moving to finer-grained ones as they learn more about the services.

Table 21-2. Comparison chart for the Grains of Sand pitfall

Capability	Adopting Pitfall	Avoiding Pitfall
IMPROVED		
None	---	---
DEGRADED		
Adaptability	★	★★★★★
Cost	$$$$$	$
Data Consistency	★	★★★★★
Data Integrity	★	★★★★★
Initial Time to Market	★	★★★★★
Simplicity	★	★★★★★

The following sections describe each of the capabilities found in Table 21-2.

Improved

This particular pitfall improves no capabilities — it only degrades them.

Degraded

This list contains the degraded capabilities when adopting this pitfall:

Adaptability

Making microservices too small tends to spread cohesive behaviors across multiple services, making it more difficult to change existing logic and data.

Cost

Significant cost and effort overruns are common when starting out with too many fine-grained services. Licensing fees and the effort of data modeling, design, and maintenance all increase costs when adopting this pitfall.

Data Consistency and Data Integrity

Forming bounded contexts when making microservices too small means also breaking the data apart to a fine-grained level to associate it with the small service. Not only does this degrade the semantic definition of the data, it also requires complex distributed transactions, degrading both data integrity and data consistency.

Initial Time to Market

Lots of very small microservices take longer to design, particularly from a data and communication standpoint, impeding the team's ability to get changes and new features to market quickly.

Simplicity

Creating fine-grained microservices requires modeling fine-grained data — a big challenge in most business systems. Stitching small microservices together through Choreography or Orchestration also adds complexity to the system, particularly when distributed transactions are involved.

Influences

No external factors influence the trade-offs discussed in the prior section.

Related Patterns, Antipatterns, and Pitfalls

The following patterns and pitfall are related to the Grains of Sand pitfall. However, they are *not* fixes for this pitfall:

Choreography pattern (see Chapter 9)

> Services that are too small often need to be stitched together to create a workflow to handle business requests.

Compensating Updates pattern (see Chapter 11)

> When errors occur in a distributed transaction, most teams use compensating updates to reverse work already committed by prior microservices within the workflow.

Microservice All The Things pitfall (see Chapter 24)

> Like Grains of Sand, this pitfall has to do with service granularity.

Orchestration pattern (see Chapter 27)

> Orchestration is an alternative to Choreography for coordinating microservices.

Service Sharding pattern (see Chapter 30)

> The related Service Sharding pattern is about adjusting the granularity of services to better manage distributed transactions.

Chapter 22. Implicit Workflow State Pattern

When working with event-driven architectures (EDAs), tracking the state of any workflow can be challenging: Which services have executed their step in the workflow? What was the result? Did the system encounter any errors? Patterns to help track workflow state include Cross-Cutting State (Chapter 13), Front Controller (Chapter 19), and Explicit Workflow State (Chapter 17). Implicit Workflow State is another workflow state management pattern that architects can add to their toolkits.

Table 22-1 summarizes the metadata for the Implicit Workflow State pattern.

Table 22-1. Metadata for the Implicit Workflow State pattern

CATEGORY	METADATA
CONTEXT:	Event-Driven Architecture, Message-Based Architectures, Microservices, Workflow
IMPROVED:	Testability, Workflow
DEGRADED:	Adaptability, Domain Alignment, Evolvability, Simplicity

How Should I Manage My Workflow State?

EDAs tend to be organic. An initiating event kicks off a workflow, to which one or more processors or services may subscribe. They process it, then

publish their own derived events for consumption by other processors. One of the biggest challenges architects face when working with EDA is building out mechanisms to show the current state of the workflow.

Consider the scenario illustrated in Figure 22-1, which is very much like the one we discussed in Chapter 17.

A customer places an order for a book, which triggers a `purchase book` event, which kicks off an event-driven workflow. As different processors go about reacting to that event, they trigger other derived events. For example, the `Order Placement` service emits the `order placed` event, to which three downstream processors subscribe: `Payment`, `Notification` and `Inventory`.

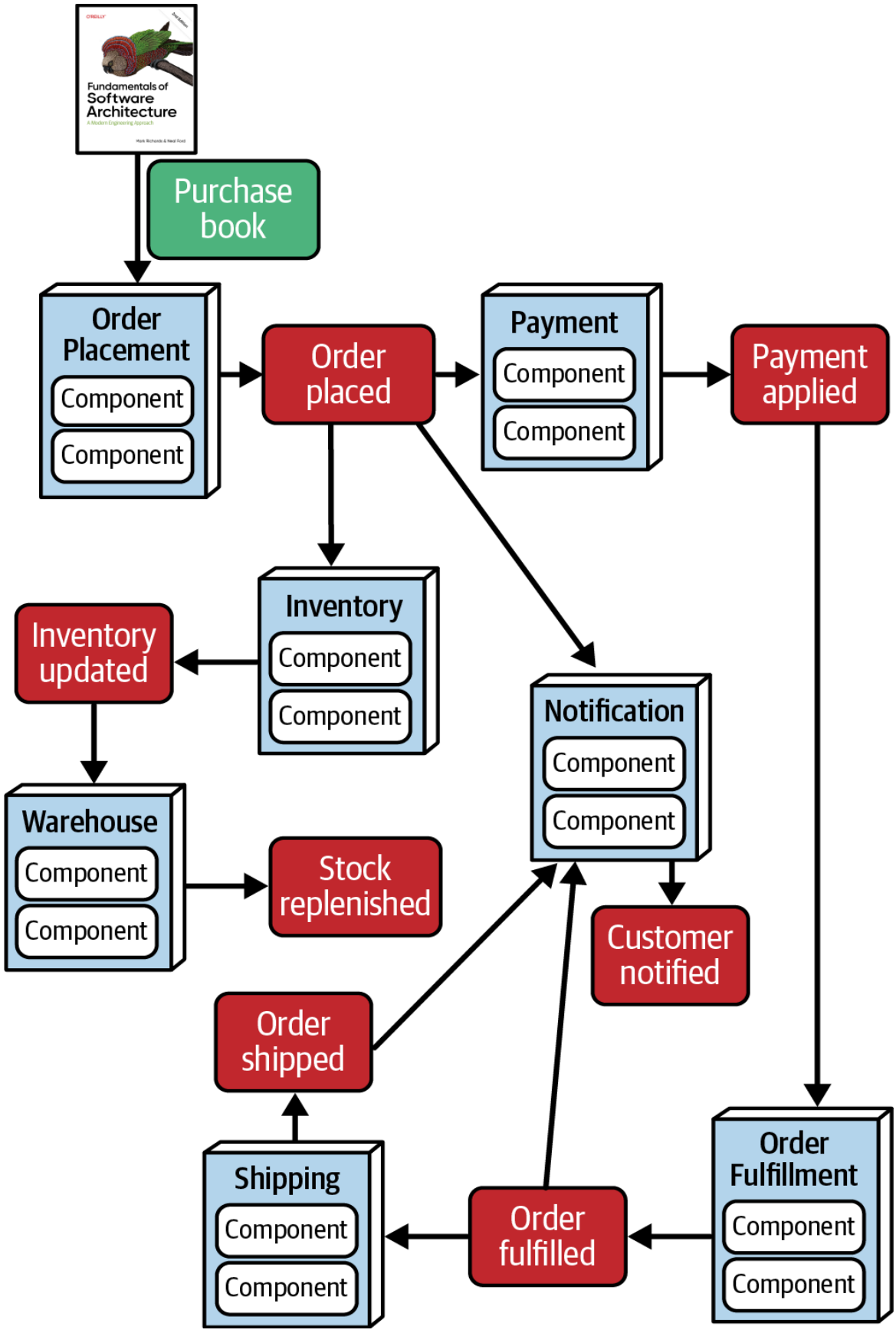

Fundamentals of Software Architecture
Purchase book
Order Placement
Component
Component
Order placed
Payment
Component
Component
Payment applied
Inventory
Component
Component
Inventory updated
Notification
Component
Component
Warehouse
Component
Component
Stock replenished
Customer notified
Order shipped
Order Fulfillment
Component
Component
Shipping
Component
Component
Order fulfilled

Figure 22-1. A typical processing flow for an event-driven system

Let's say the customer wishes to change or cancel their order. Now what is the state of their order? Has the payment gone through? Has the order been shipped? In the architecture depicted in Figure 22-1, it's hard to discern an answer, because no singular source captures state! This is where the Implicit Workflow State pattern can help.

Applying the Pattern

To apply the Implicit Workflow State pattern to an EDA topology, architects must ask what workflow state needs to be managed and which derived events they can leverage to infer workflow state. Consider Figure 22-1 once again. Where in this system can a particular customer's order be found? Looking carefully at the workflow, an architect can identify all the events that pertain to customer orders, as shown in Figure 22-2.

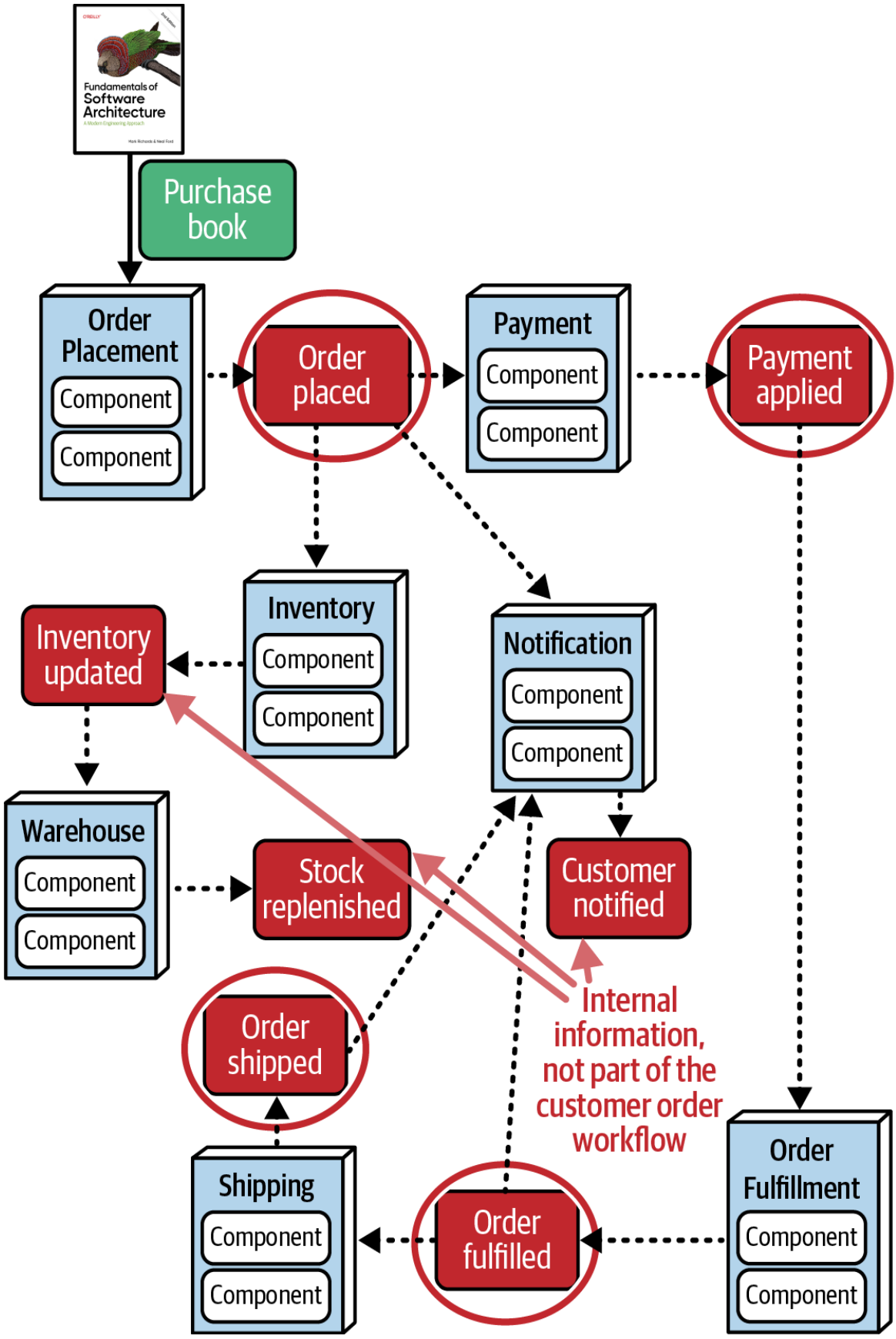

Fundamentals of Software Architecture
Purchase book
Order Placement
Component
Component
Order placed
Payment
Component
Component
Payment applied
Inventory
Component
Component
Inventory updated
Notification
Component
Component
Warehouse
Component
Component
Stock replenished
Customer notified
Order shipped
Internal information, not part of the customer order workflow
Shipping
Component
Component
Order fulfilled
Order Fulfillment
Component
Component

Figure 22-2. Events pertaining to customer orders

With the identified events in hand, the architect chooses a workflow state owner. One option here is to simply choose an existing processor and overload it to act as the state owner. This workflow state owner then proceeds to subscribe to all derived events that affect the workflow in question. Figure 22-3 illustrates this setup.

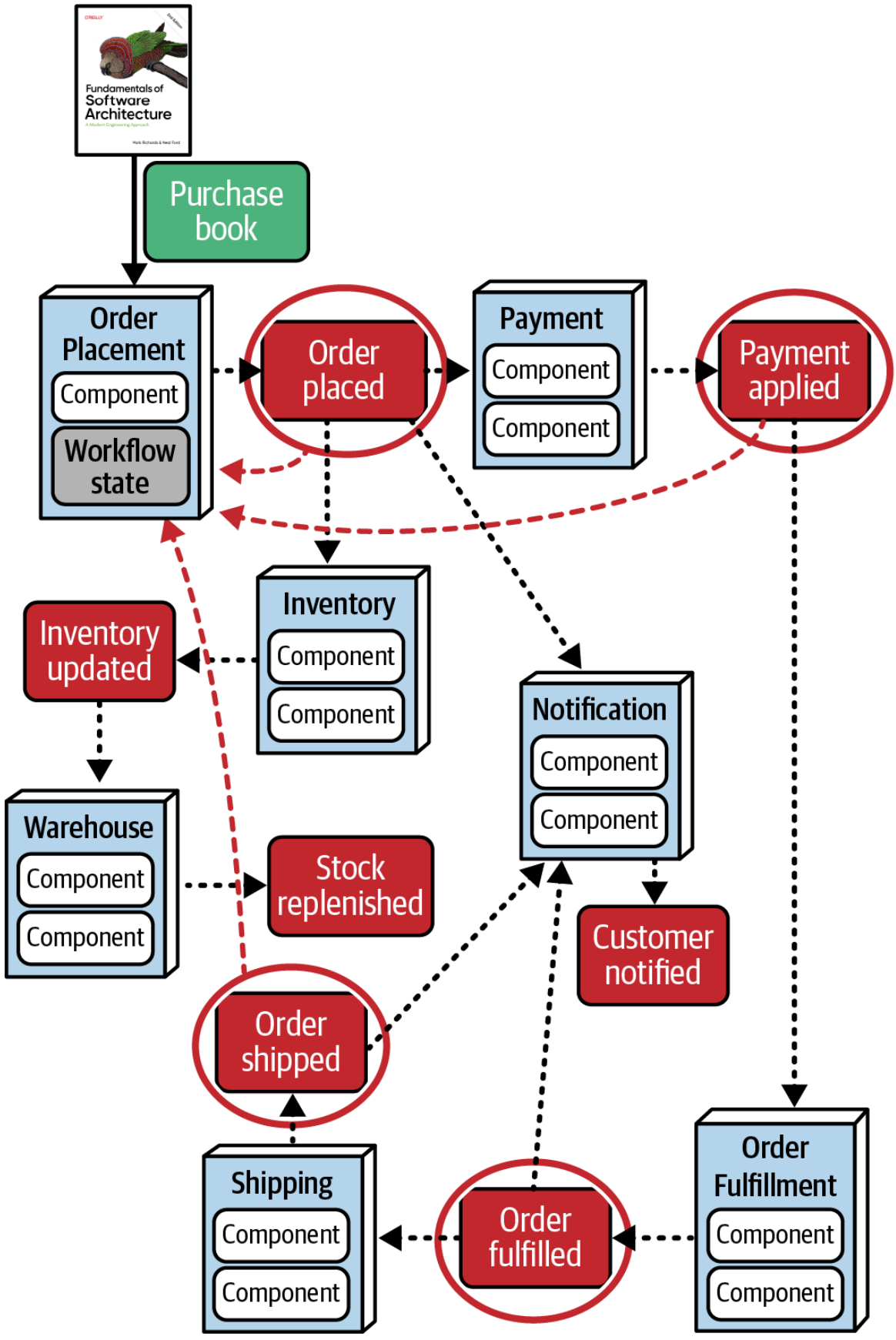

Fundamentals of Software Architecture
Purchase book
Order Placement
Component
Workflow state
Order placed
Payment
Component
Component
Payment applied
Inventory
Component
Component
Inventory updated
Notification
Component
Component
Warehouse
Component
Component
Stock replenished
Customer notified
Order shipped
Shipping
Component
Component
Order fulfilled
Order Fulfillment
Component
Component

Figure 22-3. Implicit Workflow State pattern

The state owner is essentially a consolidated state machine. Upon receiving, say, an `order placed` event, it updates the state of that particular order to `PLACED`. Then, when it sees the `payment applied` event, it updates the state of the workflow to `PAID`, and so on. This is depicted in Figure 22-4.

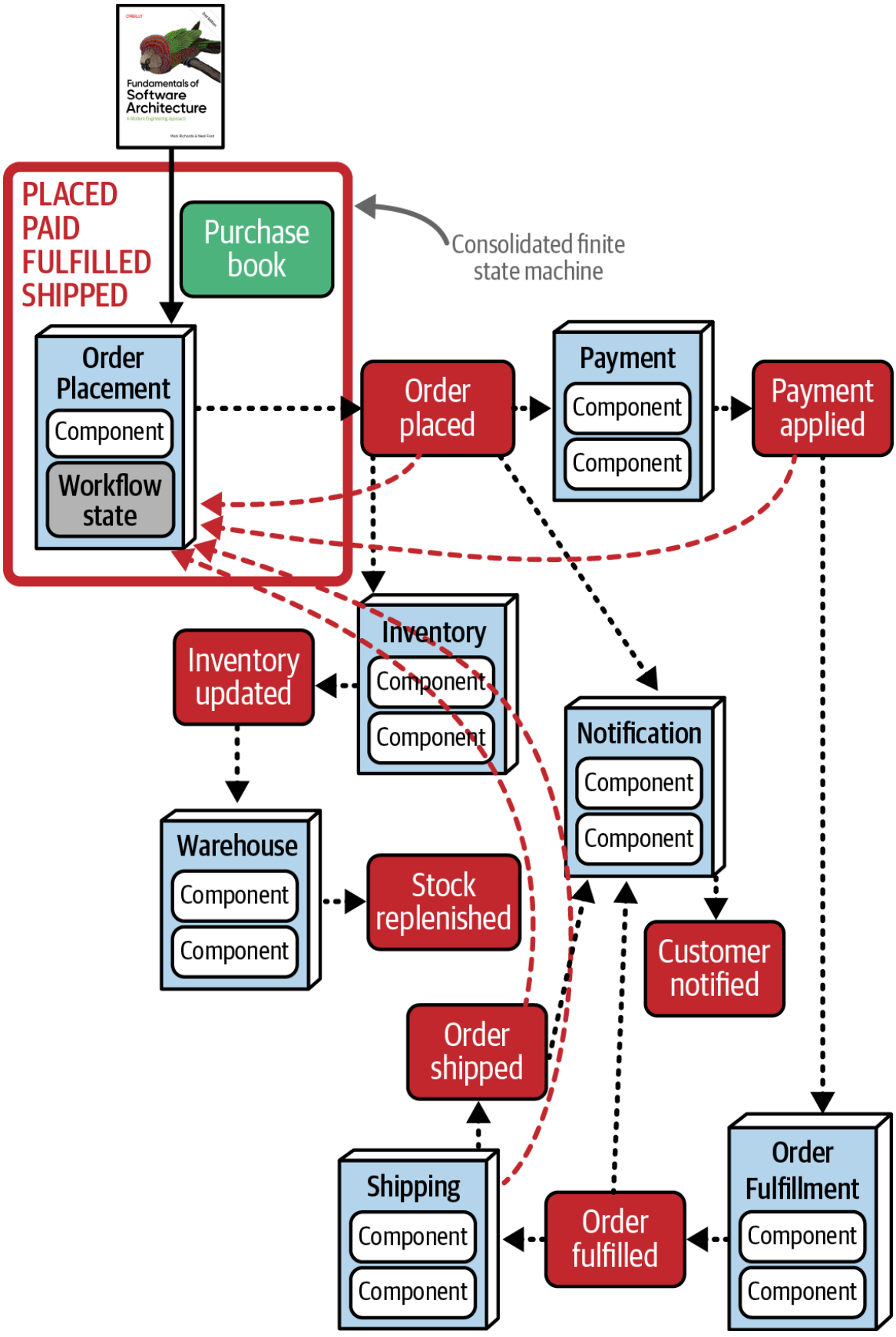

Fundamentals of Software Architecture
PLACED
PAID
FULFILLED
SHIPPED
Purchase book
Consolidated finite state machine
Order Placement
Component
Workflow state
Order placed
Payment
Component
Component
Payment applied
Inventory
Component
Component
Inventory updated
Notification
Component
Component
Warehouse
Component
Component
Stock replenished
Customer notified
Order shipped
Shipping
Component
Component
Order fulfilled
Order Fulfillment
Component
Component

Figure 22-4. Consolidated state machine

Going forward, the workflow state tracker acts as a single source of truth for any question — like "Where is this order?"

Of course, architects can always choose to create a dedicated processor to act as the workflow state owner rather than overloading an existing one. This approach works similarly, but shifts some of the trade-offs around.

Trade-Offs

This section discusses the trade-offs of the Implicit Workflow State pattern.

Pros

Benefits of the Implicit Workflow State pattern include:

Workflow

> Tracking state is such a common problem in event-driven architecture that architects created a named pattern — the Implicit Workflow State pattern. The Implicit Workflow State pattern solves a real problem. Knowing the state of any workflow in an architectural style that doesn't inherently have a centralized datastore (like EDA) is tricky.

No service involvement

> This pattern does not mandate modifying any of the processors involved in the workflow state, since the workflow state owner harvests the state from the messages these services produce.

Cons

The negatives of the Implicit Workflow State pattern include:

Coupling

> What's not immediately obvious in this pattern is that the state owner *implicitly* infers state. Recall that the state owner is subscribing to the

same events that a processor emits as part of its work. From these sets of events, the state owner attempts to figure out the state of the workflow.

Consider what would happen if a processor emitted multiple events as part of processing an incoming event. Which should the state tracker subscribe to — all of them? Some of them? How would it know? If an architect changes the internals of a processor or the number or granularity of the events without also updating the state owner, that state owner is no longer a reliable tracker.

The state tracker could always receive events (from one or more processors) out of order. It must explicitly understand how to lay out the events correctly so it can deduce the state of the workflow.

All of this suggests an implicit coupling between the workings of individual processors and the state tracker that aims to make sense of all the events flowing through the system.

As suggested earlier, this pattern offers two implementations. An architect can choose to overload an existing processor with state tracking responsibilities, or create a single-purpose processor just to track workflow state. Both options increase coupling throughout the architecture. In addition, overloading an existing processor with additional responsibilities decreases cohesion, because the processor now has two responsibilities. However, introducing a new service increases the system's operational complexity.

Comparison Chart

The trade-off analysis chart for the capabilities affected by this pattern appears in Table 22-2. It evaluates both *workflow inspection* capabilities (how easy it is to track workflow state) and *service implementation* capabilities (the cost and complexity each approach imposes on individual services). The Without Pattern column refers to the case where workflow state is not universally managed or tracked and services maintain their own state, which makes discerning workflow state extremely hard. This chart

also contrasts this pattern with the Explicit Workflow State pattern, an alternative pattern that's described in Chapter 17, where every service involved in the workflow explicitly emits a specific derived event with the sole purpose of updating the state. Much like in the Implicit Workflow State pattern, here, state is tracked by a workflow state owner.

Table 22-2. Comparison chart for the Implicit Workflow State pattern

Capability	Without Pattern	Explicit Workflow State Pattern	Implicit Workflow State Pattern
IMPROVED			
Testability	★	★★★★★	★★★
Workflow	★	★★★★	★★★★
DEGRADED			
Adaptability	★★★★★	★★★	★★★★
Domain Alignment	★★★★★	★★★	★★★★
Evolvability	★★★★★	★★★	★★★★
Simplicity	★★★★	★★	★★★

All of the capabilities that the Implicit Workflow State pattern improves are related to tracking and testing workflow state, while all service implementation capabilities suffer in this pattern.

Improved

The following capabilities are improved when using the Implicit Workflow State pattern:

Testability

> Testing is improved as compared to not tracking workflow state at all. Although the Implicit Workflow State pattern provides architects with well-known testing targets and outcomes, testing services that don't explicitly track workflow state requires setting up and coordinating multiple services and querying each one independently, making testing complex. Furthermore, since the state is inferred in this pattern, teams must use coordinating events from multiple services to make sure the test accurately reflects the actual workflow state. This adds complexity to test setup and validation. However, testing isn't as good as when using the Explicit Workflow State pattern, in which individual services explicitly update state as opposed to deriving it. This makes test verification straightforward and deterministic.

Workflow

> Workflow management is the primary reason to utilize this pattern. Workflows often have shared state across services, and this pattern provides a well-defined way to manage it. The Implicit Workflow State pattern improves this capability slightly, since the state has to be *inferred* from domain events. Without a pattern, querying for workflow state involves querying each service in the workflow and stitching the state together. Explicit Workflow State does well here too, but with a different set of trade-offs. It introduces multiple points of failure, the potential for the dual write problem, and increased coupling across the architecture, all increasing the possibility of drift between actual and recorded workflows.

Degraded

These capabilities are typically degraded when teams choose this pattern as compared to not tracking state. However, they are less degraded than with the Explicit Workflow State pattern, since the latter requires every service to be modified to emit an explicit "state update pattern," meaning every service has to be modified. In the Implicit Workflow State pattern, all these capabilities degrade because workflow state is not only being managed elsewhere but *passively* recorded, without the services' or brokers' knowledge.

Adaptability

Using this pattern allows architects to keep services closely aligned to their domain purpose. However, they might need to hold some incidental state related to cross-service interactions, which may degrade adaptability slightly.

Domain Alignment

Consider Figure 22-3 again. An architect might look at this diagram and see multiple domains at play. The "customer-facing" domain involves the `Order Placement` and `Payment` services, while `Order Fulfillment` and `Shipping` are about getting the orders to the customers. In fact, certain patterns, like the Domain-Broker pattern, leverage this to break up their architectural quanta into multiple smaller quanta.

Although the workflow state owner now subscribes to all these events, it's still just another subscriber across domain boundaries. The state owner's passive observation helps preserve domain abstraction, because the brokers are unaware of the state tracker's existence. However, it does potentially couple multiple domains to the workflow state owner, slightly diluting the domain alignment.

Evolvability

Domain services are unaware of workflow state management, so they remain focused on their domain purpose. This domain alignment makes

them more evolvable. However, if architects or teams need to make radical changes to the architecture, the inferred state can become a sticking point because the workflow owner must always be top of mind. Adding any new services, derived events, or brokers to the architecture means the workflow owner will also need to be modified.

Simplicity

Without workflow state tracking, each service is independent, but workflow inspection is a lot more difficult. This pattern infers state, which moves the logic of deducing what the state is to another party: the state owner. This inference logic can be complex. The state owner must handle event ordering, accommodate out-of-order events, and understand the semantic meanings of derived events from every service in the workflow. If an architect chooses to overload an existing service and make it the state owner, that service now handles two orthogonal concerns: its own business logic and workflow state inference. Finally, adding state means adding moving parts, which increases operational overhead and degrades simplicity.

Influences

As defined in "Architecture Quantum", architecture quantum has five facets, all of which must be considered in relation to the Implicit Workflow State pattern:

Static coupling

Static coupling refers to the dependencies required for a service or system to function independently. This pattern doesn't influence static coupling, since the workflow state owner is just another subscriber to all the topics issued by other processors. In an EDA topology, this introduces nothing additional.

However, in certain scenarios, architects may choose to have a broker per domain (see the Domain-Broker pattern in Chapter 15). This pattern

involves a static dependency between the workflow state owner and the broker itself, so if the architect decides to introduce additional brokers, perhaps in an attempt to narrow the architecture quantum, they must be careful to update the workflow state owner so that it can still reliably track state.

Dynamic coupling

Nothing about the Implicit Workflow State pattern introduces anything that a traditional EDA wouldn't already have in its topology.

Contract coupling

In this pattern, the workflow state owner is a *passive* subscriber to existing events and attempts to infer state from the events it receives. However, architects *may* have to add extra information within the events to assist with state management, thus increasing coupling between the individual processors.

Temporal coupling

Temporal coupling doesn't really apply to the Implicit Workflow State pattern, which has no specific time-based dependencies.

Atomic coupling

Atomic coupling and atomicity concerns are not influenced by this pattern.

How to Govern This Pattern

Architects choose this pattern when they need to capture some cross-cutting state, which means that the state itself is a good candidate for governance. Several things can affect how accurately the state of the workflow is depicted, and governing them serves as a good proxy for governing the overall state of the workflow.

For instance, all events that contribute to state transition must have well-defined and well-governed contracts. Teams might need to check those events for required fields and ordering semantics. A stricter change-management process may be needed when modifying those event schemas, to avoid breaking any assumptions that the state owner might be making.

Another vector is that this pattern requires a higher degree of monitoring and observability to detect out-of-order events, which can greatly influence the ability to track state correctly.

Related Patterns, Antipatterns, and Pitfalls

The patterns related to the Implicit Workflow State pattern include:

Explicit Workflow State pattern (see Chapter 17)

> The Explicit Workflow State pattern offers a more generic solution that can work in event-driven or message-based architectures. It's a complement to the Implicit Workflow State pattern, with different trade-offs. For architects seeking a solution to state management in distributed architectures, it's a worthy contender.

Cross-Cutting State pattern (see Chapter 13)

> Cross-Cutting State is a generic pattern that helps manage and track state in many different kinds of distributed architectures.

Front Controller pattern (see Chapter 19)

> This variant of the Cross-Cutting State pattern places workflow state among the first-called choreographed services.

Chapter 23. Key-Based Event Contract Pattern

When triggering events or sending messages, architects are faced with a decision about the event payload. They can choose to create a payload contract containing all the data necessary for reacting and responding to the event (that's the Data-Based Event Payload contract in Chapter 14), or they can choose to send *only the keys* of that data. When responding to the event, services must retrieve the data from a database (that's this pattern). This chapter focuses on the trade-offs and implications of the Key-Based Event Contract pattern.

Table 23-1 summarizes the metadata for the Key-Based Event Contract pattern.

Table 23-1. Metadata for the Key-Based Event Contract pattern

CATEGORY	METADATA
CONTEXT:	Contracts, Event-Driven Architecture, Message-Based Architectures, Microservices
IMPROVED:	Bandwidth Usage, Data Consistency, Data Integrity
DEGRADED:	Adaptability, Elasticity, Evolvability, Responsiveness, Scalability, Throughput

How Much Data Should I Include in the Event Payload?

In event-driven architecture, when a service completes its processing, it typically triggers derived events that broadcast the outcome to the rest of the system. For example, suppose a customer places an order in a typical order processing system. An `Order Placement` service receives the order. This is the *initiating event*, the event that kicks off a workflow. Once the `Order Placement` service validates the order and inserts it into the database, it triggers an `order_placed` derived event to the rest of the system advertising that it finished placing the order, then returns the order ID to the user.

When triggering the `order_placed` derived event, the `Order Placement` service must decide if it should send all of the data in the event payload, or just the key(s). This scenario is illustrated in Figure 23-1.

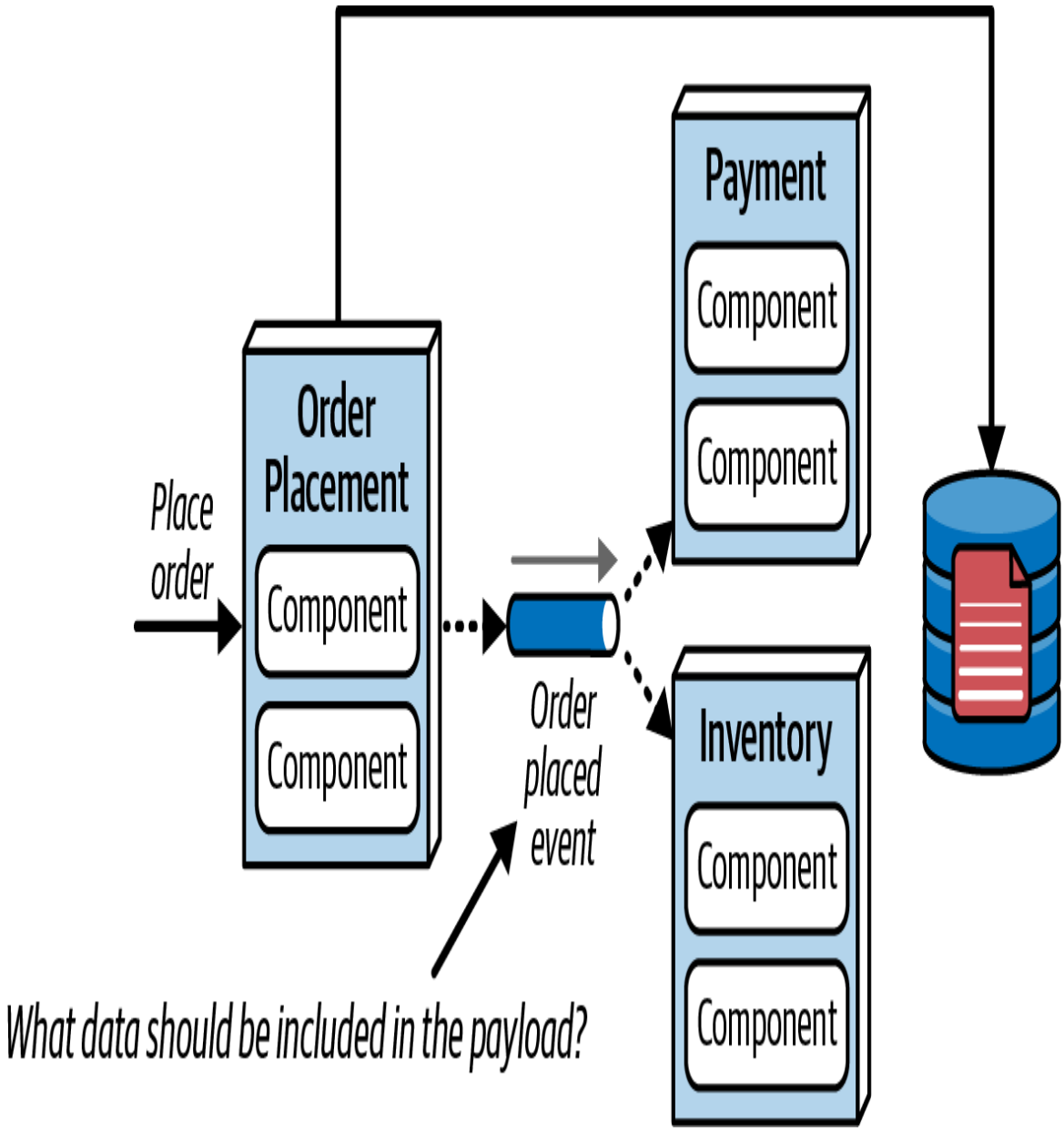


Figure 23-1. What data should be sent in the triggered derived event?

While this pattern applies to any form of remote communication (such as REST, messaging, or event processing), event-driven architectures are the most affected by this pattern, and are thus the focus of this chapter.

Applying the Pattern

In the Key-Based Event Contract pattern, the sending service only includes the key(s) of the data recently processed in the event payload (in this case,

just the `orderId`). All responding services must retrieve the data they need from the database based on the payload key.

Using this pattern, the `order_placed` event triggered by the `Order Placement` service would only contain the `orderId` of the order just processed. All responding events would then need to query the database to get the data they need to process the event:

```
{
  "orderId": 98765
}
```

Unlike data-based event contracts (see Chapter 14), key-based contracts aren't typically associated with a schema because of their simplicity. Instead, they use a *loose contract*, a contract without a schema consisting only of name-value pairs (again, in this case, just the `orderId`).

Figure 23-2 illustrates how this pattern works. The `Payment` service, upon responding to the event, must query the database to get the order amount and the customer's payment information so it can apply payment for the order. The same is true for the `Inventory` service; when responding to the event, it must query the database to get the item ID and quantity so it can adjust the inventory.

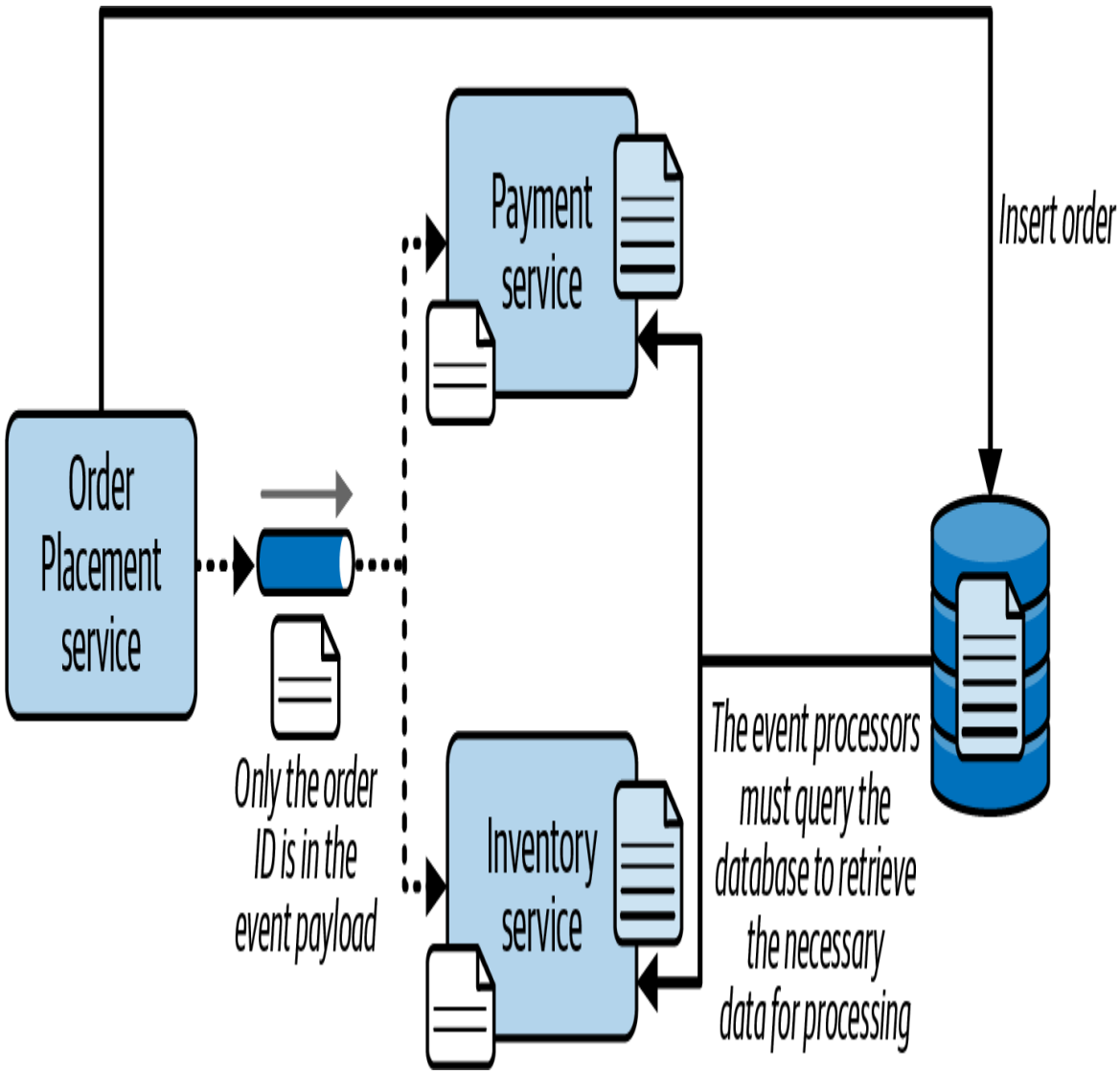


Figure 23-2. Only relevant keys are included in the event payload

Data Access Considerations

A major consideration for architects using the Key-Based Event Payload pattern is whether the responding services can access the data they'll need to process the event. Data access usually isn't an issue when using a monolithic database topology, where all services share a single database, or the domain database topology, where services within a particular domain all share a domain-based database (unless the needed data is in a different domain). However, it can be a problem when using the database-per-service topology, where each service owns its own data within a bounded context.

The bounded context means that other services can't access the data directly, so to get it, responding services must make remote calls to those owning services.

Consider a scenario where the `Order Placement`, `Payment`, and `Inventory` services all own their own data within a bounded context, using the database-per-service topology. If the `Order Placement` service triggers an `order_placed` event, it will send only the `orderId` as the data in the payload (as specified by the Key-Based Event Contract pattern). The `Payment` and `Inventory` services need data for processing the order, but don't have direct access to it. Each service must synchronously *ask* the `Order Placement` service for the data. Not only does this arrangement couple these services together, it also incurs significant latency by making remote calls to the `Order Placement` service.

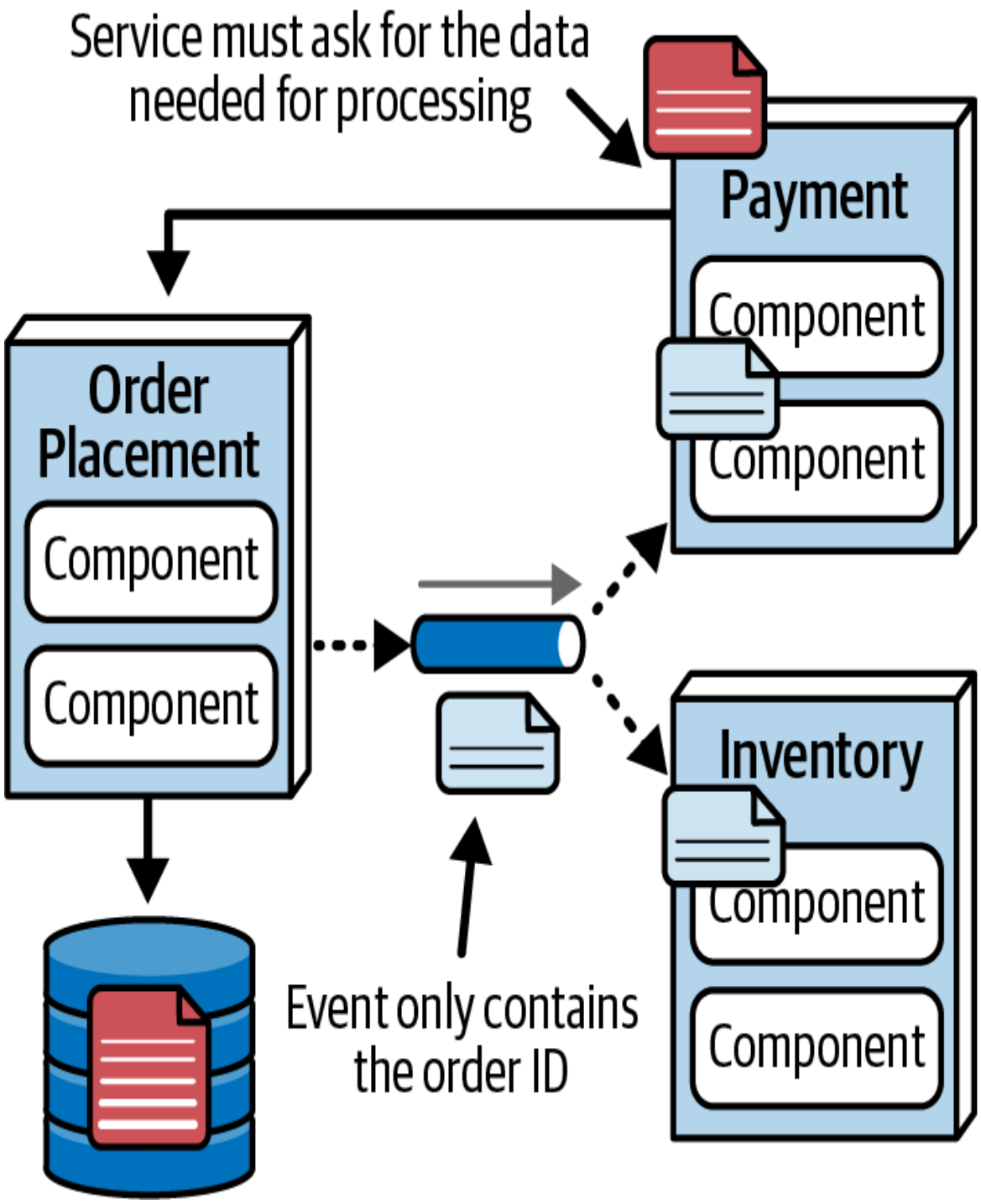


Figure 23-3. Using key-based event contracts might require responding services to call other services to get the event data

When this situation occurs, as shown in Figure 23-3, architects will typically switch to the closely related Data-Based Event Contract pattern (see "Influences" for more about such scenarios).

Not *all* event contracts have to utilize this pattern. Some events could leverage the Data-Based Event Contract pattern to pass on some or all of the data needed for processing, particularly in areas of the system where scalability and responsiveness are primary concerns or where the relevant data is not easily accessible.

Anemic Events

The Key-Based Event Contract pattern is a viable option for insert and delete actions, such as responding to an `order_placed` or an `order_canceled` event. However, for updates, derived events that use a key-based payload usually don't contain enough information for the responding services to process the event.

Let's consider another scenario. Clark Kent, a customer, updates the address in his profile: he's moving from Smallville, Kansas, where he grew up, to Metropolis, New York, where he will start his new job at the *Daily Planet*. Using the Key-Based Event Contract pattern, the `Customer Profile` service would trigger a corresponding `profile_updated` event containing the `customerId` as its payload:

```
{
  "customerId": 123
}
```

The problem here is that other responding services have no idea what data was changed or what the old and new values are. If performing updates while using the Key-Based Event Contract pattern, we recommend passing the relevant key(s) *and* including the old and new values as additional attributes in the payload:

```
{
  "customerId": 123,
  "oldAddress": "321 Maple Street, Smallville, KS",
  "newAddress": "344 Clinton Street, Apartment 3-D, Metropolis,
NY"
}
```

With this modified event payload, the responding services now have the minimal information necessary about what data has changed, and can query the database if they need additional information about the customer for processing.

Trade-Offs

This pattern has its positives and negatives, listed in the following sections. Architects should carefully consider these trade-offs and compare them against those of the Data-Based Event Contract pattern to choose the most appropriate pattern for event payloads.

Pros

The advantages of using the Key-Based Event Contract pattern are illustrated in Figure 23-4.

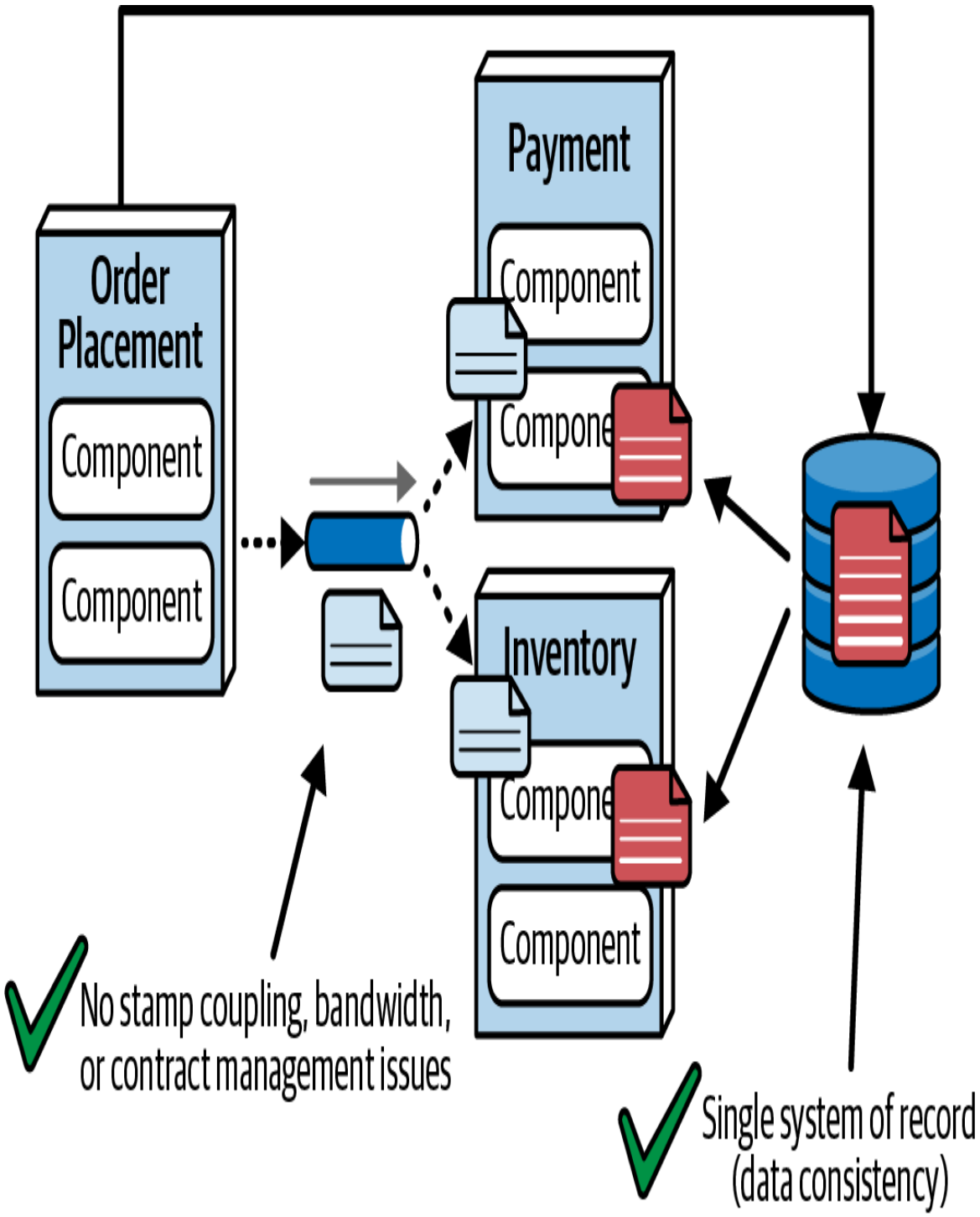


Figure 23-4. Advantages of the Key-Based Event Contract pattern

Because this pattern specifies only the key(s) in the event payload of processed data, architects don't need to be concerned about contract maintenance, versioning, and stamp coupling. Key-based event contracts change rarely, if ever. Having key-based event contracts also reduces

bandwidth usage due to the small payload size, and avoids the Stamp Coupling antipattern (see Chapter 31). Furthermore, overall data integrity and consistency are improved because services reacting to the event must query the event data from a database (the system of record), and therefore are always guaranteed to have the latest information.

Cons

The downsides of using the Key-Based Event Contract pattern are illustrated in Figure 23-5.

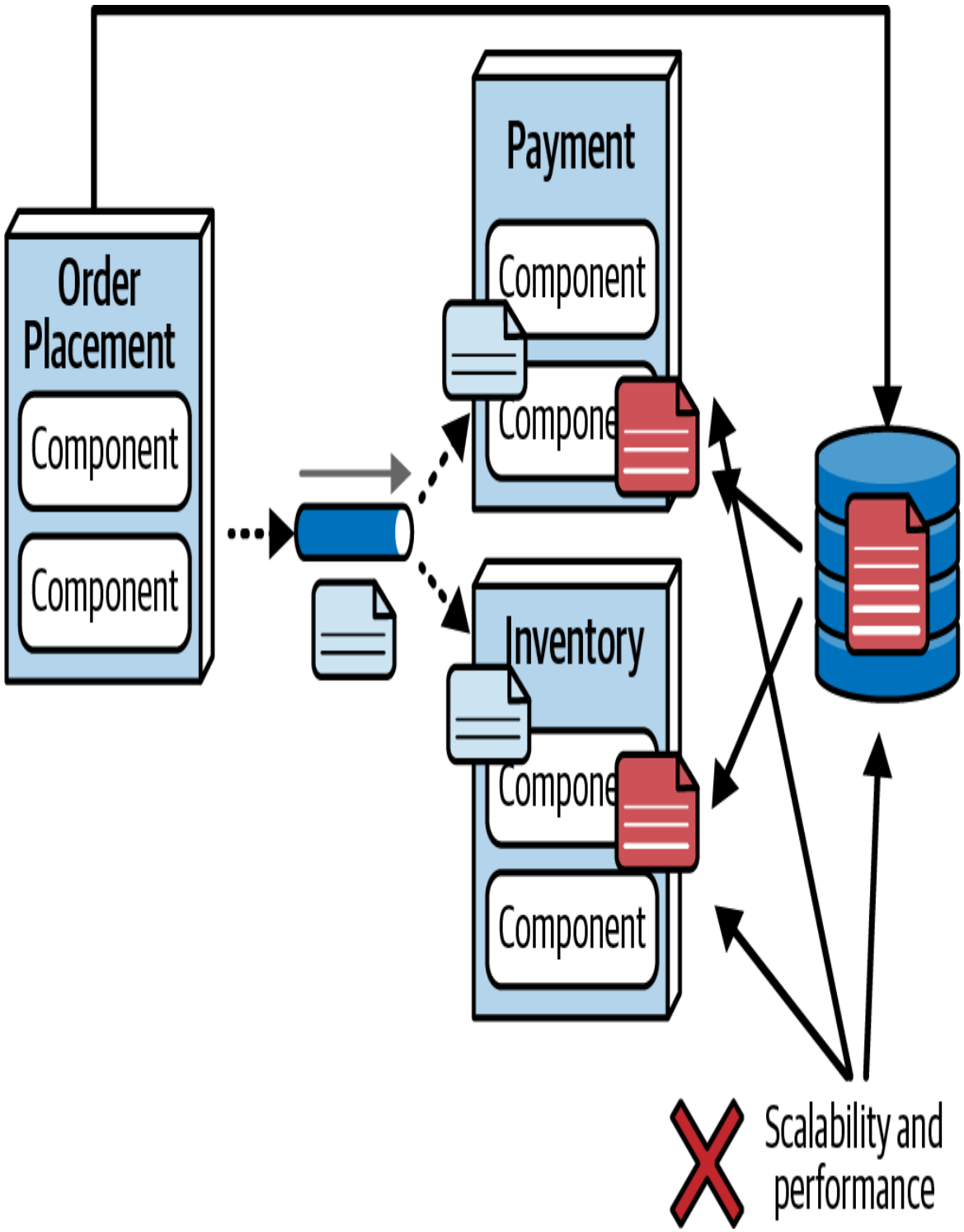


Figure 23-5. Disadvantages of the Key-Based Event Contract pattern

Using key-based event contracts means responding services must always make a database call to retrieve the event data, which, particularly in high-

volume systems, can negatively impact operational capabilities like scalability, elasticity, responsiveness, and throughput.

The Key-Based Event Contract pattern assumes that responding services will have access to the corresponding data based on the payload keys. In microservice architectures, the data needed for processing is usually in another service's database within its bounded context, requiring the responding service(s) to make synchronous calls to those services, as shown in Figure 23-3. Not only can this degrade responsiveness, it also increases service coupling.

Architects can sometimes mitigate this degradation in responsiveness by leveraging in-memory replicated caching, particularly with low data volumes and low data volatility (see the Cache Aside pattern in Chapter 8 for more details about this option). However, while this option improves responsiveness, it could degrade data consistency.

Comparison Chart

Because the Key-Based Event Contract pattern is directly related to the Data-Based Event Contract pattern (see Chapter 14), we've chosen to build a comparison chart that shows their qualitative differences. These comparison ratings, shown in Table 23-2, assume that data is readily available to responding services through simple database calls. See "Influences" for more about times when that data might not be easily accessible.

Table 23-2. Comparison chart for the Key-Based Event Contract pattern

Capability	Key-Based Contract	Data-Based Contract
IMPROVED		
Bandwidth Usage	★★★★★	★
Data Consistency	★★★★★	★★★
Data Integrity	★★★★★	★★★
DEGRADED		
Adaptability	★★★	★★★★★
Elasticity	★★★	★★★★★
Evolvability	★★★	★★★★★
Responsiveness	★★★	★★★★★
Scalability	★★★	★★★★★
Throughput	★★★	★★★★★

The following sections describe each of the capabilities found in Table 23-2.

Improved

The capabilities that are improved when applying this pattern include:

Bandwidth Usage

Key-based event contracts are very small, typically only containing a single key-based attribute (or sometimes a few). Thus they consume very little bandwidth, making their network costs much cheaper than when using data-based event contracts.

Since responding services using this pattern must query the database to get the information they need, which uses additional bandwidth, architects might wonder if key-based event contracts offer any real cost and bandwidth savings over data-based event contracts. They do: because the services only query for the data they need for processing, they consume much less bandwidth than a full data-based event contract would.

To illustrate this point, consider a scenario where the `Order Placement` service triggers an `order_placed` event containing all of the order data. This data-based event payload contains 45 attributes and 500 KB of data. The system receives 500 orders per second, which translates to 250,000 KB per second of bandwidth usage. However, the `Inventory` service only needs 30 bytes of data (the order ID and quantity). If the system processes 500 orders per second, that's only 15 KB per second of bandwidth usage — a big difference.

If the architect were to use a key-based event contract, then the `order_placed` event would only contain a small amount of data (say, 40 bytes). At 500 orders per second, that uses only 20 KB of bandwidth instead of 250,000 KB. However, the `Inventory` service must now query the database for the `itemID` and `qty`, which will consume 30 bytes of bandwidth. At 500 orders per second, that's 15 KB of bandwidth. Together, these two actions total 35 KB per second of bandwidth — still significantly lower than with data-based event payloads.

Data Consistency and Data Integrity

Because the event payload doesn't contain any data (just keys), data related to the event is kept only in one place — the database. This provides much better data consistency and integrity than data-based event payloads offer.

Degraded

The capabilities that are degraded when applying this pattern include:

Adaptability

This capability is degraded when using the Key-Based Event pattern because services reacting to the event must query the event data from a database, and those services may not have access to the data they need, particularly as those services change based on new requirements (see "Influences").

Elasticity and Scalability

Each responding service must use a database connection when querying the data from a database based on the keys in the event payload, which can limit scalability and elasticity. Systems at high scale are particularly impacted by this negative trade-off. Architects should understand the system's scalability and throughput needs, and weigh those quantitative requirements against the number of database connections available within each service instance to determine the viability of this pattern.

Evolvability

This capability is degraded because new services responding to an event might not have direct access to the database containing the event data. This makes it more difficult to evolve systems, particularly when using microservices.

Responsiveness

Responding services incur network and data latency when querying the database, resulting in some loss of responsiveness. This latency can

significantly degrade responsiveness if the responding services have to send synchronous requests to other services to obtain the data from the event keys (see "Influences").

Throughput

Overall throughput is degraded because all responding services must query a database to retrieve event information and incur the latency involved with database access. Furthermore, consistent database use can overload a database, causing further latency, hence less throughput.

Influences

A team's choice of database topology can significantly influence this pattern and possibly degrade certain architectural capabilities. The star rating chart in Table 23-2 shows the qualitative differences between the Key-Based and Data-Based Event Contract patterns when the data is easily accessible from responding services. However, using either a domain-based database topology or database-per-service topology would change these qualitative capabilities, as Table 23-3 shows.

Table 23-3. Comparison chart for the impact of database topologies on Key-Data-Based Event Contract patterns

Capability	General Ratings	Monolithic Database	Domain Database	Dat per·
IMPROVED				
Bandwidth Usage	★★★★★	★★★★★	★★★★	★★
Data Consistency	★★★★★	★★★★★	★★★★★	★★
Data Integrity	★★★★★	★★★★★	★★★★★	★★
DEGRADED				
Adaptability	★★★	★★★	★★	★
Elasticity	★★★	★★★	★★	★
Evolvability	★★★	★★★	★★	★
Responsiveness	★★★	★★★	★★	★
Scalability	★★★	★★★	★★	★
Throughput	★★★	★★★	★★	★

The following sections describe each of these topologies and how they affect the system's overall capabilities, as compared in Table 23-3.

Monolithic Database Topology

In the monolithic database topology (Figure 23-6), all services share the same physical database and schema. This topology is used in monolithic architectures, service-based architectures, space-based architectures, and in some cases event-driven architectures.[1]

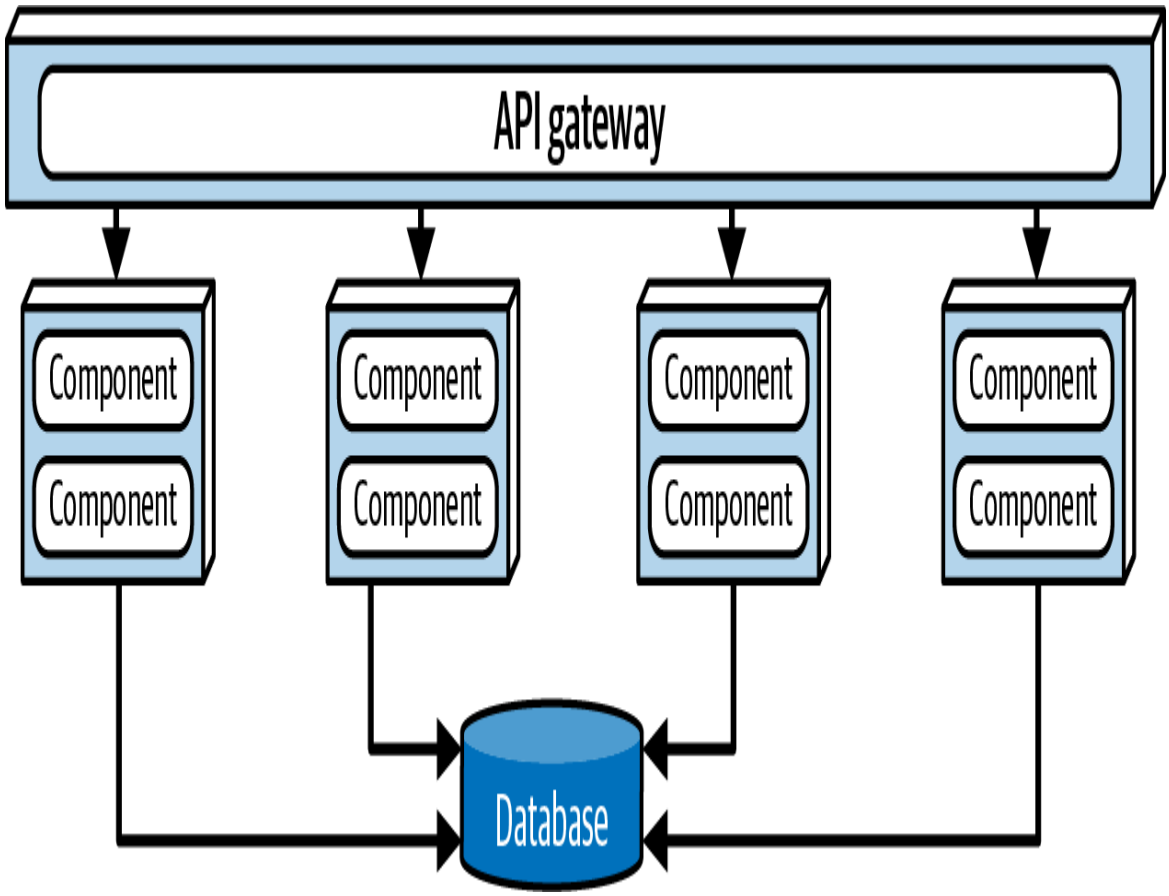


Figure 23-6. In the monolithic database topology, all services share the same database

The Key-Based Event Contract pattern works well with the monolithic database topology because all services have access to a central database to get the necessary data they need for processing. When using this topology, the base star ratings for key-based event contracts (shown in Table 23-3) apply.

Domain-Based Database Topology

With a domain-based database topology, one or more services are grouped by domain and share a database dedicated to that domain. This topology, illustrated in Figure 23-7, is a popular choice among architects using service-based architectures, space-based architectures, and event-driven architectures.

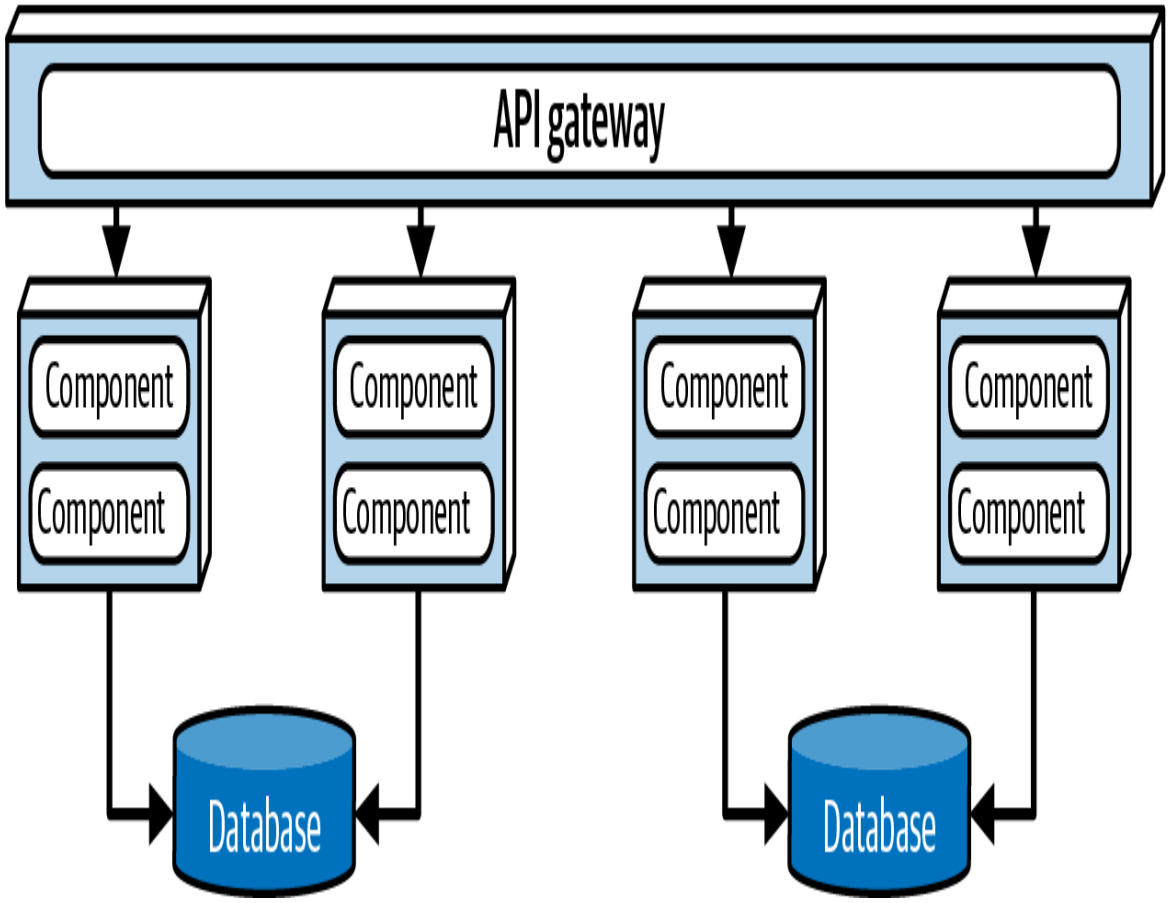


Figure 23-7. Services within a particular domain all share the same domain database

The Key-Based Event Contract pattern works well with the domain-based database topology when all of the data needed to process the event is in the corresponding domain database. Problems can arise when an event is triggered from one domain and services in another domain must respond.

For example, let's say that in our order processing system, a `Payment` service applies Clark's payment to the order within the `Ordering` domain. It then triggers a `payment_applied` event containing only the

`OrderID` as the key in the payload. The `Order Fulfillment` service, which is located within the `Order Processing` domain, responds to the event, but to get the necessary order details for processing, it must then make a remote call to the `Order Placement` service in the `Ordering` domain. This kind of synchronous cross-domain communication is highly undesirable because it couples the domains together, forming a single architectural quantum between them. It also degrades the architectural capabilities normally improved by the Key-Based Event Contract pattern.

Notice in Table 23-3 that when using a domain-based database topology, scalability and responsiveness each lose one star due to the possibility that synchronous remote calls will be required to obtain needed data. Contract management and bandwidth usage are also degraded by one star for the same reason: synchronous calls to other services require a contract and consume additional bandwidth. We only reduced these capabilities by one star because, typically, most of the data that responding services need is contained within the domain database, with only a few events needing data from other domains.

For events that cross a domain boundary, architects usually switch to a data-based event payload so that responding services in another domain have all the available data they need for processing.

Database-Per-Service Database Topology

In the database-per-service topology (also known as the *dedicated database* topology), each service owns its own database (or *schema*) in a strict bounded context. Only the owning service within the bounded context can access that data. Microservices uses this topology, and so do service-based architectures when each domain service owns its own data. Occasionally event-driven architecture uses it too — generally when the data is highly contained within the service boundary and is not needed by other services. This topology is illustrated in Figure 23-8.

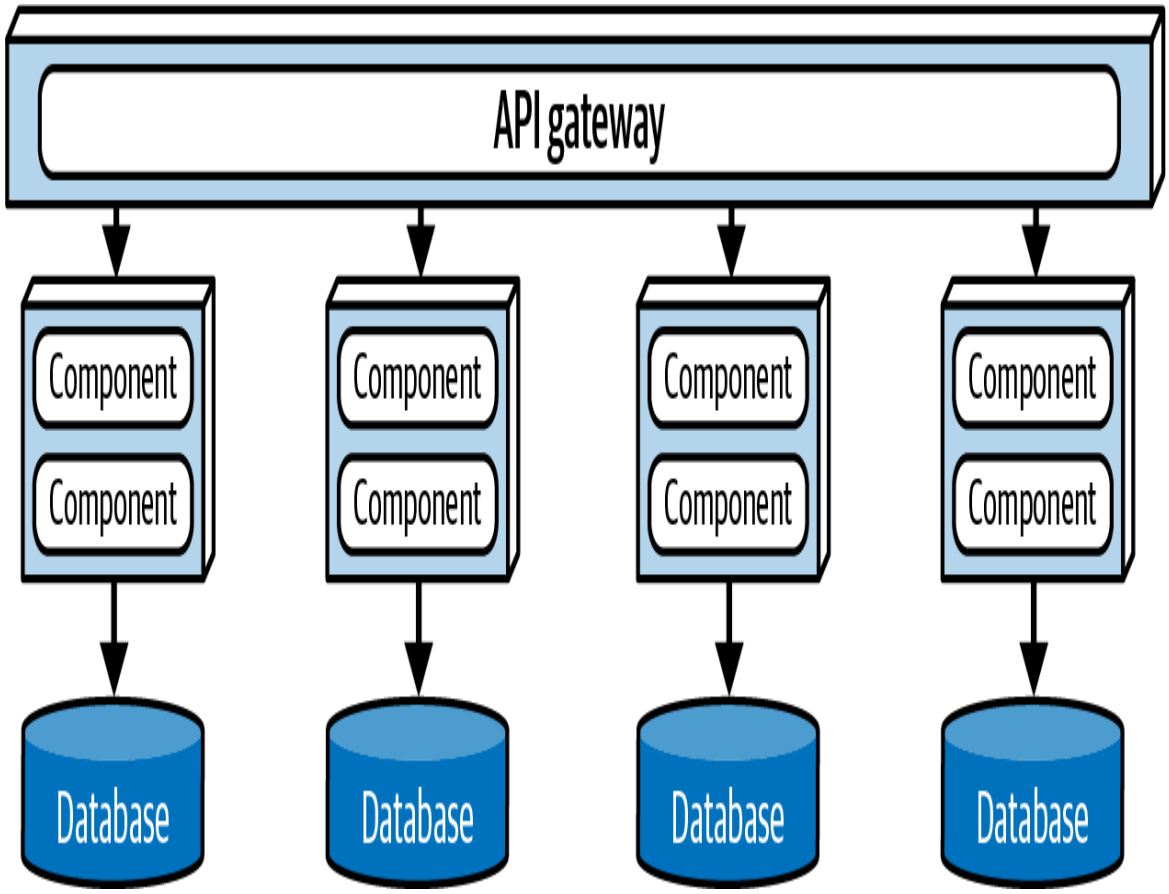


Figure 23-8. Each service owns its own data within a strict bounded context

Key-based event payloads are not well suited for the database-per-service topology when data needed to process the event is located in other bounded contexts. Responding services will need to make synchronous remote calls to those services to get the necessary data, with a significant impact on the architectural capabilities shown in Table 23-3.

Using a key-based event contract with the database-per-service topology significantly degrades scalability, responsiveness, contract management, and bandwidth usage. This is because responding services will *usually* need to make remote calls to other services to obtain data. We lowered our ratings for these capabilities by two stars (as opposed to just one star for the domain-based database topology) due to the frequency and probability that responding services will need to make remote calls to other services.

Because of these degraded qualitative ratings, architects usually switch to a data-based event contract payload when using the database-per-service

topology so that responding services have all the available data they need for processing.

How to Govern This Pattern

Architects can verify whether this pattern is the most appropriate one to use for event payloads by measuring and monitoring the following metrics:

Data access

If responding services don't have direct access to the database, architects might want to consider using the related Data-Based Event Contract pattern (see Chapter 14).

Responsiveness

Inconsistent or degraded response times might indicate that the database is overwhelmed or the connection pool is saturated. Service responsiveness is relatively easy to measure and track, and is a good indicator that the system isn't performing or scaling enough, which might point to the use of key-based event payloads as a possible problem.

If responsiveness is a critical concern in the system, architects can also write fitness functions to measure and analyze the time spent querying the database by timing the query and streaming this observability metric:

```
...
var start_time = get_time()
query_database(key)
var query_duration = get_time() - start_time
stream_metric(query_duration)
...
```

Architects can then track these query durations to determine whether the database queries are degrading responsiveness too much.

Connection waits

Architects can write fitness functions or use database monitoring tools to look for connection waits to the database. Because responding services must query the database for the data, services can possibly saturate their connection pools when responding to events. To get a more fine-grained analysis of connection waits on a per-service basis, architects can measure the amount of time it takes to obtain a connection to the database in each service:

```
...
var start_time = get_time()
var connection = get_connection()
var connection_duration = get_time() - start_time
stream_metric(connection_duration)
...
```

Architects can determine whether the database connection pool is saturated by observing how often the connection duration exceeds the average (say, 10 ms). If the connection pool is getting saturated, architects can attempt to add more connections to the pool or switch to the related Data-Based Event Contract pattern (see Chapter 14) for that particular event.

Database responsiveness

Architects can use database monitoring tools to analyze the health and overall responsiveness of the database, particularly when it's experiencing high user load. If the overall database responsiveness suffers when using key-based event contracts, the architect can try switching to data-based event contracts to see if that helps.

Related Patterns, Antipatterns, and Pitfalls

The following patterns are related to the Key-Based Event Contract pattern:

Cache Aside pattern (see Chapter 8)

When using the Key-Based Event Contract pattern, responding services may have to make remote calls to other services to obtain the data they need for processing. As mentioned earlier in this chapter, in-memory replicated caching might be a way to mitigate degraded responsiveness and scalability. The Cache Aside pattern discusses these options in detail.

Contextual Events pattern (see Chapter 12)

Contextual events are specific derived events that describe the outcomes of a service processing an incoming event. Architects must decide on the type of payload contract for those contextual events. In general, most architects applying the Contextual Events pattern choose to use key-based event contracts, sometimes supplementing the key with any necessary data based on the context of the derived event.

Data-Based Event Contract pattern (see Chapter 14)

When triggering an event from a service, the Data-Based Event Contract pattern is the alternative to the Key-Based Event Contract pattern. Architects should understand the trade-offs when choosing between them.

Extensible Events pattern (see Chapter 18)

The Extensible Events pattern describes the practice where a service triggers a derived event based on its processing outcome, even though no other services are responding to that event. It's a way of supporting architectural evolvability and overall agility. However, if, while using the Key-Based Event Contract pattern, the new service needs data in event payloads, the event contract (and the corresponding service that triggered the extensible event) will have to change. Alternatively, the

new service might just use the keys in the event payload to query the necessary data, in which case the Extensible Events pattern preserves the system's architectural evolvability and overall agility.

---

[1] See *Fundamentals of Software Architecture*, 2nd edition, for complete descriptions of these architectural styles.

Chapter 24. Microservice All the Things Pitfall

Microservices has been a very popular architectural style for the past decade. Many architects and development teams, whether creating a microservices architecture from scratch or migrating to one from a legacy system, assume that their entire system must be microservices. This chapter demonstrates that this assumption is a pitfall: it's never a good idea, even from the start.

Table 24-1 summarizes the metadata for the Microservice All the Things pitfall. While trade-offs are still involved when adopting this pitfall, no architectural capabilities are improved. Many are degraded when adopting this pitfall.

Table 24-1. Metadata for the Microservice All the Things pitfall

CATEGORY	METADATA
CONTEXT:	Microservices, Modularity, Service Granularity
IMPROVED:	None
DEGRADED:	Cost, Data Consistency, Data Integrity, Simplicity

Should I Make Everything a Microservice?

Done right, microservices is a powerful architectural style that supports high levels of the following architectural capabilities:

Maintainability

How easy it is to locate and apply changes to the system. Delineating and compartmentalizing a system's behavior by single-purpose microservices makes it easier for development teams to locate and maintain that behavior.

Testability

The ease and completeness of testing. Due to its fine-grained, single-purpose nature, microservices has a very small testing scope, making it easier to completely test the self-contained behavior.

Deployability

The amount of ceremony involved with releasing the software, the frequency of releases, and the overall risk involved in deployment. Because individual microservices are small and independent, deploying them requires minimal ceremony, can be done daily, and (depending on the amount of synchronous communication between services) carries much less deployment risk than larger services and monolithic systems.

Scalability

A function of system capacity and growth over time: as the number of users or requests increases, scalability measures how well responsiveness and performance remain constant. Services within a microservices architecture can scale at a *function* level, providing scalability only where it's needed.

Elasticity

The system can expand and respond quickly to unexpected or anticipated extreme loads (such as going from 20 to 250,000 users instantly). This is a superpower of the microservice architectural style due to its extremely low Mean Time To Start (MTTS). Elastic systems usually require a very fast response to immediate user load, and thus a fast MTTS. Microservices, being single-purpose and fine-grained,

typically have an MTTS of several hundred milliseconds, whereas large monolithic systems may measure MTTS in minutes.

Fault tolerance

The degree to which, when fatal errors occur, other parts of the system continue to function. Because services fail at a *function* level in microservices, when failures do occur, other parts of the system remain functional. Furthermore, the MTTS is so fast that, most of the time, users don't even realize a failure has occurred.

Evolvability

The ease with which a system can be extended with additional features and behavior. Adding new behavior is usually a matter of dropping a new service into the microservices ecosystem, with few or no changes to other parts of the system.

It's no wonder why microservices are so popular!

However, while these superpowers seem very attractive, microservices comes with Kryptonite[1] — things it's *not* so good at:

Overall cost

Microservices is an extremely expensive architectural style to develop and maintain. Since microservices are fine-grained, single-purpose services, it's not uncommon to have several hundred or even several thousand within a system — a force multiplier for licensing fees. Furthermore, because each microservice typically owns its own data within a bounded context, data licensing fees can be costly, too, not to mention the time and effort involved in breaking apart and modeling fine-grained data.

Complexity

Significant decisions in a microservices architecture include service granularity, contract type, communication protocols, workflow

management, and transaction management (just to name a few). Modeling data and forming bounded contexts are typically challenging tasks that require a lot of trial and error. All of this adds up to high levels of complexity.

Data consistency and data integrity

Because data is broken down and grouped into its smallest possible bounded context, it's difficult to keep that data consistent across multiple microservices, so data integrity and data consistency are degraded.

Why Is This a Pitfall?

Unfortunately, when creating or migrating a microservices architecture, architects have a tendency to make the *entire* system microservices. Take, for example, the typical order-placement and fulfillment system shown in Figure 24-1, which is implemented in a microservices architecture.

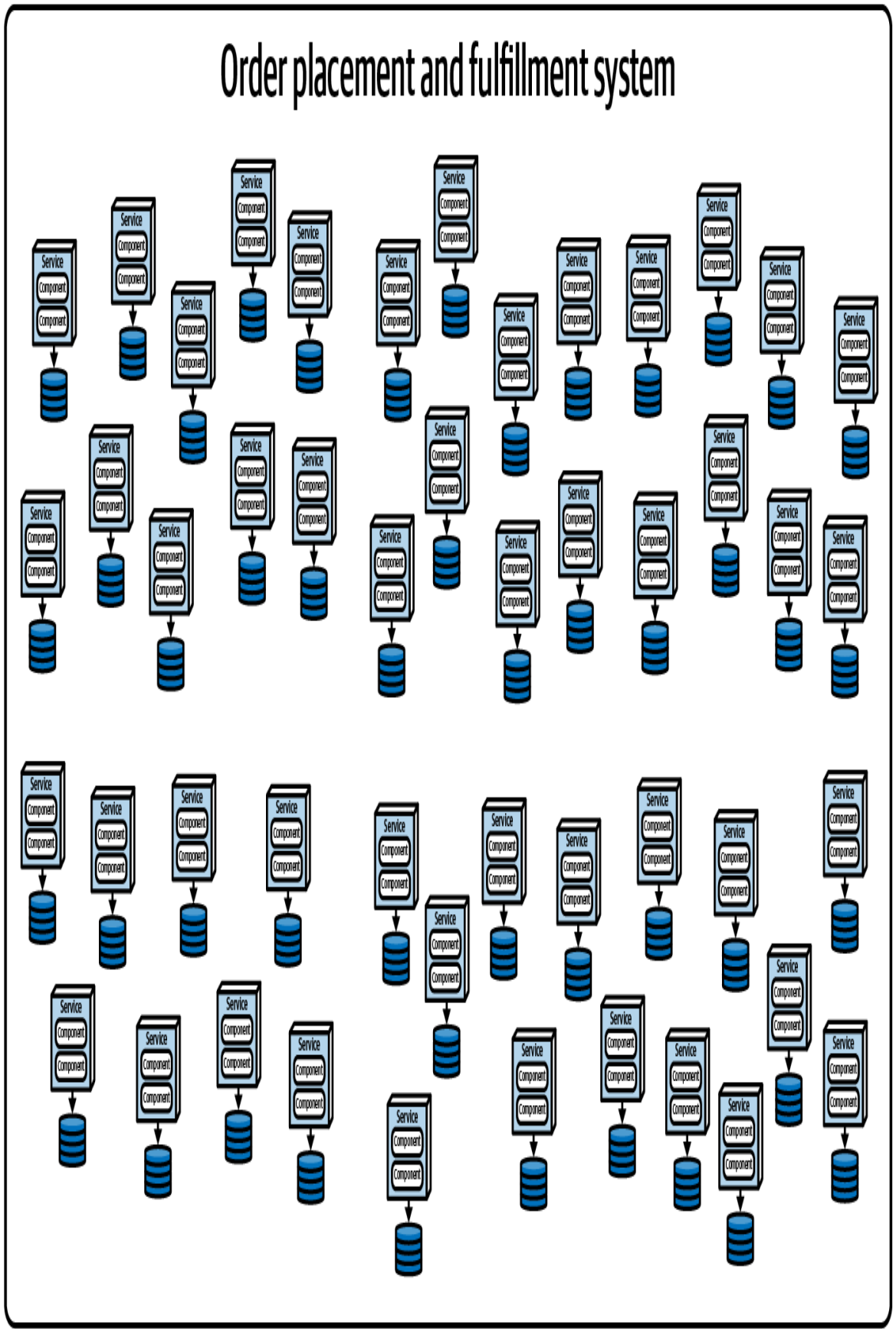

Order placement and fulfillment system
Service
Component
Component

Figure 24-1. An order placement and fulfillment system implemented in microservices

Notice that all of the behavior for this system is implemented as microservices. However, this system can be divided into four major domains, as illustrated in Figure 24-2: administration, warehousing, customer-facing ordering, and the backend order fulfillment.

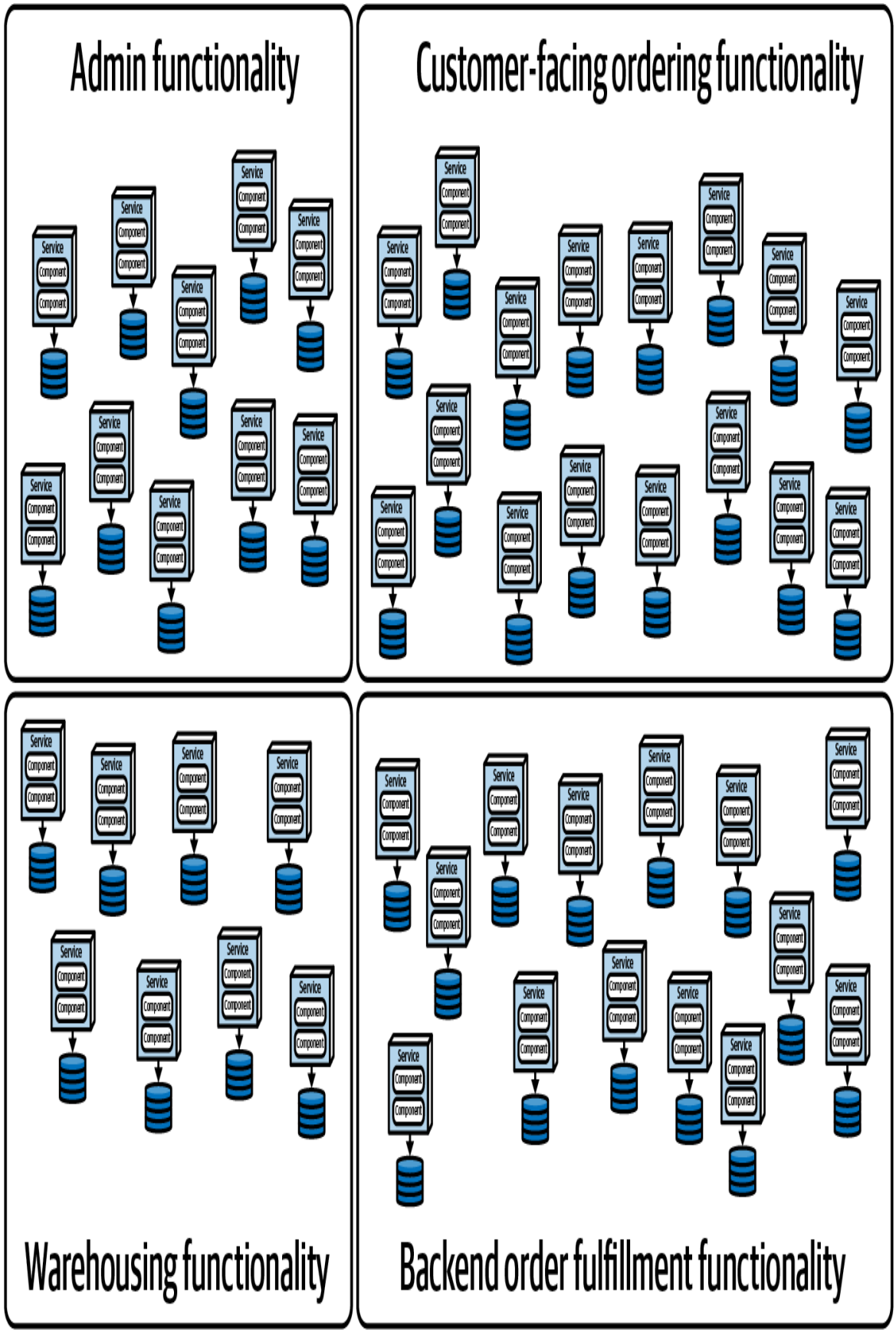

Admin functionality
Customer-facing ordering functionality
Warehousing functionality
Backend order fulfillment functionality
Service
Component
Component

Figure 24-2. The major domains of an order placement and fulfillment system implemented in microservices

The problem with making *everything* a microservice is that, in most cases, not all parts of the system need microservices' capabilities, as shown in Figure 24-3. Using microservices unnecessarily makes systems overly complex, requires more time to develop and maintain, increases costs, and degrades data integrity and consistency.

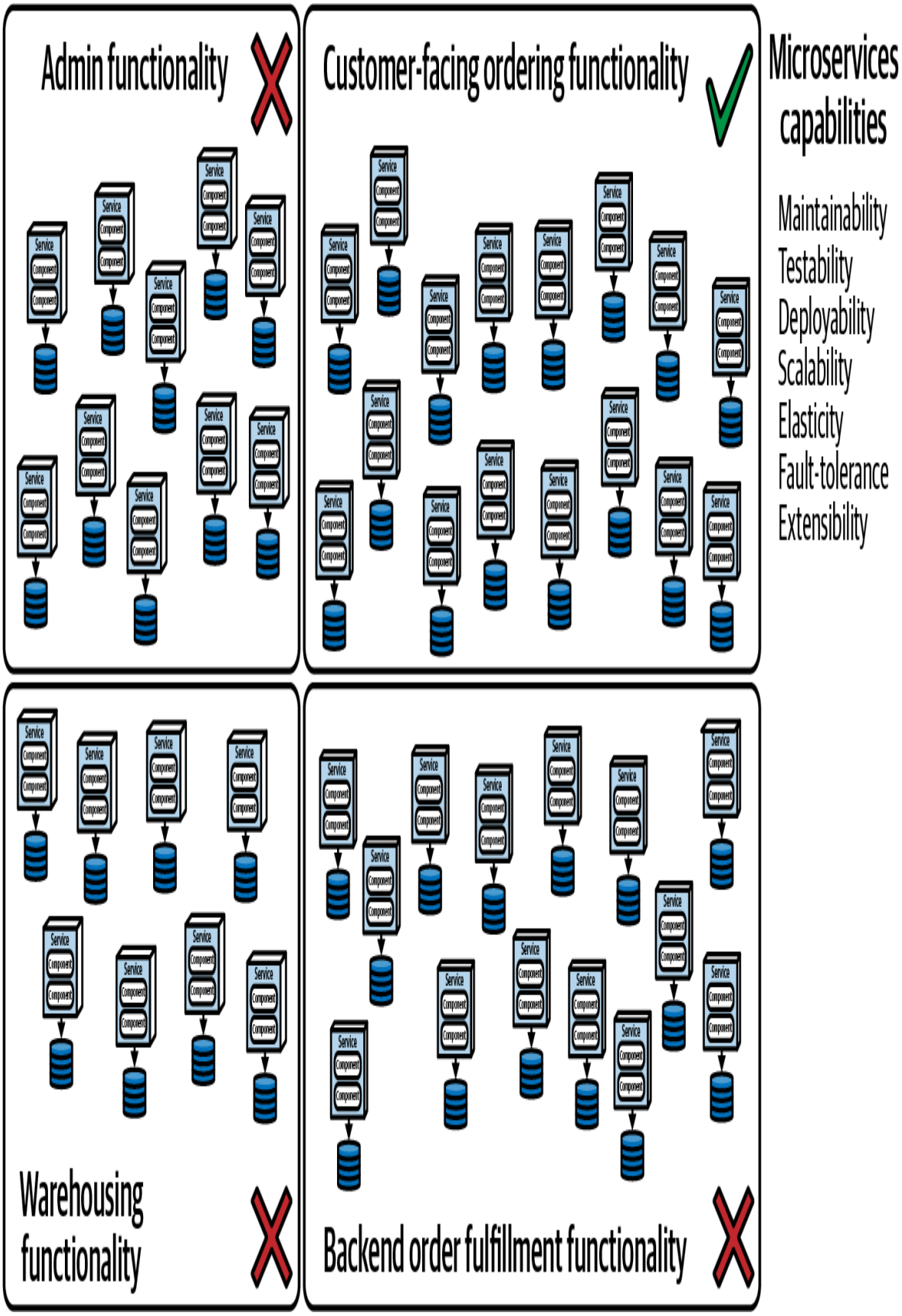

Admin functionality
Customer-facing ordering functionality
Microservices capabilities
Maintainability
Testability
Deployability
Scalability
Elasticity
Fault-tolerance
Extensibility
Warehousing functionality
Backend order fulfillment functionality
Service
Component

Figure 24-3. Certain parts of the system don't benefit from being microservices

Just because you *can* make everything microservices doesn't mean you *should*. Architects can avoid this pitfall by remembering a simple phrase: *not every portion of a system has to be microservices.* The following section shows techniques for how to avoid this pitfall.

How to Avoid This Pitfall

This pitfall is about arriving at the right level of modularity. Recall that *modularity* is about breaking up systems into parts (or developing new systems as separate parts), whereas *granularity* is about the size of those parts. We discuss service granularity in the related Grains of Sand pitfall in Chapter 21.

Whether migrating from a monolithic legacy system or developing a greenfield system, the easiest way to avoid this pitfall is to move to a service-based architecture first. Rather than fine-grained, single-purpose microservices, the services within a service-based architecture are coarse-grained and typically represent a specific domain or subdomain. Because of this, they're called *domain services* and usually contain all of the behavior for that particular domain (such as order fulfillment or order shipping). This architectural style can also use a monolithic database and UI, eliminating the need to break apart these architectural artifacts.

Using our ordering-system example, moving to or developing a service-based architecture first would yield the four separate domain services illustrated in Figure 24-4: one containing the administrative (admin) behavior, one containing the warehousing behavior, one containing the backend fulfillment behavior, and finally one containing the customer-facing ordering behavior.

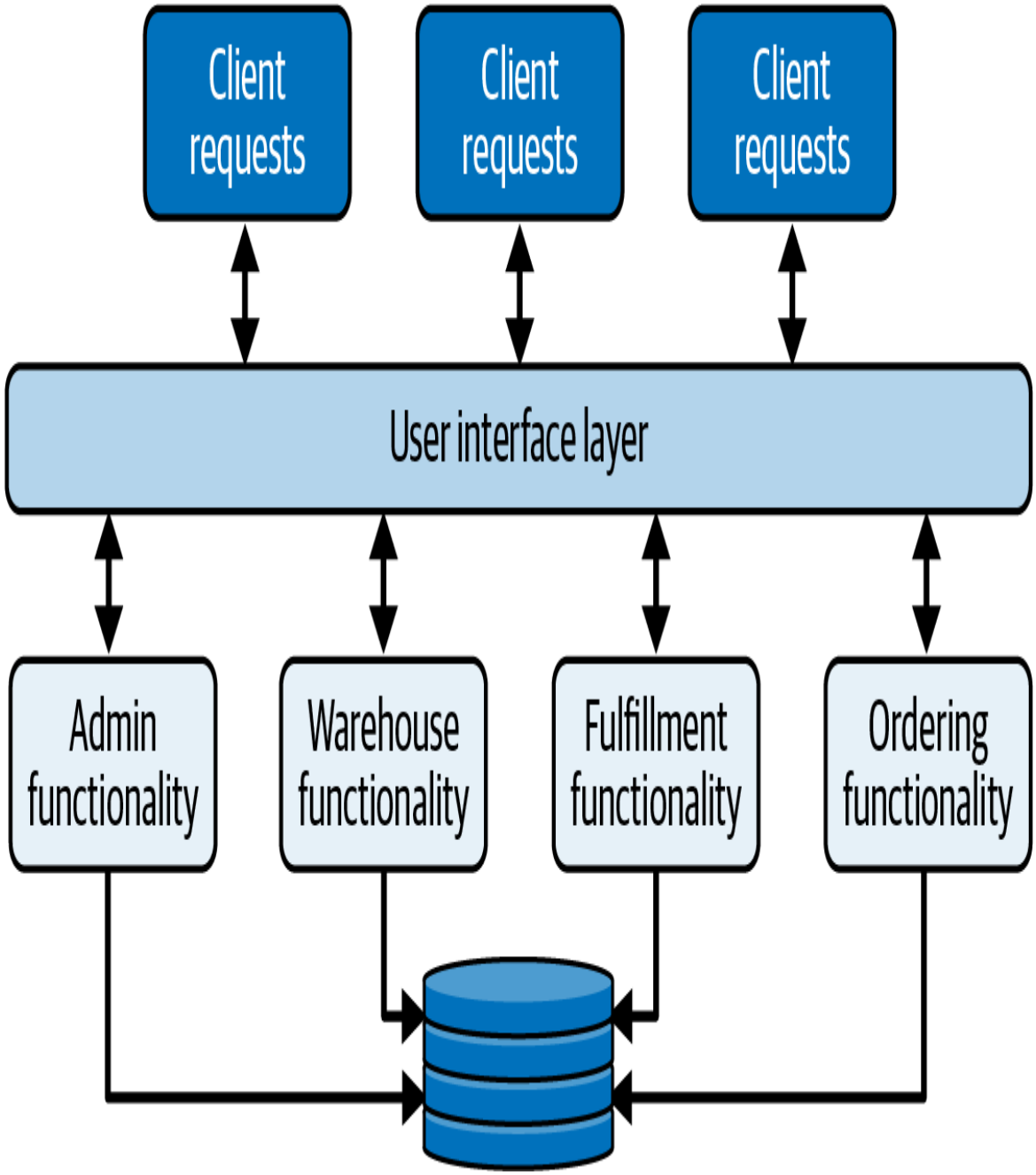
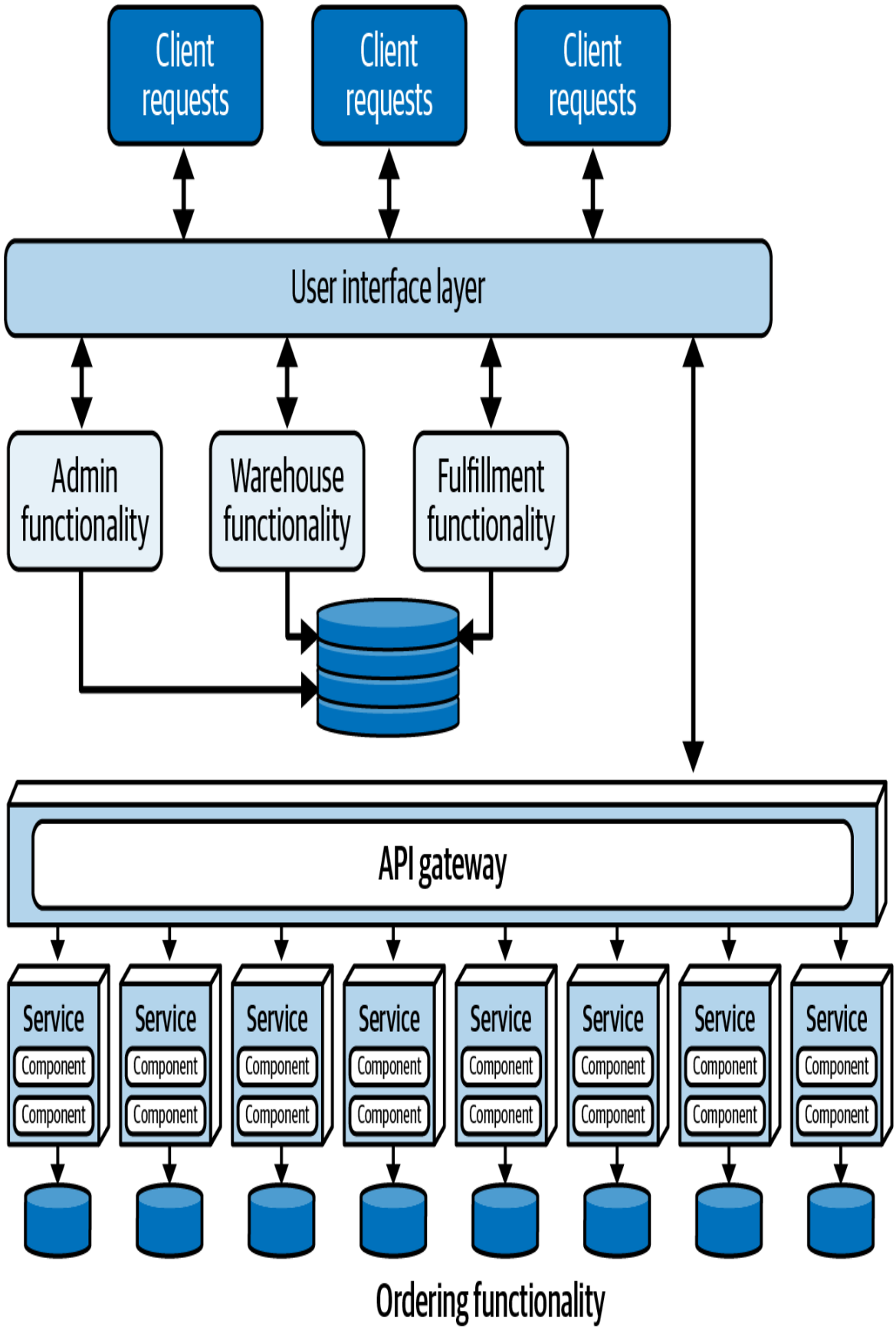


Figure 24-4. Moving to a service-based architecture first yields four domain services

The architect can now analyze each domain service to determine whether to break any of them into smaller subdomains to maximize their advantages. For example, the `Admin`, `Warehousing`, and `Backend Fulfillment` domain services typically don't change often, so they don't need high levels of maintainability, testability, deployability, and evolvability. Furthermore, since these services are not customer-facing, they

don't need high levels of scalability, elasticity, and fault tolerance. These particular domains don't need the superpowers of microservices — so they're perfectly fine remaining as domain services.

Now, consider the customer-ordering behavior contained in the `Ordering` domain service, shown in Figure 24-4. Because this behavior *is* customer-facing, new features should be added and any defects fixed as soon as possible. This behavior *would* thus benefit from the high levels of maintainability, testability, deployability, and evolvability typically found in a microservices architecture. Furthermore, being customer-facing, it would also benefit from improved scalability, elasticity, and fault tolerance. That makes this domain service an excellent candidate for breaking down further — into microservices. The resulting architecture, shown in Figure 24-5, is an example of a hybrid between service-based architecture and microservices that maximizes microservices' benefits while minimizing its negatives.



Ordering functionality

Figure 24-5. Only apply microservices where it's beneficial

Moving to a service-based architecture first allows architects to determine which behaviors need to be implemented as microservices and which can remain coarse-grained domain services, avoiding this common pitfall.

Since a big part of avoiding this pitfall is about knowing which portions of a system should be further broken down into microservices, architects can measure and track the following metrics:

Volatility

> Measuring the volatility of domain-based behavior — that is, how frequently it changes — helps architects determine whether they can improve a domain service's maintainability, testability, and deployability by breaking it down into microservices. Because domain services are generally coarse-grained and their testing scope encompasses all the behavior within the service, having to apply changes very frequently makes maintenance cumbersome and increases the risk of breaking something. Breaking that domain service apart can improve all three of these capabilities, improving overall agility.

Scalability

> Architects can measure the scalability of each domain service. If a service doesn't meet the requirements, the architects can break it apart further until it meets the scalability requirements.

Fault tolerance

> While microservice architectures provide excellent fault tolerance, domain services generally do not, due to their coarse-grained nature. By measuring domain services' fatal errors and service failures alongside their MTTS, architects gain insight into whether breaking them apart would improve fault tolerance and MTTS. See the related Grains of Sand pitfall in Chapter 21 for more information about service granularity and its impact on scalability and elasticity.

Trade-Offs

The First Law of Software Architecture, coined in the book *Fundamentals of Software Architecture*, 2nd edition, states that "Everything in software architecture is a trade-off." While this pitfall offers no clear advantages, teams nevertheless might be tempted to fall into it because of certain *perceived* advantages.

Pros

Teams might be tempted to embrace this pitfall due to microservices' promise of superior scalability, elasticity, and fault tolerance capabilities. Adopting this pitfall confers no advantages because, while these are advantages of microservices in general, parts of a system that *don't* require these capabilities don't realize these benefits — only the negative parts of microservices.

That said, we acknowledge that some systems might benefit from having all parts of it implemented as microservices. We urge teams to apply the techniques outlined in this chapter, and see which portions *can* benefit from the capabilities provided by the microservices architectural style. If it turns out that all parts do, then this is no longer a pitfall for those rare situations.

Cons

Teams adopting this pitfall and making every part of a system a microservice will typically find challenges in maintaining data consistency and data integrity across service boundaries. Workflows become unnecessarily complex, costs increase, and overall complexity increases without any benefit.

Comparison Chart

Table 24-2 summarizes the capabilities trade-offs associated with this pitfall. In the comparison chart, the Adopting Pitfall column refers to teams naively making everything in their system a microservice (as described in

this chapter), whereas Avoiding Pitfall refers to teams starting out with service-based architecture with coarse-grained domain services first, then breaking apart those domain services when needed.

Table 24-2. Comparison chart for the Microservice All the Things pitfall

Capability	Adopting Pitfall	Avoiding Pitfall
IMPROVED		
None	---	---
DEGRADED		
Cost	$$$$$	$
Data Consistency	★	★★★★★
Data Integrity	★	★★★★★
Simplicity	★	★★★★★

The following sections describe each of the capabilities found in Table 24-2.

Improved

This particular pitfall degrades capabilities — it does not improve any.

Degraded

This list contains the degraded capabilities when adopting this pitfall:

Cost

Making everything microservices is very costly in terms of design, development, data, and licensing. Remember, not every portion of a system needs to be a microservice!

Data Consistency and Data Integrity

Because microservice architectures require services to own their own data within a tightly formed bounded context, teams adopting this pitfall will find challenges with data modeling and maintaining overall data integrity and consistency. By keeping the domain behavior and data together in a combined coarse-grained service that *doesn't* require the capabilities of microservices, teams can significantly improve overall data integrity and consistency, and reduce the effort and cost of data modeling.

Simplicity

Microservices is perhaps the most complex architectural style to develop and maintain. By recognizing which portions of the system should be microservices and which ones shouldn't, teams can significantly reduce the system's overall complexity.

Influences

No external factors influence the trade-offs illustrated in the prior section.

Related Patterns, Antipatterns, and Pitfalls

The following patterns and pitfall are related to the Microservice All the Things pitfall:

Choreography pattern (see Chapter 9)

Making every part of the system a microservice usually requires architects to stitch the services together using Choreography. Architects

should be familiar with the Choreography pattern and the trade-offs associated with it.

Compensating Updates pattern (see Chapter 11)

When multiple microservices are involved in a single unit of work transaction, necessarily they're involved in a distributed transaction. As such, when errors occur within the transaction, the committed updates made by the successful microservices must be undone through compensating updates. Architects should understand the negative trade-offs and complexities associated with compensating updates when deciding to use microservices.

Grains of Sand pitfall (see Chapter 21)

While the Microservice All The Things pitfall largely has to do with modularity, the Grains of Sand pitfall has to do with granularity. Since domain services are usually coarse-grained, understanding the Grains of Sand pitfall helps architects better understand the appropriate size of a service, whether it's a domain service or a microservice.

Orchestration pattern (see Chapter 27)

As an alternative to Choreography, services can be coordinated using Orchestration, similar to a conductor directing an orchestra. Architects should be familiar with this related pattern when choosing between the Orchestration and Choreography patterns for service coordination.

---

[1] Kryptonite is a fictional green crystal that emits radiation and is Superman's primary weakness.

Chapter 25. Multi-Broker Pattern

Standard messaging, using queues or topics, works well for message throughput less than around 4,000 persisted messages per second. After that, architects must either use nonpersisted messages or streaming brokers (such as Kafka) to achieve higher throughput rates. Fortunately, architects have another option to increase message throughput: the Multi-Broker pattern.

Table 25-1 summarizes the metadata for the Multi-Broker pattern.

Table 25-1. Metadata for the Multi-Broker pattern

CATEGORY	METADATA
CONTEXT:	Event-Driven Architecture, High-Volume Processing, Message-Based Architectures, Microservices
IMPROVED:	Elasticity, Responsiveness, Scalability, Throughput
DEGRADED:	Cost, Message Processing Order, Simplicity

How Can I Improve Message Throughput and Capacity?

Most message brokers can handle a message throughput rate of 4,000 messages per second when using persisted messages. However, when the throughput rate exceeds this capacity, messages can back up, resulting in a

loss of responsiveness, scalability, and throughput, and often timeouts or error conditions in the message broker.

Consider an auditing or logging system that receives 3,000 messages per second from three message producers, as illustrated in Figure 25-1. Collectively, the message broker is receiving 9,000 messages a second, 5,000 more than its maximum capacity.

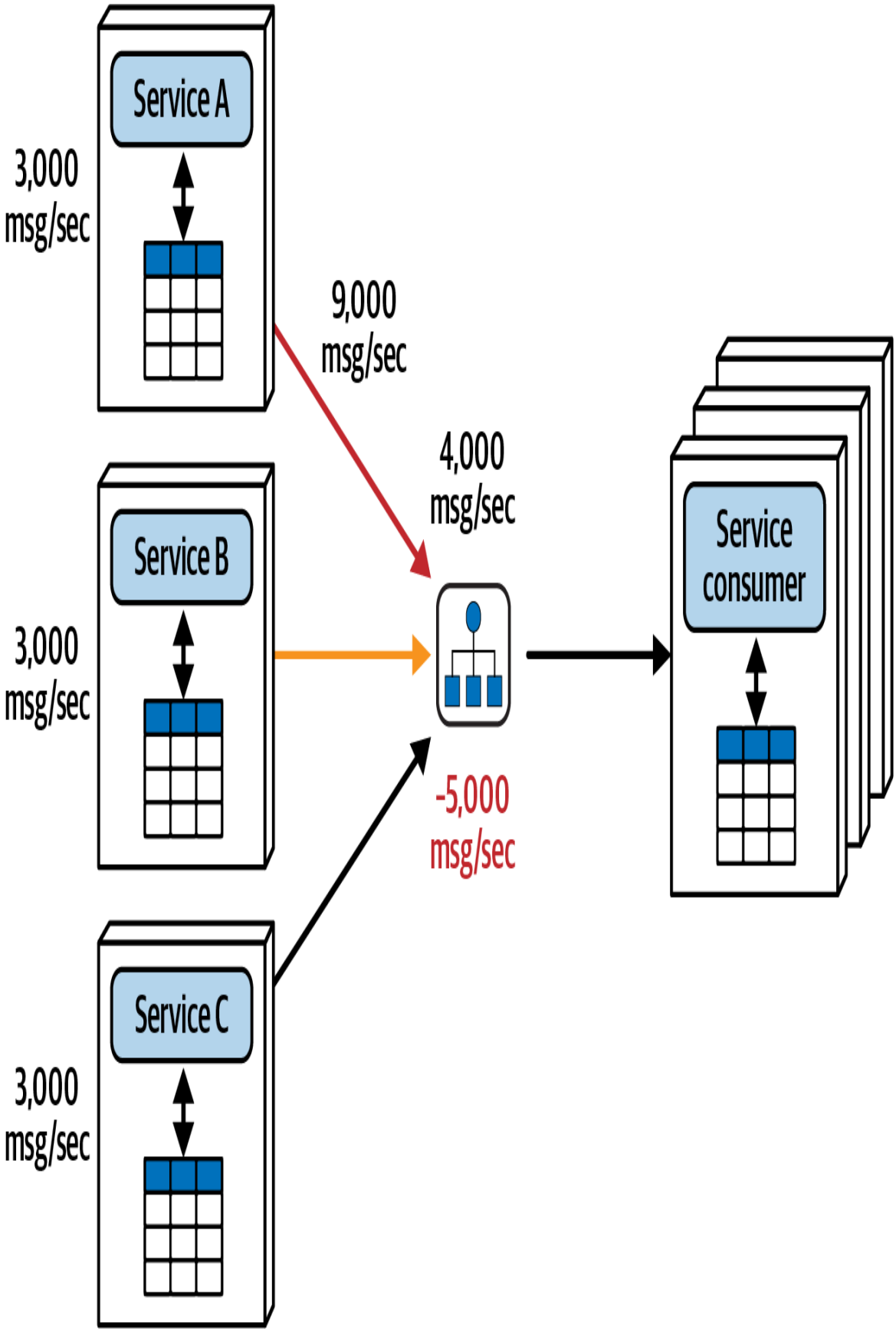

Service A
3,000
msg/sec
9,000
msg/sec
Service B
3,000
msg/sec
4,000
msg/sec
Service
consumer
-5,000
msg/sec
Service C
3,000
msg/sec

Figure 25-1. The number of messages sent by the message producers exceeds the capacity of the message broker

Clearly, this topology won't keep up with the message throughput. However, architects can significantly increase overall message throughput by adding multiple brokers, which is what this pattern is all about.

Applying the Pattern

To apply the Multi-Broker pattern to a basic messaging or event topology, architects create additional message brokers to handle excess throughput. With this pattern, *consumer groups* (the components or services processing the message) are assigned to each message-broker instance to ensure a consistent message throughput rate. For example, an architect could create two additional message brokers, as illustrated in Figure 25-2, producing a total throughput of 12,000 messages per second. This effectively creates a *surplus* of 3,000 messages per second.

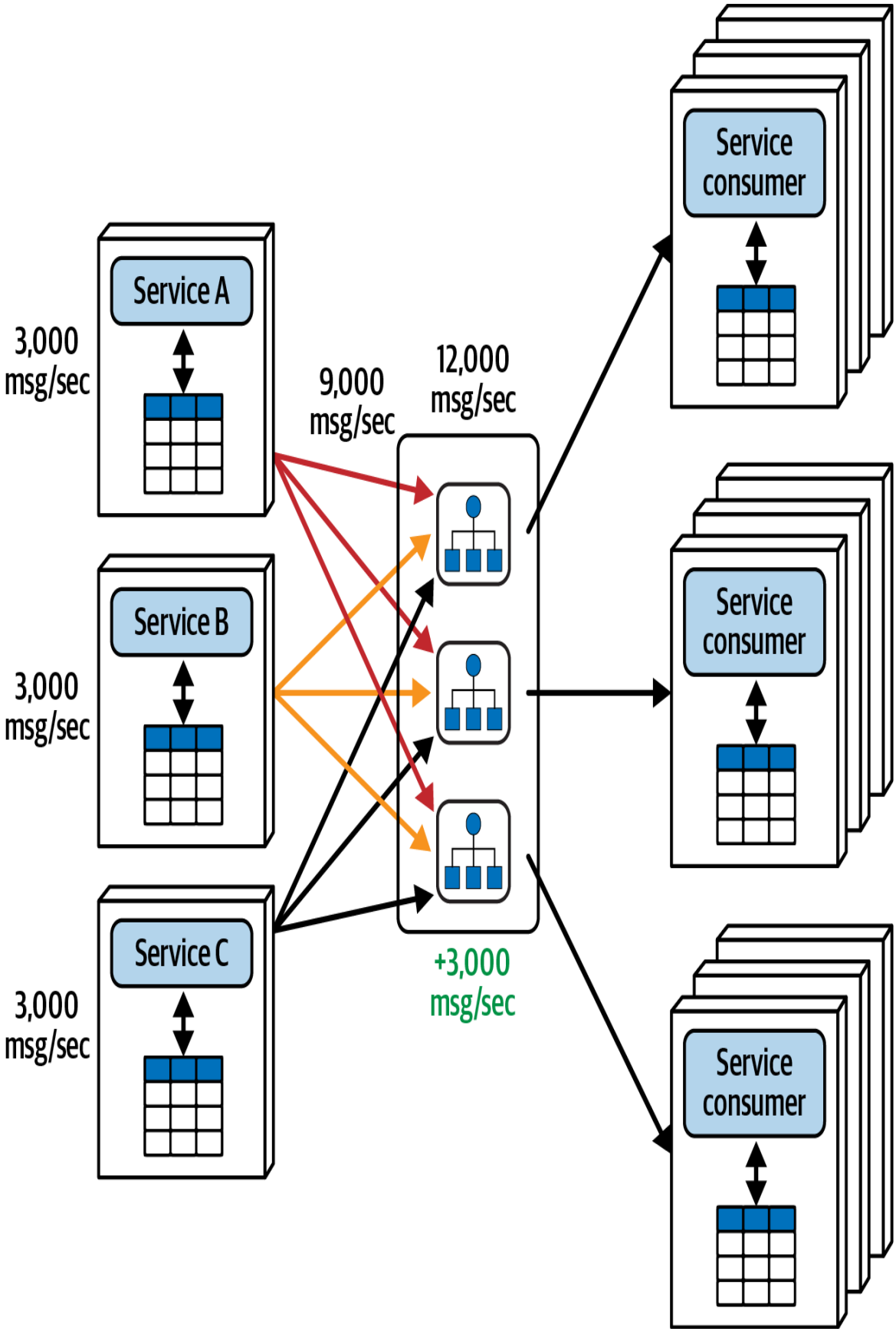

Service A
3,000
msg/sec
Service B
3,000
msg/sec
Service C
3,000
msg/sec
9,000
msg/sec
12,000
msg/sec
+3,000
msg/sec
Service
consumer
Service
consumer
Service
consumer

Figure 25-2. Adding additional message brokers increases the system's message-throughput capacity

Notice that the message brokers in Figure 25-2 are logically grouped together, similar to the partitions in a topic within Kafka. Each message-broker instance has the same queue or topic name to simplify this pattern and allow for flexibility. This way, architects or AI agents can dynamically assign consumer groups to a particular broker.

Message producers typically use a standard round-robin algorithm to alternate sending messages between different brokers. This distributes the message throughput load more evenly among the available brokers and the corresponding message consumers. The following pseudocode illustrates this technique:

```
var list msg_brokers = [broker:svr1:5672, broker:svr2:5672,
broker:svr3:5672]
var list connections = connect_to_brokers(msg_brokers)
var queue_name = "queue_1"
var index = 0

while true
   var msg = create_message()
   send_message(msg, queue_name, connections[index]) ❶
   add 1 to index
   if index = connections.size() set index = 0 ❷
end
```

❶ Even though this example uses three separate brokers, the queue name within each message broker remains the same, simplifying the infrastructure and code. Developers can modify this method so that if an error occurs while trying to connect to one of the message brokers, the code (or an AI agent) can dynamically modify the connection list to remove that connection until the message producers reestablish it. Architects (or their AI agents) can also dynamically expand and contract the connections list as needed based on the throughput needs.

❷ This simple round-robin algorithm cycles through each broker connection, starting over once it reaches the end of the list.

Using this pattern, architects can add message brokers as needed to accommodate message throughput.

Trade-Offs

The Multi-Broker pattern is a way to increase overall message throughput and scalability, and remove bottlenecks in the system. However, architects should be aware of the trade-offs of this pattern before deciding to adopt it.

Pros

The primary advantage of this pattern is to significantly improve operational capabilities such as scalability, elasticity, responsiveness, and overall message throughput. Systems that don't require high levels of these sorts of capabilities are unlikely to see any benefits from applying this pattern.

Cons

While this pattern improves message throughput, scalability, and responsiveness, its disadvantages include increased cost and complexity. Also, teams should be aware that applying this pattern will not preserve message processing order — another significant trade-off.

Comparison Chart

Table 25-2 summarizes the trade-offs associated with this pattern. The Without Pattern column in the table refers to using only a single broker for all messages.

Table 25-2. Comparison chart for the Multi-Broker pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Elasticity	★	★★★★★
Responsiveness	★	★★★★★
Scalability	★	★★★★★
Throughput	★	★★★★★
DEGRADED		
Cost	$	$$$
Message Processing Order	★★★★★	★
Simplicity	★★★★★	★★★

The following sections describe each of the capabilities found in Table 25-2.

Improved

The capabilities that are improved when applying this pattern include:

Elasticity and Scalability

Because this pattern improves overall message throughput, it also improves scalability and elasticity: message consumers can now process more messages, keeping up with the number of messages produced.

Responsiveness

When using this pattern, messages are processed immediately after being sent to the queue, improving the overall responsiveness of each message. This is particularly important when the message producer needs a response (such as requesting data or waiting on an acknowledgement that the information was processed).

Throughput

By increasing the number of message brokers, architects can significantly increase message throughput. Using data-driven techniques, configuration servers, or AI agents, architects can dynamically increase and decrease the number of message brokers (and corresponding message consumer groups) based on demand, similar to the techniques we describe in the related Supervisor Consumer pattern (see Chapter 33).

Degraded

The capabilities that are degraded when applying this pattern include:

Cost

The licensing costs associated with the message broker will likely increase when adopting this pattern because teams must create more instances of the message broker. Architects should thus weigh the benefits of increased message throughput using a cost-benefit analysis.

Message Processing Order

Because message producers send messages to alternating message brokers, the message processing and first in, first out (FIFO) orders will *not* be preserved when dequeuing and processing messages. Although message processing order is not normally preserved when using multiple consumers anyway (see the Thread Delegate pattern in Chapter 34 for how to handle this), FIFO order usually is. Architects

should pay attention to the needs of FIFO queues or message processing-order requirements before adopting this pattern.

Simplicity

Adding message brokers means adding complexity, particularly if the team adds the capability to dynamically add and remove message brokers based on varying demand. This adds complexity to the logic because the message producers would need to dynamically connect and disconnect as the number of the brokers changes.

Influences

No outside influences affect the use of this pattern, and using this pattern doesn't influence anything external.

How to Govern This Pattern

Architects can both govern this pattern and determine its effectiveness by measuring and tracking *queue depth*, particularly during heavy load times. Periodically measuring the queue depth (the number of pending messages in a queue or topic) provides insights into overall message throughput and scalability. A low queue depth (or none at all) for a single message broker is a good indication that this pattern is not needed to maintain the current message load. However, an increase in queue depth (particularly a significant increase) is a good indication that adopting this pattern would improve message throughput by alleviating a bottleneck in the system.

This measurement is also useful if an architect is using a dynamic allocation algorithm or AI agent to spin up message brokers as demand increases, and tear them down as demand decreases. This pattern provides the flexibility to maintain a *variable* number of message brokers, making it a more effective alternative to streaming brokers such as Kafka, which typically use a fixed number of partitions within a topic.[1]

Related Patterns, Antipatterns, and Pitfalls

The following patterns are related to the Multi-Broker pattern:

Supervisor Consumer pattern (see Chapter 33)

> The Supervisor Consumer pattern programmatically expands and collapses message consumers within a service based on variable message throughput. The same pattern can also be applied to message brokers within the Multi-Broker pattern: the message consumers become message brokers, and the supervisor component becomes an external service that monitors throughput and dynamically adds or removes message brokers as needed.

Thread Delegate pattern (see Chapter 34)

> This pattern allows messages to be processed in order. It can be used in conjunction with the Multi-Broker pattern to preserve message processing order when using multiple consumers. Architects can use *broker affinity* within the message producers, where all messages of the same type, context, or key go to the same broker using an allocation map to direct all messages within a particular context to the same broker, much like the Thread Delegate pattern does.

---

[1] Topics within Kafka can always be *repartitioned* (added or removed), but not dynamically. Repartitioning a topic resets key affinity for the partitions, so using it might require changes in the corresponding message consumers.

Chapter 26. Nexus Architecture Pattern

The Nexus Architecture pattern (or, more accurately, the Nexûs Architecture pattern[1]) describes a way of implementing cross-cutting concerns in distributed architectures such as microservices.

The key to this architecture pattern is finding (and preserving) the cross-cutting coupling point; as we'll show, different implementations of this pattern use different mechanisms.

The metadata for the Nexus Architecture pattern appears in Table 26-1.

Table 26-1. Metadata for the Nexus Architecture pattern

CATEGORY	METADATA
CONTEXT:	Microservices
IMPROVED:	Adaptability, Evolvability, Maintainability, Reusability, Testability
DEGRADED:	Cost, Simplicity

How Do I Manage Cross-Cutting Concerns?

One of the pervasive problems in software architecture is how to handle *cross-cutting concerns*, a classic example of which is cross-service logging. Generally, services in microservices are based on domains and bounded contexts — the logical partitioning of the problem domain. But where does logging fit into domain modeling? It isn't part of the domain — rather, it's

part of the *plumbing* of the system, a capability that is necessary for success but not part of the system's behavior. No architect would design an entire architecture around logging, yet that capability needs to be there.

Teams have long struggled with how best to handle common cross-cutting concerns in architecture. Then the advent of microservices brought us the *service mesh*, which is the foundation for a clever way to handle a whole family of these concerns. Generally, a service mesh is a dedicated infrastructure coupling point via a proxy object in each service to connect to the mesh. The service mesh arose because of a common conflict about domain versus operational coupling. One of the design goals of microservice architectures is a high degree of decoupling, often manifested in the advice that "duplication is preferable to coupling."

Let's say that two services need to pass information about customer addresses back and forth, but the architecture's domain-driven bounded context insists that implementation details should remain private to each service. A common solution is to give each service its own internal representations of entities such as `Address`, then pass that information in loosely coupled ways, such as name-value pairs in JSON. This lets each service change its internal representation at will — including the technology stack — without breaking the integration. Developers generally frown on duplicating code, which can cause issues with synchronization, semantic drift, and more, but some things are worse than duplication…and in microservices, that includes coupling.

Architects who design microservices generally resign themselves to the reality that sometimes they have to duplicate implementations to preserve decoupling. But what about capabilities that *benefit* from high levels of coupling? Each service should have certain common operational capabilities, such as monitoring, logging, authentication and authorization, and circuit breakers. However, when each team manages these dependencies separately, the system often descends into chaos.

For example, consider a company trying to choose one standard monitoring solution for all of its services, to make it easier to operationalize them. The

architect decides to make each team responsible for implementing monitoring for its service: the `Payment` service team, the `Inventory` service team, and so on. But how can the operations team be sure that each team actually does this? Also, what about unified upgrades — if the standardized monitoring tool needs an upgrade across the organization, how should the teams coordinate that?

Service Mesh and Sidecar

Some capabilities in microservices *do* need to be shared, so architects created the Sidecar pattern, a pattern embedded within a service mesh, as illustrated in Figure 26-1.

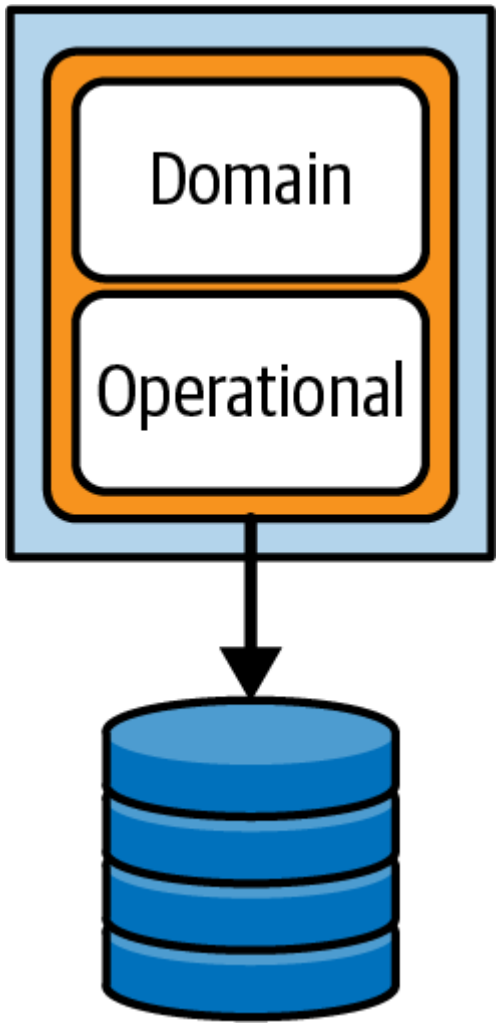


Figure 26-1. The Sidecar pattern to manage operational versus domain coupling

In Figure 26-1, the microservice is split into two primary pieces — one for the domain responsibilities and the other for operational coupling, such as monitoring, logging, service discovery, and other plumbing concerns. This operational component is known as a *sidecar*, inspired by the same-named passenger extension for a motorcycle. The sidecar is the proxy by which all the services in the mesh communicate.

By separating these concerns, the organization can have different teams work on the different pieces (such as the infrastructure and application development teams). This strategy also effectively separates the two concerns by *volatility*. The change rates of these two parts should be quite different, with the operational part (the sidecar) changing much more slowly, while the domain components constantly change (they are, after all, the point of the application). For example, teams don't change their logging strategies on a regular basis.

If architects can assume that all services have a sidecar that implements the same interface, they can build a service mesh, illustrated in Figure 26-2.

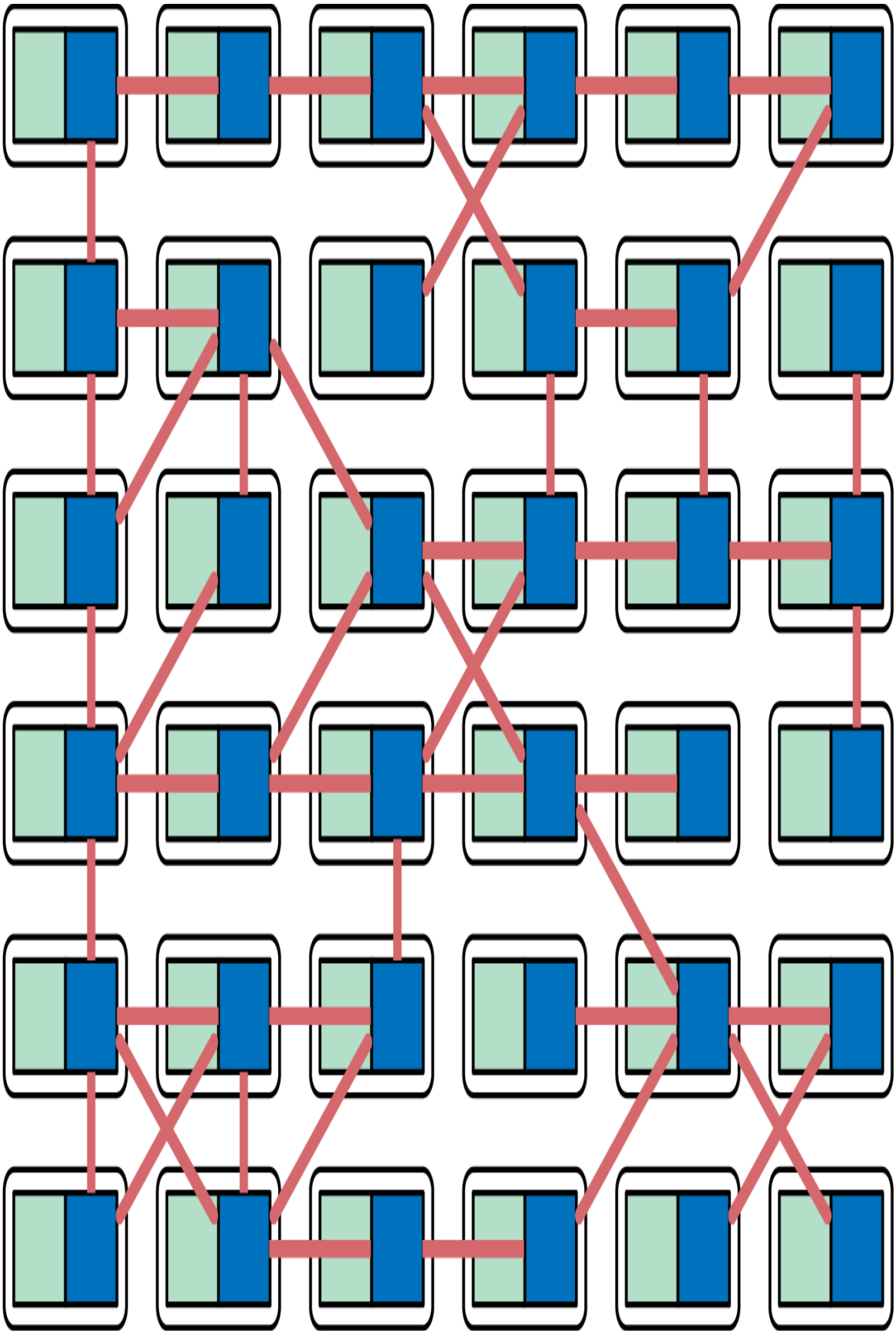


Figure 26-2. A service mesh connects all sidecars and provides an entry point to control operational capabilities

In Figure 26-2, the service mesh connects each of the services via the sidecar interface, which allows teams to control the operational aspects of the system, independent of the domain or behavioral coupling. The service mesh allows teams a single interface to make operational changes without affecting domain coupling. Most service-mesh implementations include a dashboard that allows teams to change monitoring or logging levels, scaling, and other operational concerns via the unified interface provided by the service mesh.

In this architectural pattern, the consistent presence of the sidecar provides a well-defined coupling point, creating an interface for cross-cutting operational concerns that is independent from domain or bounded-context coupling. For example, a set of microservices may communicate asynchronously to provide better responsiveness; this would have no effect on the sidecar communication, which the architect would design consistently for the overall system.

It turns out that operational coupling isn't the only common cross-cutting concern in architecture. However, some approaches are more complex to implement than a simple consistent interface. To handle this, the concept of the service mesh extends to the *data mesh*.

Data Mesh and Cooperative Quantum

The *data mesh* is a concept, created and defined by Zhamak Dehghani while at Thoughtworks, that applies the basic idea of a service mesh to a more complex cross-cutting concern, analytical data.

Teams can separate project *data* into two broad types: operational and analytical. *Operational data* consists of the information that runs the business: catalog items, transactions, sales, and so on. However, many organizations also need *analytical data*, which is derived from operational data and includes things like business forecasting and modeling, market predictions, and aggregations.

Teams have struggled for years with this divide, implementing a variety of architecture solutions like data warehouses[2] and data lakes.[3] Both suffer from problems, mostly stemming from poor topological alignment between the architecture and the data. In both, the goals driving the analytical data architecture differed from those driving the typical application architecture. For example, when architects were building layered architectures and using data warehouses, they stored operational data in a data warehouse using a specialized schema (the *star schema*, which reorganizes data to optimize querying). Teams exerted a tremendous amount of effort on this schema transformation, whether the data was ever queried or not. While the application and the data warehouse were both monolithic systems, their data partitioning was completely different, which led to serious long-term complications.

Most data warehouses were built in conjunction with complex service-oriented architectures, both of which were giant monoliths. However, the software-architecture pendulum eventually swung in the opposite direction, with architects embracing microservice architectures and highly decoupled solutions such as data lakes. In data lakes, operational data is bifurcated: some follows its normal route, but the rest is routed to an unstructured datastore (the "lake"). This philosophy was meant to invert the schema transformation problem of data warehouses: with data lakes, teams only expend effort when data is needed — at which point, they can construct whatever combinations of data they need, on demand.

However, while teams didn't waste effort on preprocessing, data lakes still often duplicated efforts. For example, when building microservice architectures, teams spend time determining the domain service boundaries (which include operational data) and making rules about the data, such as how to handle personally identifiable information. However, when the unstructured data is cast into the data lake, those boundaries and rules are lost — meaning the analytical teams must recreate their work.

The idea of the data mesh resulted from trying to *match* the topologies of analytical and operational data rather than building a translation

mechanism. It was inspired by the concepts of the service mesh and the Sidecar pattern.

However, this pattern is too simple for something as complex as analytical data: while the operational coupling represented in the sidecar has no effect on the system's architectural capabilities, the data does. For example, if an architect tried to implement another sidecar component to handle analytical data, two insurmountable problems would arise.

First, operational and analytical data are not in the same format. Analytical data comes in a variety of formats, depending on how it will be used, including aggregates, composites, averages, and so on.

Second, when an architect designs a microservice that contains operational data, they worry about its scalability and other operational architectural capabilities. Unpredictable queries from an analytical sidecar would make it impossible to build reliable capabilities. For example, consider the case when the domain service is handling a normal number of sales transactions when suddenly the analytics sidecar issues a query on the (shared) database for a complex join, which effectively locks up the database for several minutes, making availability unpredictable.

Thus, in a data mesh, rather than using an embedded sidecar component, architects create a *cooperative quantum*, which is another service with its own datastore (or several, depending on analytical needs) and set of architectural capabilities. The two quanta are *cooperative* because they're tightly coupled but operationally decoupled: they each have their set of architectural capabilities (low static coupling) and only use asynchronous communication (low dynamic coupling). The ideal in the data-mesh world is to build domain-centric teams that include both domain and analytical responsibilities, each separating responsibilities between the two quanta, as illustrated in Figure 26-3.

Alongside the operational service, the architect builds another service to encapsulate analytical data. The code in this service has no domain responsibility; instead, it manipulates the operational data based on the business's analytics needs. The cooperative quantum has a database (or

several, based on need) to store the derived data. For example, analytics may need aggregates of the operational data for business intelligence. The operational datastore and the cooperative quantum have no direct relationship. For example, a particular analytics requirement may dictate that the analytics quantum should contain relational, name-value, and graph databases.

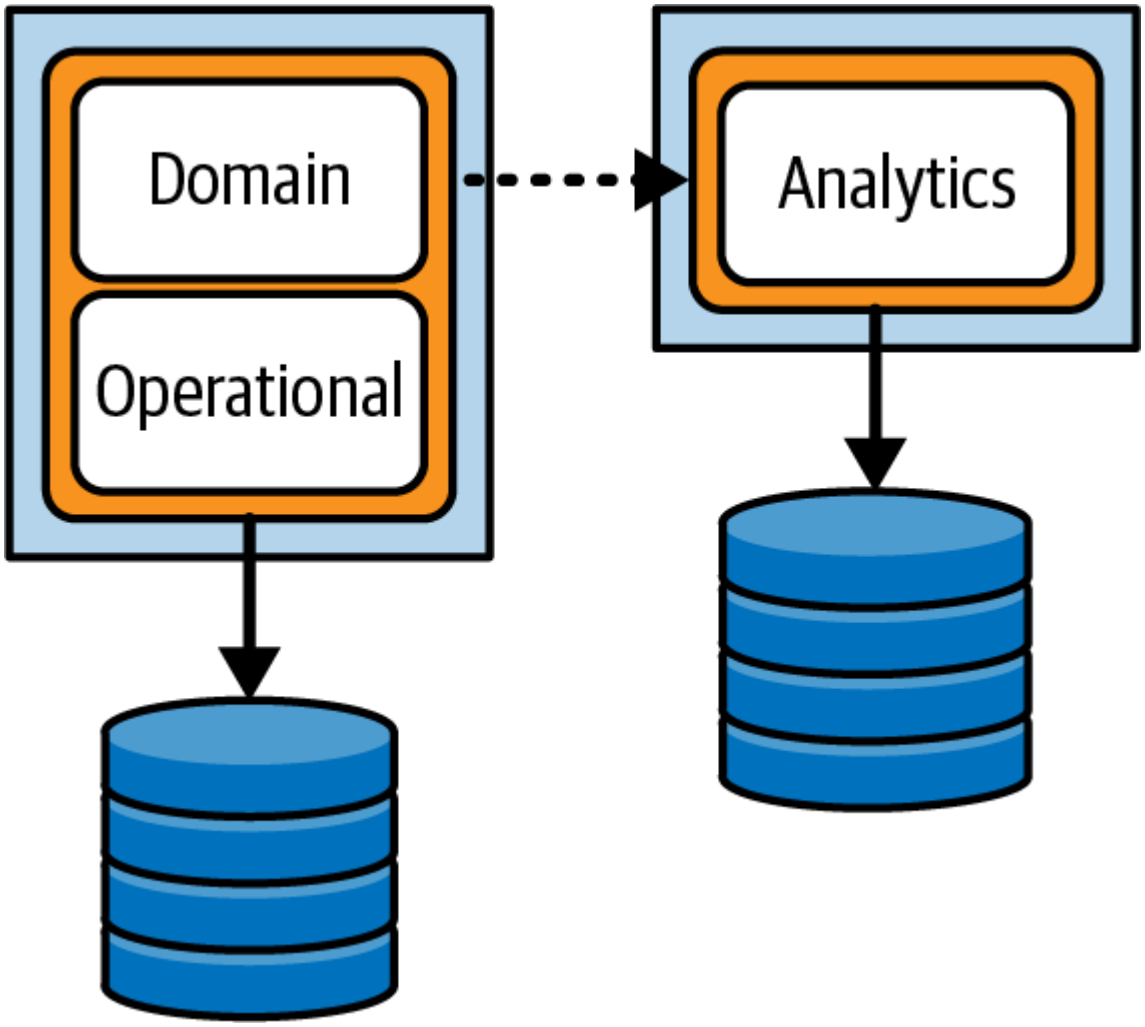


Figure 26-3. The Data Mesh pattern builds a cooperative quantum dedicated to analytical concerns

The join point for the two quanta is an asynchronous connection from the operational service to the analytics service. The *asynchronous* part is important because synchronous communication between the services would dynamically bind the quanta together, creating the same negative side effect on the service's operational architectural capabilities as a sidecar. Only by utilizing asynchronous communication can the architect ensure that the

analytics service won't have a negative effect on the operational service. (See our discussion of architecture quantum in "Architecture Quantum" for more.)

The nicest part of the data-mesh solution is that it doesn't impose a new topology onto the architecture for the sake of analytics, like the data warehouse and data lake patterns do (both well known patterns in the analytical data space, but out of scope for this book). It allows teams to build microservice architectures using domain-driven design and keep those design elements for every part of the architecture, not just the domain parts.

Agentic Mesh, LLMs, and MCP

When Dehghani invented the data mesh, the industry considered it a one-off solution for solving the longstanding problem of separating operational and analytical concerns. However, it points to a larger pattern that was next highlighted by another burgeoning cross-cutting concern: generative artificial intelligence (GenAI). A large language model (LLM) is a global resource for an application, yet each operational (and/or analytical) service may need to use different capabilities of the LLM.

For example, a team designing a chatbot will need additional architectural elements beyond the LLM to make that work:

- One or more retrieval-augmented generation (RAG) databases to handle matches from the LLM
- *Evals*, the GenAI equivalent of fitness functions, used to measure performance and other metrics
- *Guardrails* to manage input, output, and other interactions with the LLM

All of these elements are unique to a particular operational service, as shown in Figure 26-4.

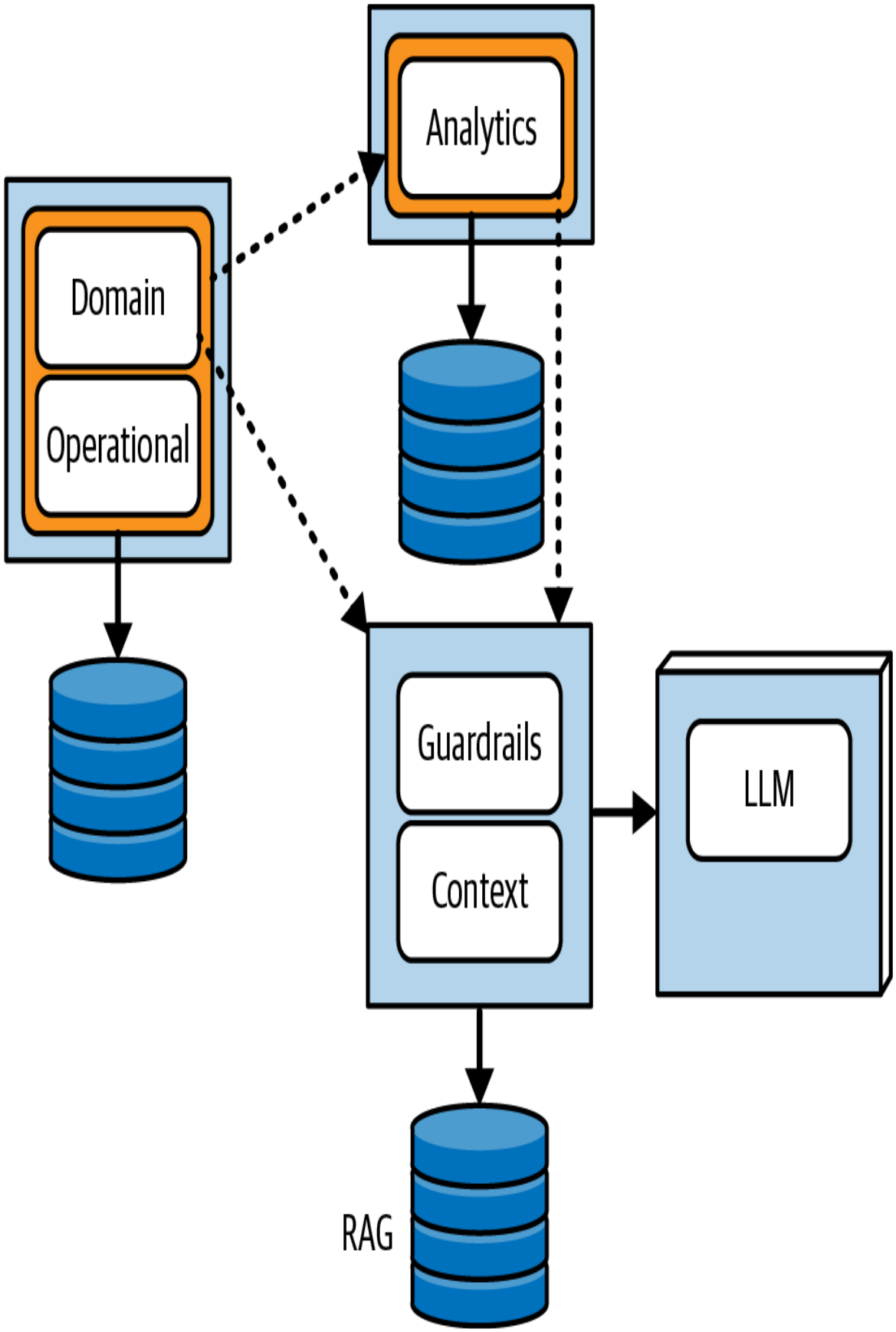

Analytics
Domain
Operational
Guardrails
Context
LLM
RAG

Figure 26-4. Utilizing the same basic topology as data mesh, agentic mesh creates a similar quantum relationship with domain services

In Figure 26-4, the operational and analytical services appear as they do in data mesh, but with another cooperative quantum to handle GenAI tasks.

Just as in a data mesh, communication between the domain service and the agentic quantum should be asynchronous to avoid creating negative side effects for the domain service. The exception would be if the interaction requires some level of synchronicity, in which case the architect will have to consider the impact on the other cooperative quanta.

Nexus Architecture

These examples encourage us to identify a general architecture pattern, which we call the Nexus Architecture pattern, illustrated in Figure 26-5.

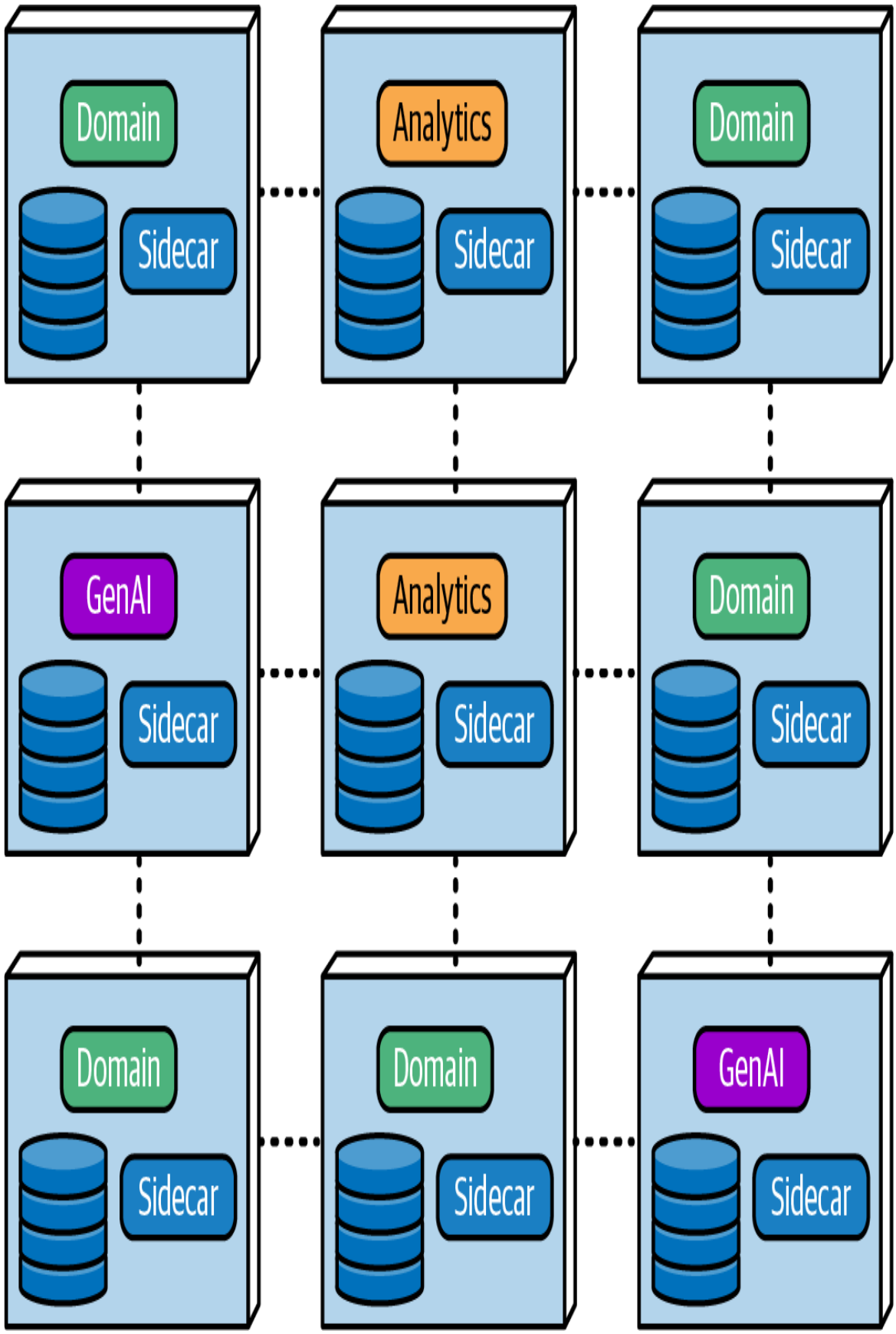

Domain
Sidecar
Analytics
Sidecar
Domain
Sidecar
GenAI
Sidecar
Analytics
Sidecar
Domain
Sidecar
Domain
Sidecar
Domain
Sidecar
GenAI
Sidecar

Figure 26-5. The Nexus architecture pattern defines arbitrary cross-cutting concerns with well-defined join points

In Figure 26-5, we have basically three different architectures overlaid atop one another. The domain services exist to handle day-to-day operations and transactions, the analytical architecture handles business intelligence and forecasting, and the agentic architecture handles interactions and results from GenAI. Each architecture is essentially separate from the others, *except for the well-defined join point* for each, which is the feature of the pattern that allows operational independence.

Aspect-Oriented Architecture?

Experienced developers may feel a sense of déjà vu when reading this section, because it sounds an awful lot like something else from the recent history of computer science: *aspect-oriented programming* (AOP), a metaprogramming technique developed in 2001. AOP was primarily meant to help object-oriented languages handle the common problem of capabilities that cut across the object hierarchy a team was using to model software (thus the common term *cross-cutting concerns*). Consider the simple banking example shown in Example 26-1.

Example 26-1. A simple banking `transfer()` method

```
public void transfer(Account from, Account to, int amount, User
user)  {
    if (!isUserAuthorised(user, from))
        throw new UnauthorisedUserException();

    if (from.getBalance() < amount)
        throw new InsufficientFundsException();

    from.withdraw(amount);
    to.deposit(amount);

    database.commitChanges(); // Atomic operation.
}
```

In Example 26-1, we check to see if we have a valid user and enough money, then make the transfer and commit the changes. This code is simple

and straightforward — but it doesn't reflect the reality of what developers *really* need to write, as shown in Example 26-2.

Example 26-2. The simple `transfer()` method, tarted up with logging calls

```
public void transfer(Account from, Account to, int amount, User
user)  {
    logger.info("Transferring money...");

    if (!isUserAuthorised(user, from)) {
        logger.log(Level.WARNING, "User has no permission.");
        throw new UnauthorisedUserException();
    }

    if (from.getBalance() < amount) {
        logger.log(Level.WARNING, "Insufficient funds.");
        throw new InsufficientFundsException();
    }
     from.withdraw(amount);
    to.deposit(amount);

    database.commitChanges(); // Atomic operation.

    logger.log(Level.INFO, "Transaction successful.");
}
```

In Example 26-2, to make this method work, it also needs to produce log messages each time this code executes reflecting behavior. Because logging has nothing to do with banking, accounts, or anything else in the problem domain, it's a cross-cutting concern.

Consider the code in Example 26-3, which implements an *aspect*, a mechanism that allows programmers to express cross-cutting concerns.

Example 26-3. Defining an aspect that "wraps" specified methods with logging calls

```
aspect Logger {
    Logger logger;

    void Bank.transfer(Account from, Account to, int amount, User
user)  {
        logger.info("Transferring money...");
    }
```

```
    // Other crosscutting code.
}
```

In Example 26-3, we define an aspect named `Logger` which wraps the `Bank.transfer()` method with logging code. This aspect is woven into the code with a special compiler. For example, AspectJ is a compiler for Java that takes existing Java byte code and augments it with a compilation step to add the aspects (this is called *weaving*).

As this example shows, the ability to cleanly define cross-cutting concerns is extremely useful because, as much as developers like hierarchies, not everything fits nicely into them. AOP also coined some useful terminology, which we'll introduce in the next section.

AOP was (and is) heavily used within platforms to provide exactly the kinds of facilities illustrated above: logging, transactions, and other cross-cutting concerns. However, it was profoundly unpopular as a mainstream programming tool, for reasons we uncover in "Why "Nexus Architecture" and Not "Aspect-Oriented Architecture"?".

WHY "NEXUS ARCHITECTURE" AND NOT "ASPECT-ORIENTED ARCHITECTURE"?

We thought about calling this pattern "Aspect-Oriented Architecture," but decided against it for several reasons. First, many developers came to loathe AOP because it was overused in the decade after tools like AspectJ became popular (as metaprogramming often is), with detrimental results. One of AOP's most powerful features is its ability to hide important things that need to happen. In our logging example, we wanted to hide the logging code and get it out of the way of the developer site. However, what happens if the logging code throws an unexpected exception? To the developer, that code isn't there! This inevitably leads to a very frustrating debugging session and some choice words when the hidden behavior is finally uncovered.

One of the best things about aspects is that they're invisible, and one of the worst things about aspects is that they're invisible. Even the developers who created an aspect may forget that it's affecting their code. Fortunately, nexus architectures don't suffer from this problem — this pattern adopts the "cross-cutting" part of aspects, not the hidden compiler. All join points are explicit in the Nexus Architecture pattern.

The second reason we didn't want to use this name, even though it is quite descriptive, is that the industry experienced a significant backlash against AOP. We didn't want to adopt terminology with strongly negative connotations, especially since our pattern doesn't suffer from the problems it connotes.

Applying the Pattern

Despite the complexity of this pattern's context, its implementation is straightforward: in an existing microservices (or other distributed) architecture, create independent services and a well-defined, consistent join

point to allow the necessary communication between the two parts of the architecture.

The key success factor is finding truly cross-cutting concerns. One of AOP's common points of failure was when developers tried to use aspects for concerns that were not purely cross-cutting. The classic use case for aspects is logging — observability is a capability architects want in a distributed architecture that has no effect on domain behavior. Conversely, overusing aspects for behaviors such as atomicity (for example, to define transactions across entities in tools like enterprise service buses) adds considerable complexity. AOP's hidden nature (discussed in "Why "Nexus Architecture" and Not "Aspect-Oriented Architecture"?") aggravated the problem.

To use the Nexus Architecture pattern successfully, architects must find a true orthogonal coupling point. Here are some examples based on uses of this pattern:

Service mesh and sidecar

> In microservices that use a service mesh, the sidecar component is meant to be completely decoupled from domain concerns. Typically, the domain team's view of the operational capabilities exists as an interface they call that acts as a proxy to the sidecar that handles logging, service discovery, monitoring, and so on. The sidecar component implements the interface, hiding the details of how those capabilities happen from the domain team's standpoint. It's considered "cheating on the service mesh" for an architect to add a domain component to the sidecar. For example, for several services that need `Customer` information, a sneaky architect could add a common definition in a component in the service mesh, making it available to everyone. That's the problem: microservices try to *avoid* cross-service coupling of domain concerns, which is what this kind of sharing is.

Data mesh and cooperative quanta

The modularity defined by the concept of an *architecture quantum* (see the definition in "Architecture Quantum") helped inspire the architectural approach of data mesh. A system's architecture quantum defines the scope for a set of architectural capabilities, which is why the sidecar component approach that originated in service mesh wouldn't work for separating domain and analytical concerns for two reasons. First, data access can affect operational architectural capabilities, so architects can't rely on a single database for both concerns. Second, analytical data is derived from, but often not in the same shape as, domain data. For example, analytics needs aggregations, cross-references, and lots of other data shapes.

Thus, in a data mesh, architects build a separate service for analytics with its own set of architectural capabilities and data. The nexus join point for this mesh architecture is the asynchronous data pump from the domain service to the analytical service. This allows the services to segregate analytical responsibilities while keeping domain information semantically coupled.

Agentic mesh and MCP

When domain services need to utilize generative AI via an LLM, they must deal with a global component (the LLM) and a localized one (the localized customizations and operationalization for the LLM). For example, consider an `OrderPlacement` service that allows customers to order products through an LLM-powered chatbot. The architect will want to create localized context with RAG database(s), guardrails, evals, and other localized requirements. Like in a data mesh, the extra structure required for the LLM interaction should reside in a separate service with a well-defined connection to the global resource.

While data mesh was the first to show how to build cross-cutting concerns atop a distributed architecture, it illustrated this general Nexus Architecture pattern. As long as architects can identify and ensure orthogonal coupling or join points, they can build overlapping architectures.

Trade-Offs

The trade-offs for the Nexus Architecture pattern are as follows, along with the comparison chart.

Pros

The primary advantage of this architectural pattern relates to its stated purpose: the ability to cleanly separate concerns in a distributed architecture with the fewest possible negative side effects. When used well, this pattern allows architects to build useful separations in architecture that model team structures. For example, using a service mesh allows a clean separation between the infrastructure and domain teams. Similarly, in a data mesh, physically separating domain and analytical data within the same domain allows for specialized teams within the same context.

Cons

Two problems could negatively affect this pattern, both of which mostly indicate that it's being misused:

Misaligned coupling

> The key advantage of this pattern is the well-defined decoupling of the join point. If the join point has side effects on the domain (or vice versa), the architect must guard against inappropriate coupling "bleeding" across the pattern. A classic example of this is the "cheat" of placing shared domain components in the service mesh — it's convenient, but runs counter to the intent of the pattern.

Hidden magic

> Experienced developers' lingering dislike of AOP often comes down to hidden magic. AOP's programming language support augments the existing language, sometimes leading to surprising results (in a bad way). For example, say the writer of a method doesn't even realize that logging code has been woven around that method. However, what if the

aspect developer didn't take into account that the hard drive holding the logs could become full and throw an exception? As a writer of a method, it's infuriating to see a mysterious bug that claims to reside in the code but was sneakily woven into it at compile time.

Fortunately, because this is an architectural pattern, the behavior it supports isn't hidden by default. For example, in all the mesh examples we discussed, none of the coupling is hidden, just organized in a specific way.

Comparison Chart

The comparison chart for the Nexus Architecture pattern appears in Table 26-2, where the lefthand column, "Without Pattern," refers to a distributed architecture that doesn't use any kind of mesh to handle cross-cutting concerns.

Table 26-2. Comparison chart for the Nexus Architecture pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Adaptability	★★★	★★★★★
Evolvability	★★	★★★★★
Maintainability	★★	★★★★★
Reusability	★★	★★★★★
Testability	★★	★★★★★
DEGRADED		
Cost	★★★★★	★★★
Simplicity	★★★★★	★★★

In Table 26-2, we explain the default trade-offs for this pattern.

Improved

The following capabilities are generally improved when using this pattern:

Adaptability

> Because it separates cross-cutting operational concerns, this pattern makes it easier to add new behavior to the system without worrying about those concerns. However, because adaptability implies leaving

existing behavior intact, this capability becomes more and more difficult over time just because of the sheer amount of behavior in the system.

Evolvability

Similar in this case to adaptability, evolvability describes the ability to fundamentally change core parts of the system over time. Separating operational and other cross-cutting concerns makes it easier for teams to evolve behavior without entangling it with other concerns. This impact is slightly larger than with adaptability because architects are making changes to the underlying system to evolve it — using this pattern keeps the cross-cutting concerns cleanly separated to facilitate that.

Maintainability

Utilizing this pattern helps teams partition different concerns into different well-defined parts of the architecture. For example, if the team needs to upgrade monitoring, they know that all that code resides within the service mesh. One of the benefits of technically partitioned architectures like the layered monolithic architecture lies with the team's knowledge that all technical capabilities reside in one well-known place. This pattern preserves that distinction without also imposing the disadvantage of layered architectures that requires domains to "smear" across the layers.

However, beware the temptation to degrade the separation this pattern depends upon by "cheating" with error handling. For example, let's say an architect designs a choreographed architecture (see the Choreography pattern in Chapter 9) but decides to put error handling in the service mesh. If that error-handling code includes domain logic, suddenly the service mesh has become an orchestrator (see the *Orchestration pattern* in Chapter 27). Now the team is no longer using Choreography but rather Orchestration, with an orchestrator that spans the entire architecture — violating the architecture's foundational philosophy.

Reusability

Many architects like to identify reuse targets based on volatility (discussed in the Overcentralization antipattern in Chapter 28), and it turns out that in many situations, the operational parts of the system (the "plumbing") change at a slower (and more predictable) rate than domain concerns. Thus, using this pattern to separate concerns also helps partition reusability.

Testability

When architects design fitness functions for distributed architectures like microservices, people often ask, "Who runs the fitness function?" For example, if an architect writes a fitness function to check that the referential integrity between two databases stays in sync, that function needs a place from which to execute. The service mesh provides an excellent location to run tests and other verifications. Thus, architects using this pattern automatically have a platform to execute validations and governance tasks.

The orthogonal decoupling of operational concerns has another benefit: teams can test domain behavior without worrying about operational concerns like logging, monitoring, and so on. Clean separation of cross-cutting concerns benefits many engineering practices like testing.

The absence of this pattern doesn't make the actual governance more difficult, but without it, the architect must choose a time and place to execute the fitness function.

Degraded

The following capabilities are degraded when using this pattern:

Cost

Implementing this pattern requires infrastructure and effort to implement, all of which adds cost. However, architects should also

evaluate the cost of implementing all the features found in the mesh individually to see the true comparison.

Simplicity

Building a nexus architecture certainly adds complexity to the underlying architecture, since all of the services must connect to the mesh. While this pattern increases the number of moving parts, when used appropriately, it allows true cross-cutting concerns that match the structure of the overall architecture, without degrading important architectural capabilities.

Influences

This pattern exists at the macro level of a distributed architecture, controlling the top-level partitioning and communication for services and their helpers. This results in a couple of influences over the architecture:

Granularity

Generally, this pattern improves granularity because it provides a well-defined place for cross-cutting concerns to live. For example, if each team creates its own logging solution, those solutions are likely to be wildly inconsistent across the organization. Similarly, if each team implements its own way of isolating domain versus analytical concerns, the architecture will be inconsistent. Using this pattern encourages architects to find appropriate join points for coupling, allowing service granularity to be more consistent.

Architecture quantum

The architecture quantum defines the scope for a set of architectural capabilities, including static and dynamic coupling — both of which influence this pattern. As in previous chapters, we highlight the five different types of coupling defined by architecture quantum and how this pattern affects them:

Static coupling

For static coupling, this pattern creates a well-known dependency on cross-cutting operational functionality. For example, in the service mesh, each service depends on a sidecar component. However, in a data mesh, the data-product cooperative quantum is optional, since some domain services don't need to contribute to analytics. The type of cross-cutting concern dictates dependencies.

Dynamic coupling

The quantum definition also includes the effects of entangling calls with other services. Recall that, based on dynamic quantum coupling (covered in "Architecture Quantum"), synchronous calls between two otherwise decoupled services will temporarily combine their operational architectural capabilities, with the system exhibiting the lesser of the two values. Thus, architects using this pattern should take care that the join point they use to "bind" the cross-cutting concerns doesn't accidentally create large architecture quanta. For example, in a data mesh, the join point is an asynchronous data pump, which avoids synchronous entanglement between the services.

Temporal coupling

For architectures that use caching patterns, the service mesh may be where the cache code executes as a cross-cutting data concern. In these cases, cold-start dependencies may play a part in filling caches with values.

Contract coupling

Generally, architects design contracts to correspond to domain concerns, which lie firmly within the domain part of the architecture, not in a cross-cutting concern. The only impact that a mesh pattern might have on contracts is to provide a location for

teams to execute consumer-driven contract tests to validate service interaction.

Atomic coupling

Atomic coupling also lies firmly within domain concerns. Architects should diligently avoid this most difficult type of coupling from leaking into operational coupling, which will greatly complicate and entangle things that should remain firmly decoupled.

This pattern is mostly expressed via either static coupling (as exemplified in service mesh) or dynamic coupling (exemplified by the asynchronous data pumps commonly used in data mesh).

Architects should pay attention to both service granularity and architecture quantum scope when using this pattern. Choosing poorly for either of these influences can degrade its usefulness and create a disorganized, distributed Big Ball of Mud.

How to Govern This Pattern

Governing the technological coupling and company-wide choices that service meshes encompass falls under the responsibility of enterprise architects in many organizations — they're responsible for choosing platform, tools, and choices that cut across teams.

A common governance strategy in enterprise architecture for ensuring join point presence and execution in microservices is the *distributed governance strategy*, illustrated in Figure 26-6.

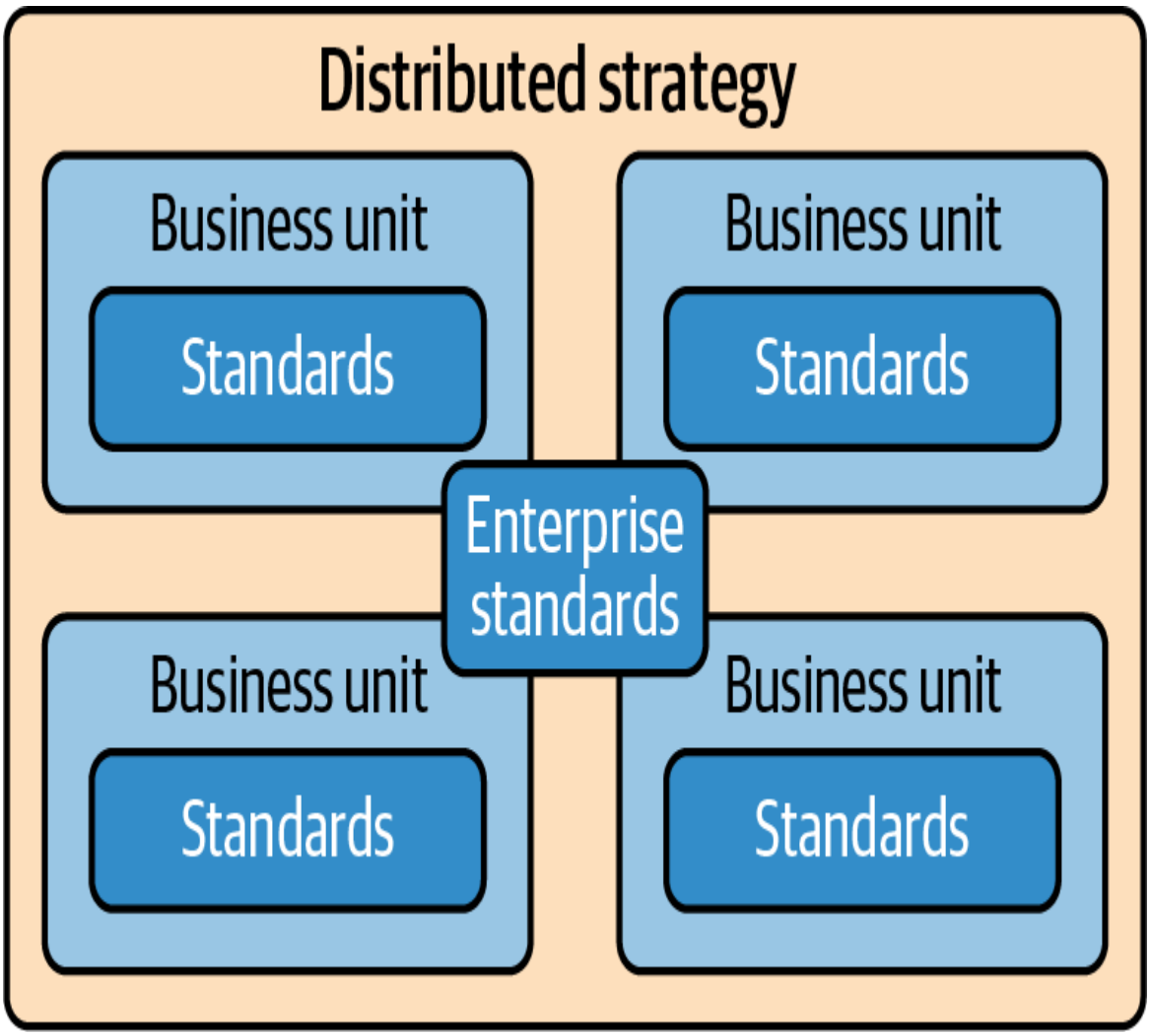


Figure 26-6. The common governance strategy for many microservice architectures

Individual business units choose their own technology stack, frameworks, databases, and so on, while enterprise architects define some core enterprise-wide standards, as illustrated in Figure 26-7.

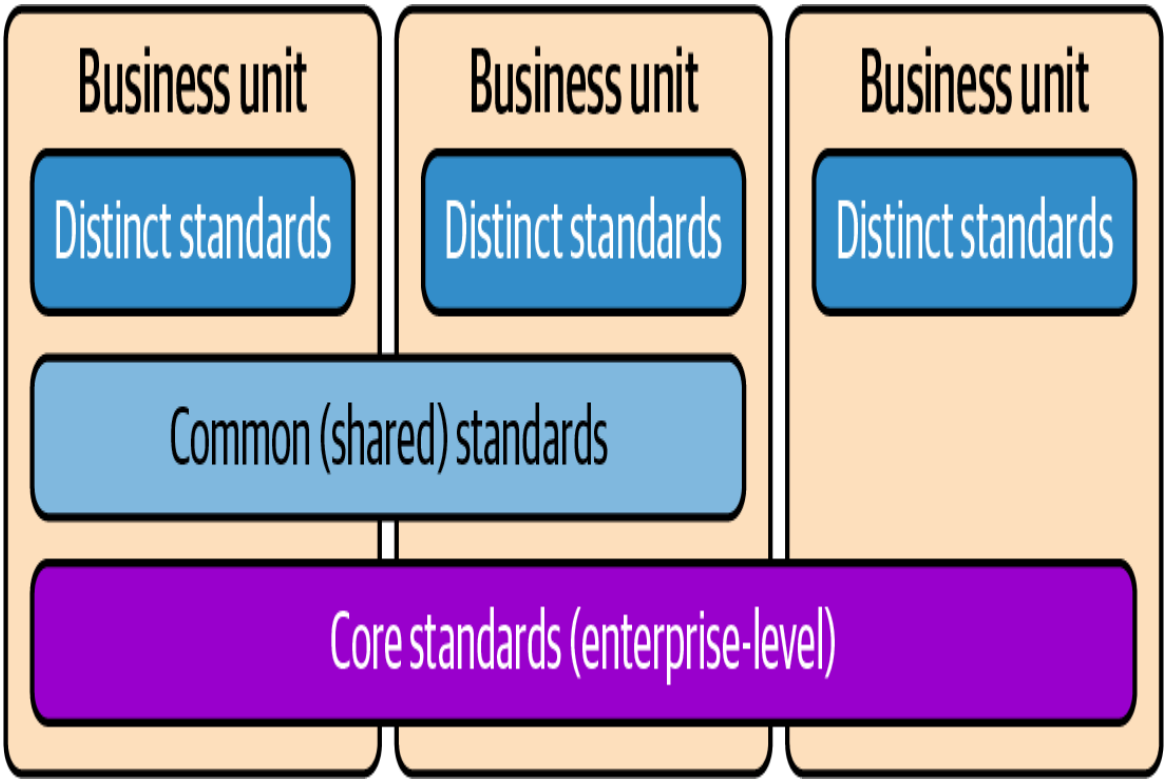


Figure 26-7. Layers of responsibility and governance in the distributed strategy

In Figure 26-7, enterprise architects define the core technology choices, teams share common standards (generally based on technology-stack overlap and synergy), and individual business units can choose how they manage their own technology choices as their needs and preferences dictate. For example, two different strategies seem common in the microservices world for service mesh and sidecar ownership.

In one model, generally used in small organizations with just a few teams, the sidecar component is a shared responsibility among all teams, as illustrated in Figure 26-8.

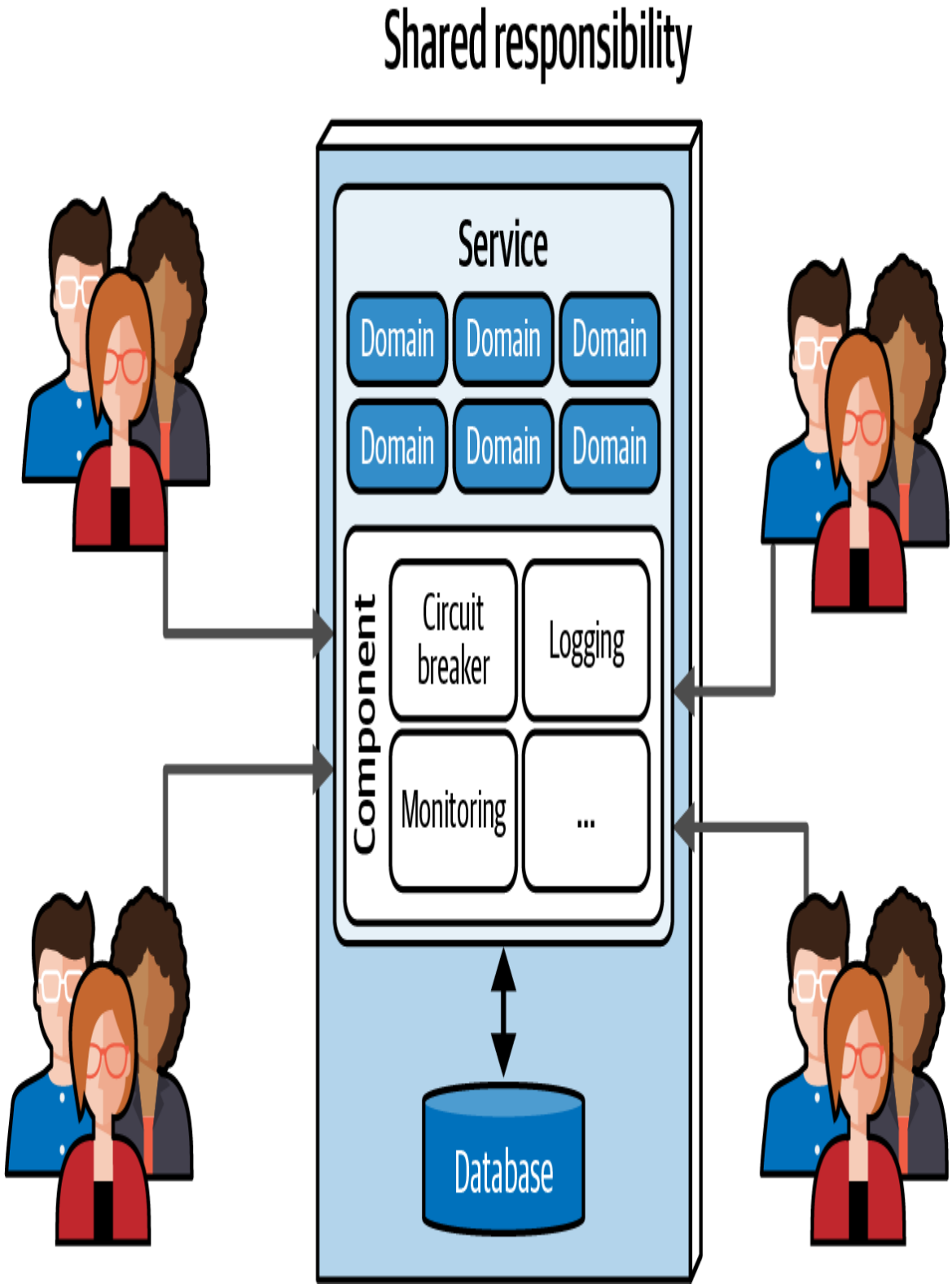


Figure 26-8. Some teams share responsibility for the common sidecar component

In Figure 26-8, everyone owns the sidecar, meaning that everyone must agree on changes and propagate them consistently. However, more teams and/or larger teams make this model more complex because of coordination overhead.

In the second model, used in most larger organizations (certainly any large enough to justify enterprise architects), the more common governance pattern delegates responsibility for the sidecar to the infrastructure team and all other coupling to the domain teams, as shown in Figure 26-9.

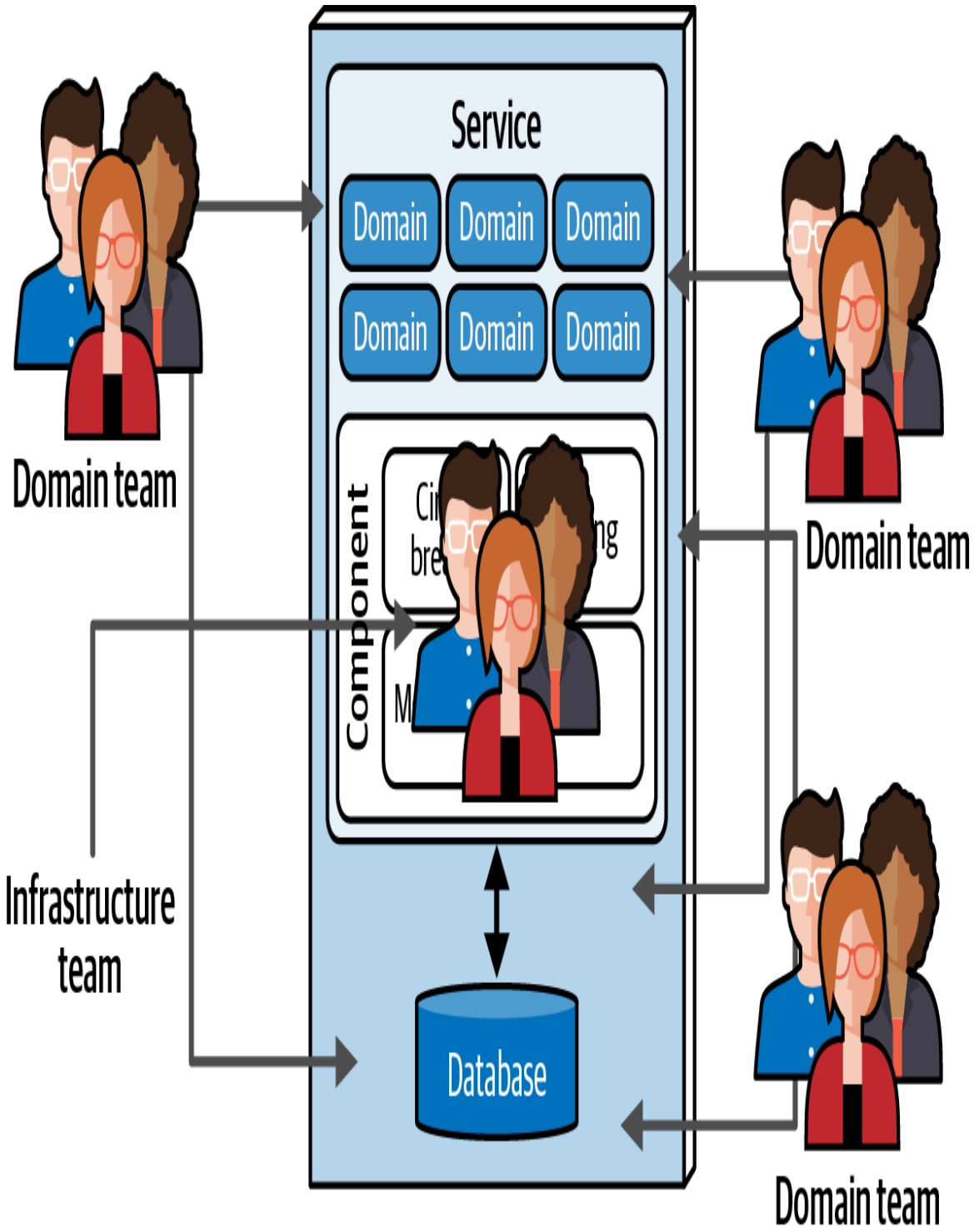


Figure 26-9. This model isolates operational (sidecar) responsibilities to an infrastructure team and leaves other coupling to domain teams.

This separation, illustrated in Figure 26-9, makes perfect sense in the distributed governance enterprise strategy: large organizations *need*

consistency for critical infrastructure components such as monitoring, logging, security, and a host of other important factors.

Related Patterns, Antipatterns, and Pitfalls

This pattern defines cross-cutting architectural reusability and is unique in being the only pattern in this book that's explicitly concerned with reusability. However, it coexists with a number of other patterns that relate to coupling and communication in distributed architectures — see, for example, Chapter 9 (the Choreography pattern), Chapter 27 (the Orchestration pattern), or Chapter 13 (the Cross-cutting State pattern).

---

[1] *Nexus* describes a connection between multiple things: a crossroads. The Latin plural of *nexus* is *nexûs* (pronounced "nex-oos"), which more accurately describes this pattern — it's a combination of more than one nexus, thus the plural of *nexus*. The English plural of *nexus*, *nexuses*, also works, but is less interesting and surprisingly hard to say.

[2] *Data warehouses* are formerly popular enterprise-level tools that massaged operational data into formats that could be applied to business intelligence.

[3] The *data lake* was a popular solution in microservice architectures that separated operational and analytical data into common unstructured datastores, allowing ad hoc data access.

Chapter 27. Orchestration Pattern

Orchestration, metaphorically named for a musical group with a coordinating conductor, is one of two primary communication patterns in distributed architectures such as event-driven and microservices (the other being Choreography, discussed in Chapter 9). Systems that use Orchestration utilize an orchestrator service to coordinate workflow, including state, boundary conditions, and error handling.

Table 27-1 summarizes the metadata for the Orchestration pattern.

Table 27-1. Metadata for the Orchestration pattern

CATEGORY	METADATA
ALSO KNOWN AS:	Mediator
CONTEXT:	Event-Driven Architecture, Microservices, Workflow
IMPROVED:	Data Consistency, Data Integrity, Debuggability, Error Handling
DEGRADED:	Adaptability, Availability, Domain Alignment, Elasticity, Evolvability, Fault Tolerance, Responsiveness, Reusability, Scalability, Throughput

How Should I Manage My Workflows?

The concept of microservices sounds great — small, single-purpose services that are highly decoupled — but quickly runs into impractical realities such as complex workflows and coordination. Often, the level of granularity dictated by the problem domain is different in scope from the granularity of the workflows necessary to implement the desired system behavior.

One common solution to this problem is *orchestration* (in microservices) or *mediation* (in event-driven architecture), illustrated in Figure 27-1.

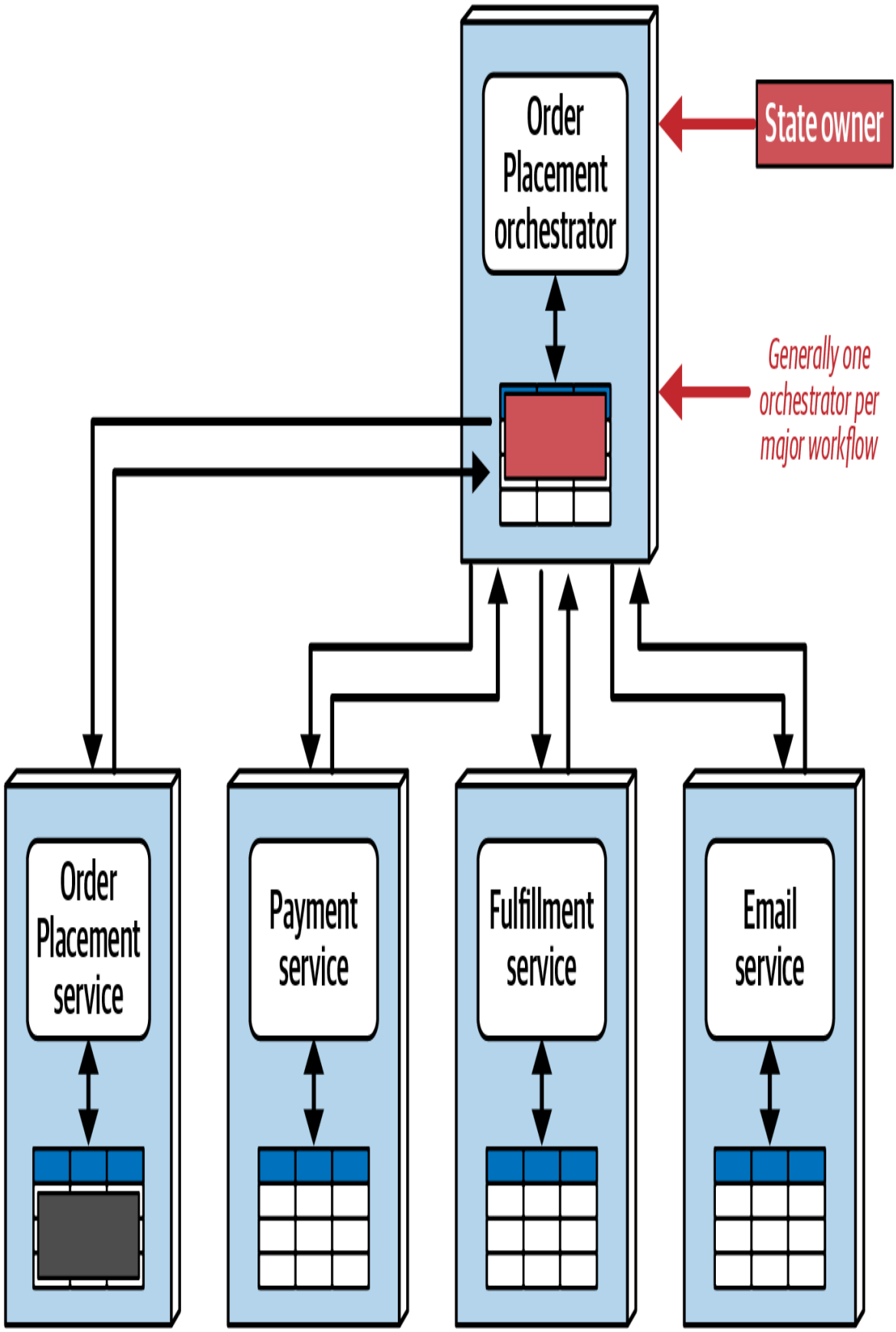

Order Placement orchestrator
State owner
Generally one orchestrator per major workflow
Order Placement service
Payment service
Fulfillment service
Email service

Figure 27-1. The general orchestration pattern in a microservices architecture

In Figure 27-1, the business process at each stage of the transaction involves the domain services `Order Placement`, `Payment`, `Fulfillment`, and `Email`. The orchestrator service coordinates between the domain services. It often contains state, but only for the workflow itself; the domain services will manage state for the problem domain.

Real-world orchestras feature a conductor, who manages timing and other coordination. An individual musician's score may be quite simple: it's the cumulative sound from all of the instruments that makes the orchestras sound cohesive. Similarly, the orchestrator service handles timing as well as boundary cases and error conditions. For example, if an error occurs, the domain service can kick the error back to the orchestrator, which contains logic to handle all possible error scenarios. This is particularly beneficial if some of the domain services participate in more than one workflow. For example, consider the `Payment` service shown in Figure 27-1. It offers a generic service — payment — and thus may be used in multiple workflows that require payment. Because of the orchestrator, the `Payment` service doesn't have to "understand" the context of any errors — it can simply return them to the orchestrator.

In microservice architectures and most event-driven architectures (EDAs), architects use one orchestrator per workflow rather than a single global orchestrator. Some integration architectures do feature a global orchestrator, in the form of an Enterprise Service Bus (ESB). However, that solution is quite uncommon in microservices because it creates tight coupling between all services, which violates one of the primary philosophies underlying microservice architectures and changes the architecture from microservices to a distributed Big Ball of Mud.

Applying the Pattern

This communication pattern coordinates the workflow details for a particular cross-service behavioral outcome. In the example highlighted in

Figure 27-1, the behavior modeled concerns placing an order.

For many event-driven architecture patterns, because they feature so many boundary cases and error possibilities, architects should consider the "happy" path (the workflow with no errors or boundary conditions) and at least one "unhappy" path.

The Case Where Everything Works

When a consumer places an order in the microservices architecture shown in Figure 27-2, the orchestrator intercepts the call and dispatches the details to the relevant domain services.

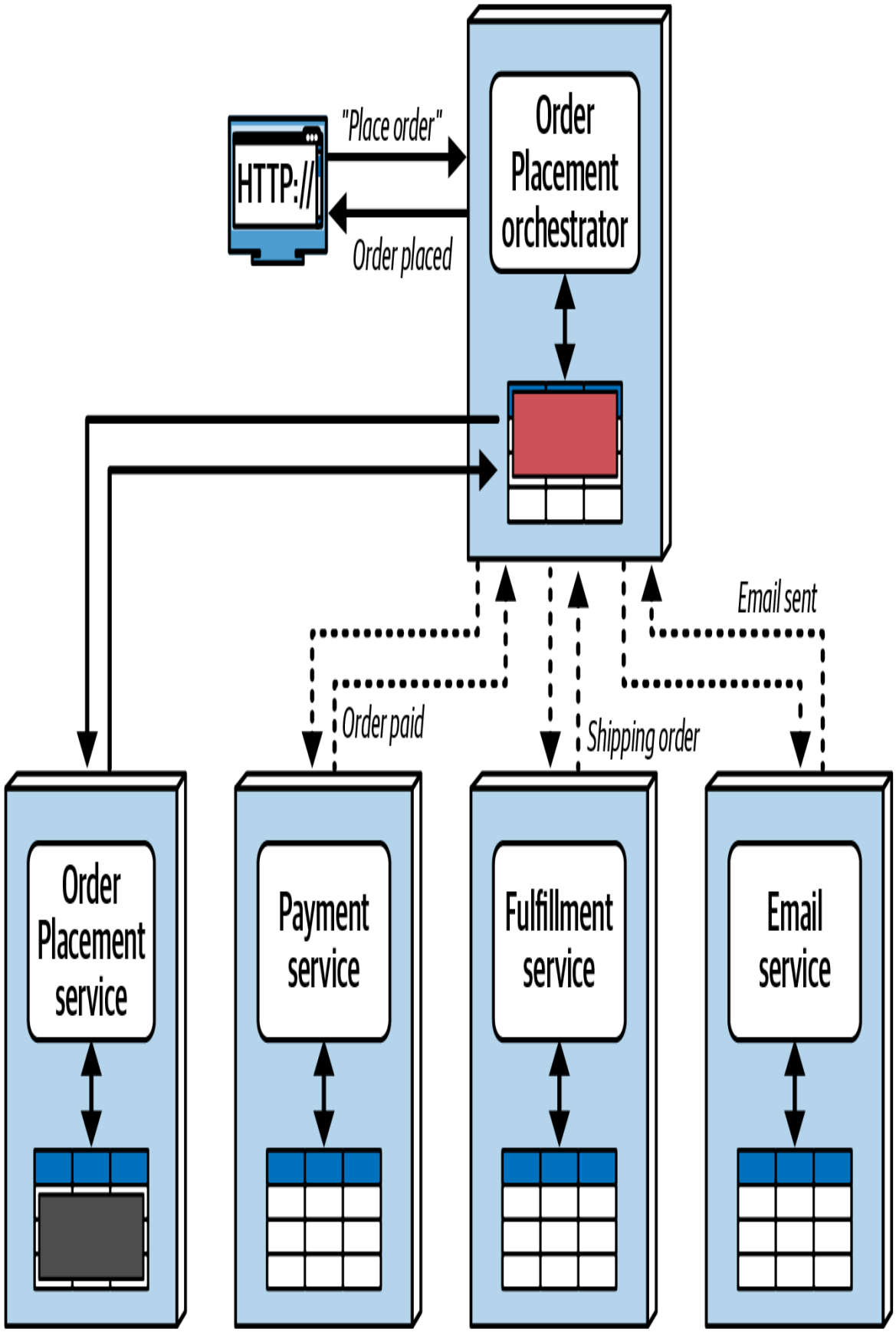

HTTP://
"Place order"
Order placed
Order Placement orchestrator
Email sent
Order paid
Shipping order
Order Placement service
Payment service
Fulfillment service
Email service

Figure 27-2. The no-errors processing workflow

In Figure 27-2, the web client calls the orchestrator, which then dispatches *messages* to each domain service in turn. First, it calls the `OrderPlacement` service; once `OrderPlacement` acknowledges the new order via its return, the orchestrator returns a status to the customer indicating successful completion of the workflow. Next, the orchestrator calls the `PaymentService`, `Fulfillment`, and `Email` services in turn.

Notice that the call to the `OrderPlacement` service is a *synchronous* call (indicated by the solid arrow), whereas the other calls are *asynchronous*. This is an implementation choice, not part of the pattern. In this case, the creator of the workflow wanted to be able to return success when the order is acknowledged and perform the processing steps in the background asynchronously.

Calls from the orchestrator are typically *messages* rather than *events* (see "Events Versus Messages") in both microservice and event-driven architectures. Ultimately, however, this distinction is an implementation decision that depends on specific trade-offs.

The Case Where Something Breaks

The workflow illustrated in Figure 27-2 seems straightforward and linear. However, things don't always work so well. Consider a case where the workflow tries to fulfill the order, but the inventory doesn't contain enough of the product; this error workflow appears in Figure 27-3.

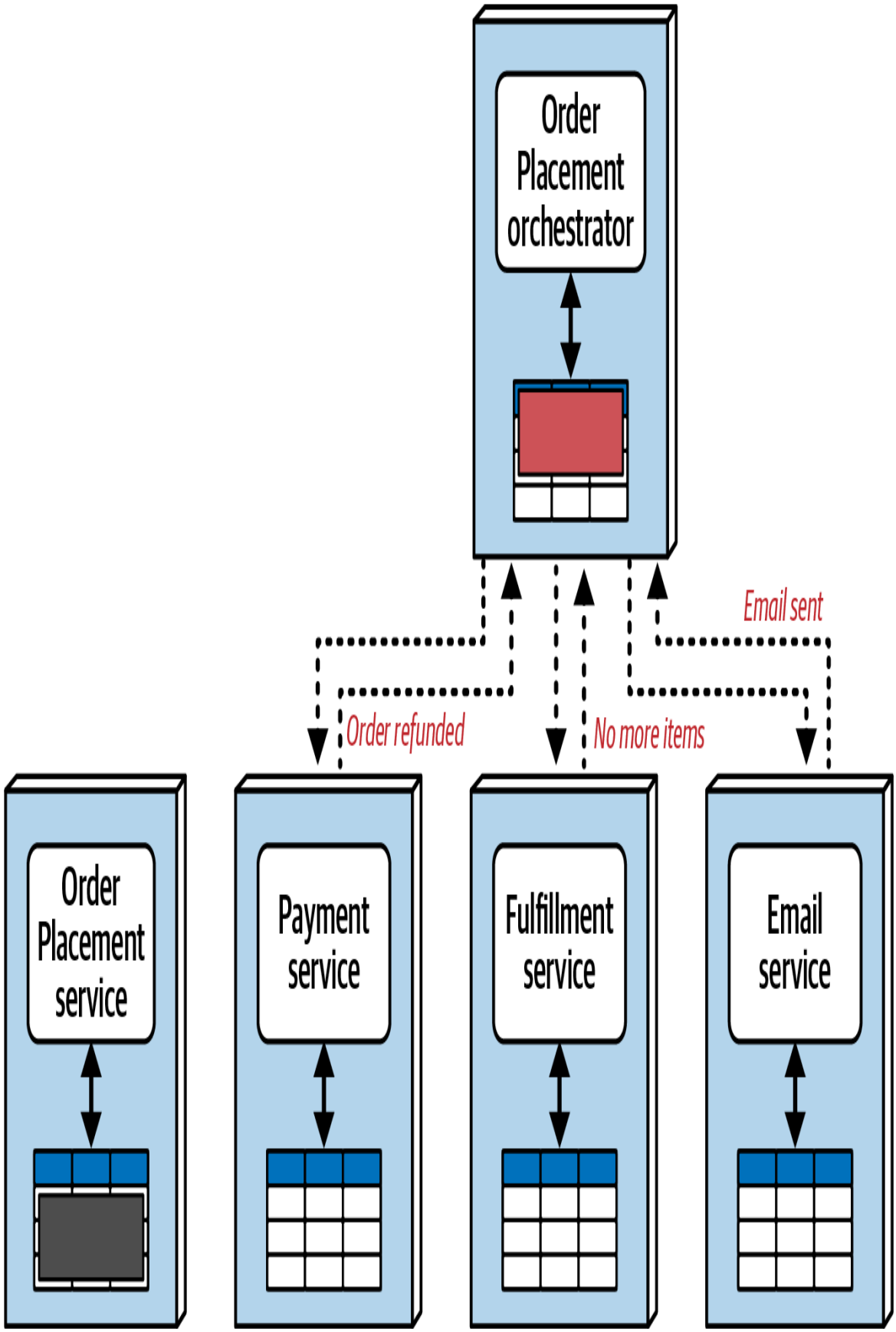

Order Placement orchestrator
Email sent
Order refunded
No more items
Order Placement service
Payment service
Fulfillment service
Email service

Figure 27-3. The error pathways for the Out of Inventory error

In Figure 27-3, the orchestrator tries to fulfill the order but the `FulfillmentService` can't complete the request. One of the orchestrator's benefits is that it provides a default location for all the logic to handle errors, edge cases, boundary conditions, and all other eventualities. Upon receiving the error from the `FulfillmentService`, the orchestrator sends an email to the customer indicating the delay. It also makes a compensating call to the `PaymentService` to undo the already posted payment (which is why most ecommerce sites may choose to charge payments to customers' cards *after* shipping the order — with trade-offs, of course: they don't collect payment as quickly).

Hybrids

In the example in Figure 27-1, the architect has assumed that the *entire* workflow uses Orchestration. However, in the real world, architects often mix and match Orchestration and Choreography (see Chapter 9), as shown in Figure 27-4.

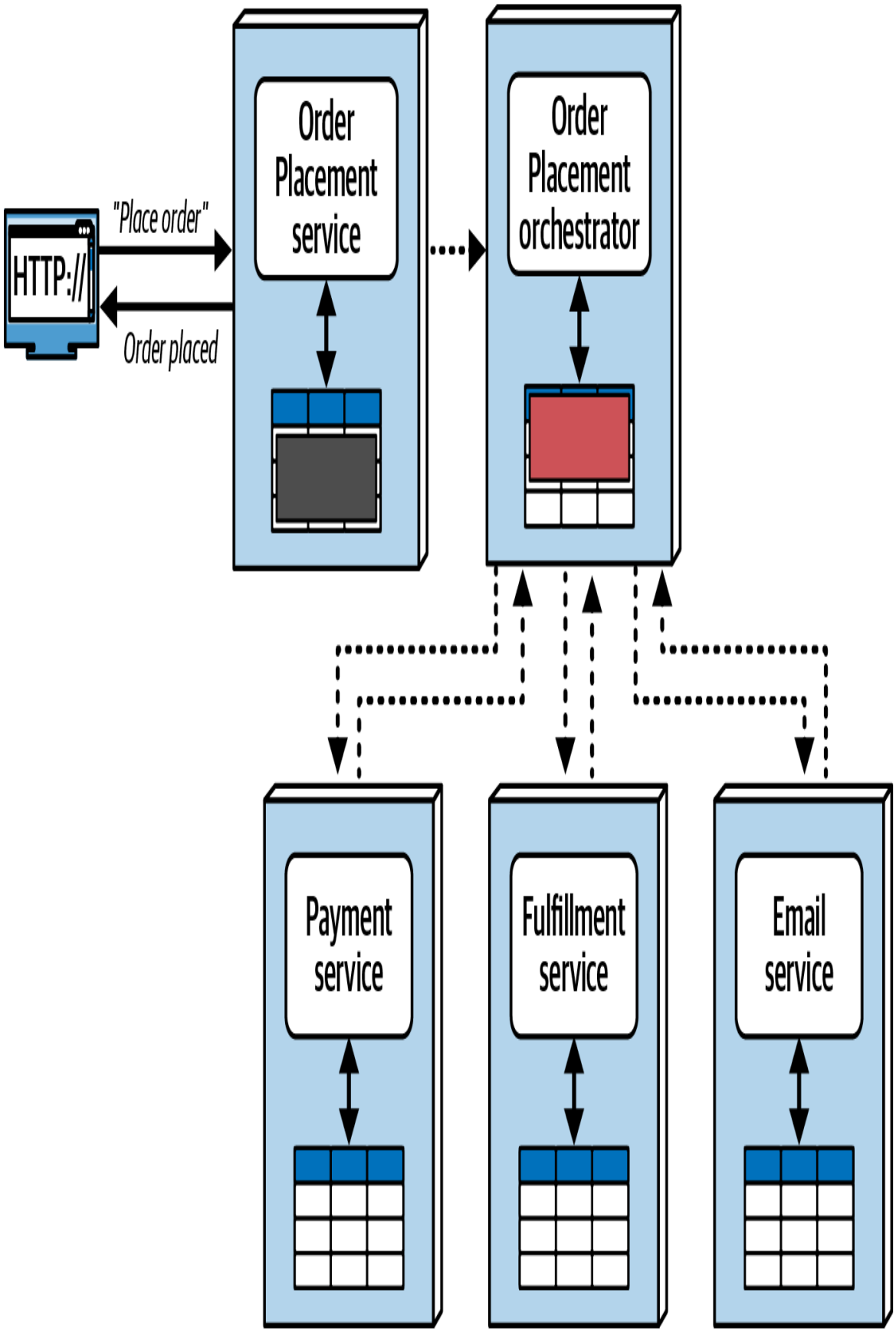

HTTP://
"Place order"
Order placed
Order Placement service
Order Placement orchestrator
Payment service
Fulfillment service
Email service

Figure 27-4. A hybrid book-ordering workflow using both Orchestration and Choreography

In Figure 27-4, the `OrderPlacement` domain service interacts with the customer-facing application directly. Once the order is received, the `OrderPlacement` service dispatches it to the orchestrator that handles all the backend order processing. The advantage of this particular workflow is that it confirms the order faster while dispatching all noncustomer-facing activities to a set of asynchronous services.

Implementing the Pattern

Orchestration is common in all types of distributed architectures, sometimes in the guise of another name (it's typically called the *mediator* pattern in event-driven architectures). Implementing this pattern requires two types of actors: the `Orchestrator` (or `Mediator`) service and one or more domain services. A simple EDA implementation of this pattern that utilizes event processors appears in Example 27-1.

Example 27-1. Simple orchestration happy path implementation

```
define event_processor orchestrator
define event_processor domain_service1
define event_processor domain_service2
define event_processor domain_service3
define system requester
workflow_state = {SUCCESS, ERROR}

orchestrator.add_services(domain_service1, domain_service2,
domain_service3)
state = SUCCESS
on receive message M from requester
   orchestrator.each |service| do
      if (not service.handle(message)) then
         state = ERROR
      end if
   end do
end on receive
return state
```

In this code, the architect defines the relevant event processors. In this case, we've modeled mediation rather than orchestration: event processors rather

than microservices. That simplifies what to do in the `ERROR` condition — simply return the fact that the workflow failed.

All the domain event processors are generic in this case. A `requester` initiates the workflow and the orchestrator enumerates the workflow state (`SUCCESS` or `ERROR`) based on message passing as it iterates through each domain event processor, executing its part of the workflow. If just one event processor invocation returns an error, the entire workflow returns an error.

But what if we were using microservices, where message processing would include a state change? In that case, the architect would apply an additional pattern, Compensating Updates (see Chapter 11).

Trade-Offs

Choosing a type of coordination for a distributed architecture involves many factors.

Pros

The Orchestration pattern is a good choice for architects when the following dominate the decision criteria:

Complex workflows

> In every complex workflow, the coordination logic must reside somewhere, so it's convenient to have a designated place in the architecture for it. In choreographed architectures (see Chapter 9), the workflow logic is spread among the domain services. In an orchestrated workflow, the orchestrator is the ideal place to handle edge cases, boundary conditions, and other workflow logic.
>
> Just as the individual instruments in an orchestra might have quite simple scores, in an orchestrated workflow, the individual domain services require little knowledge of the workflow they're involved in. For example, when an error occurs, the domain service can just return the error condition to the orchestrator and be done with it.

Multiple boundary or error conditions

When a workflow has many different conditions that branch out to other services and/or complex error-handling scenarios, the orchestrator provides a default location for all that logic.

More reusable domain services, less workflow reuse

Relying on the orchestrator to handle exceptional conditions frees the domain services from including that logic, increasing their reusability across a wider set of contexts. However, orchestration also typically utilizes synchronous communication, making the overall workflow less reusable.

Cons

Having an orchestrator service for a workflow in a distributed architecture does have shortcomings:

Increased coupling

As mentioned before, in microservices and other domain-partitioned architectures, the scope of orchestration should be as narrow as possible. For example, an architecture that uses a global orchestrator (such as an Enterprise Service Bus) would no longer be classified as a microservices architecture — coupling everything in the architecture to a single orchestrator would defy one of the key philosophies of microservices.

Even a narrowly scoped orchestrator can create a processing bottleneck, since all messages must go through the orchestrator. Even orchestrators designed for redundancy and discovery still limit throughput. Similarly, orchestrators typically use messages rather than events, which means they require more request/response chatter than do the fire-and-forget events common in choreographed workflows.

Single failure point

When the `Orchestrator` service isn't functioning, no part of the workflow that uses it can function. A choreographed architecture can still partially function when some services are down. The same is true in Orchestration…*except* if what's down is the `Orchestrator` service itself. Thus, any architect using Orchestration to build a highly reliable architecture must take special care to ensure its reliability and availability.

Reduced operational architectural capabilities

Orchestration generally reduces operational architectural capabilities, such as scalability, elasticity, and responsiveness. Because of the presence of the single orchestrator bottleneck, the scalability of the architecture comes down to the scalability of the orchestrator.

Stamp coupling

Using an orchestrator to coordinate complex workflows can sometimes lead to *stamp coupling*, as defined in the Stamp Coupling antipattern (Chapter 31). Workflows tend to grow in complexity over time, with more and more fields added to the contract sent to various services even as the number of services expands. Architects should keep an eye on expanding workflows; the emergence of stamp coupling is an indicator of that because the more collaborators an architect adds to a workflow, the more potential information they pass, leading to stamp coupling.

Architects must evaluate on a case-by-case basis whether the upsides of Orchestration outweigh the negative consequences.

Comparison Chart

When architects build workflows in distributed architecture, they can either coordinate them (some form of Orchestration) or can allow the services to coordinate among themselves (Choreography). Thus, "Without Pattern" with Orchestration implies Choreography, reflected in the chart.

Table 27-2 shows the trade-offs for architects using the Orchestration pattern, contrasting it with its opposite, the Choreography pattern (appearing in Chapter 9).

Table 27-2. Comparison chart for the Orchestration pattern

Capability	Choreography	Orchestration
IMPROVED		
Data Consistency	★★★	★★★★★
Data Integrity	★★★	★★★★★
Debuggability	★★★	★★★★★
Error Handling	★	★★★★★
DEGRADED		
Adaptability	★★★★★	★★
Availability	★★★★★	★★
Domain Alignment	★★★★★	★★★★
Elasticity	★★★★★	★★
Evolvability	★★★★★	★★★★
Fault Tolerance	★★★★★	★★
Responsiveness	★★★★★	★★
Reusability	★★★★★	★★★

Capability	Choreography	Orchestration
Scalability	★★★★★	★★
Throughput	★★★★★	★★

In Table 27-2, the general topology of the pattern creates limitations for features such as responsiveness and makes other capabilities easier. The following sections describe each of the capabilities found in Table 27-2.

Improved

These capabilities are improved when applying this pattern:

Data Consistency

> Data consistency is easier within patterns that have a well-known workflow coordinator such as an orchestrator. Data consistency concerns the synchronization of data, which is easier for architects to ensure with an orchestrator.

Data Integrity

> Data integrity is easier within patterns that have a well-known workflow coordinator such as an orchestrator. Data integrity concerns the correctness of data, which is easier for architects to ensure with an orchestrator, including patterns such as compensating updates (see Chapter 11).

Debuggability

> Architects find it easier to debug architectures that contain well-known workflow components such as an orchestrator. For example, an architect can easily determine the state of the workflow via the orchestrator to help debug inconsistencies.

Error Handling

Architects benefit by having a well-known location for handling errors. This allows workflows to reuse services without having to add specific workflow logic within the service. For example, a `Payment` service used within an orchestrated workflow can rely on it to handle any unusual circumstances, whereas a similar service within a *choreographed* workflow must include more logic (see the Choreography pattern in Chapter 9).

Degraded

These capabilities are degraded when applying this pattern:

Adaptability

Adaptability requires more effort in an orchestrated workflow because the essence of adaptability requires adding new behavior while preserving the existing behavior, requiring the architect to change both domain services and the orchestrator.

Availability

Orchestrated workflows rely first and foremost on the orchestrator; if it doesn't function, nothing in the workflow works. While architects can build redundancy via multiple orchestrators, these workflows are more fragile because of the inherent communication hub the orchestrator implements.

Domain Alignment

Orchestration sometimes adds necessary overhead to a workflow that requires extra error handling, boundary conditions, and other factors. Choreographed workflows can more easily model event flows organically, making it slightly better for domain alignment.

Elasticity

To make an orchestrated workflow more elastic requires making the orchestrator and one or more domain services scalable. Additionally, Orchestration frequently uses messages (rather than events) to and from the orchestrator and domain services, adding communication overhead. Choreographed workflows are more likely to use asynchronous events, eliminating bottlenecks as much as possible.

Evolvability

Orchestrated workflows typically use messages (rather than events) for communication — the orchestrator commands services to perform actions or return answers to queries. While each domain service can be simpler in an orchestrated workflow, the typical workflow communication in this pattern doesn't lend itself to flexible extension. Conversely, choreographed workflows that use asynchronous communication can easily evolve the workflow by simply adding new consumers to existing messages, requiring no other changes.

Fault Tolerance

Orchestrated workflows require an orchestrator, making these workflows more susceptible to faults in the critical orchestrator component. While redundancy and other scalability techniques can help, the reliance of orchestrated workflows on the single component exposes fault vulnerabilities.

Responsiveness

Because these workflows utilize the orchestrator to send and receive messages as part of the workflow, it's inherently a bottleneck. Even if architects build redundant orchestrators (which improves reliability), the fundamental communication within this pattern flows through the orchestrator. Thus, communication overhead limits responsiveness.

Reusability

Like evolvability, reusability generally degrades with Orchestration for two reasons: the increased coupling represented by the orchestrator and the typical use of messages (rather than events) in Orchestration, which hinders reuse.

Scalability

The funnel-like topology of this architecture limits its overall scalability. Architects can improve some operational architectural capabilities by changing to asynchronous communication, addressed in "Communication (Synchronous Versus Asynchronous)".

Throughput

Like scalability, throughput is hindered in this architecture by the single message flow point, the orchestrator. The throughput of the workflow is basically the same as the orchestrator.

Many of the improved capabilities suggest scenarios where Orchestration is the preferred workflow pattern: situations with numerous and varied error and boundary conditions, where teams need workflow state tracking, and can mitigate the degraded operational capabilities.

Influences

A host of other facets of architecture influence workflows and related decisions in distributed architectures. Building a system's behavior often exercises all of its most interesting parts, creating intersections with other influences that shuffle the default trade-offs for this pattern.

Part II of our book, *Software Architecture: The Hard Parts*, deals with the forces an architect must consider when building atomic workflows in a distributed architecture (with or without an orchestrator): coordination, communication, and consistency:

Coordination

Coordination refers to the type of workflow: either Orchestration or Choreography. Clearly, this pattern deals with Orchestration as one of the two options.

Communication

Communication refers to the architect's choice of communication protocol: either synchronous or asynchronous. As discussed in "Architecture Quantum", choosing synchronous communication often changes the extent of coupling in the system, which may in turn affect consistency.

Consistency

Consistency refers to the common choice in microservices to favor *eventual consistency* over *atomicity*. *Atomicity* (as a composite of both data consistency and integrity) implies the same level of transactionality offered by relational databases that feature ACID transactions (see Chapter 37 for a definition). For reasons beyond the scope of this book, distributed architectures can't have full ACID transactions. However, architects can "mimic" the *atomic* part of ACID by designing workflows that feature *compensating updates* (covered in Chapter 11).

Communication (Synchronous Versus Asynchronous)

Table 27-3 shows the trade-off impact of choosing synchronous versus asynchronous communication.

Table 27-3. The trade-off changes based on communication style

Capability	Synchronous	Asynchronous
UNCHANGED		
Domain Alignment	★★★★★	★★★★★
IMPROVED		
Data Consistency	★★★★★	★★★
Data Integrity	★★★★★	★★★
Debuggability	★★★★★	★★★
Error Handling	★★★★★	★
DEGRADED		
Adaptability	★★	★★★★★
Availability	★★	★★★★★
Elasticity	★★	★★★★★
Evolvability	★★★★	★★★★★
Fault Tolerance	★★★	★★★★★

Capability	Synchronous	Asynchronous
Responsiveness	★★	★★★★★
Reusability	★★★	★★★★★
Scalability	★★	★★★★★
Throughput	★★	★★★★★

Choosing a communication style can alter many of the capabilities of a workflow pattern like Orchestration. The following sections describe each of the capabilities found in Table 27-3.

Unchanged

The following capability remains unchanged with regard to this influence:

Domain Alignment

> Domain alignment is largely unaffected by the data consistency strategy the architect chooses.

Improved

Architects find that these capabilities improve when using synchronous communication:

Data Consistency

> Data consistency becomes easier for architects to implement when using synchronous communication — each call blocks and waits, providing a clear success or failure return. Asynchronous communication requires extra effort (and coding) to implement consistency.

Data Integrity

Similar to data consistency, data integrity tries to ensure that data is correct, which is easier with synchronous communication.

Debuggability

Because synchronous communication blocks for each call, call sequence and other useful information for debugging is readily available. When using asynchronous workflows, architects must rely on observability and logging to discern what happened.

Error Handling

Similar to debuggability, error handling is easier when calls block and don't require extra coordination.

Degraded

Conversely, architects find that these capabilities degrade when using synchronous communication (and become better with asynchronous communication):

Adaptability

Using synchronous communication blocks from one call to the next, making it harder for architects to insert additional behavior in existing workflows. In general, workflows that use synchronous communication tend to be more brittle — small changes can rupture the entire workflow.

Availability

When architects rely on synchronous communication, they're at the mercy of availability, particularly for the orchestrator service itself.

Domain Alignment

Orchestrated services that use synchronous communication must model the workflow in addition to domain concerns. Conversely, architects

designing choreographed, asynchronous systems have more flexibility for event flow and domain alignment.

Elasticity

Generally, more decoupled patterns yield better elasticity (Choreography versus Orchestration), and asynchronous communication yields better elasticity. Thus, the more coupled pattern that uses synchronous communication will degrade elasticity.

Evolvability

In general, asynchronous communication makes evolvability easier and thus degrades it when using synchronous communication. Blocking calls and messages (common in this pattern, as opposed to events) makes for more rigid workflows, whereas asynchronous communication and events create more communication flexibility.

Fault Tolerance

Fault tolerance is aggravated when also using synchronous communication, as calls that block have less flexibility to aid recoverability.

Responsiveness

Synchronous communication significantly impacts responsiveness in distributed architectures. When the orchestrator calls each service in turn synchronously, it serializes the workflow. For example, if each of the services in Figure 27-1 takes 1x time to execute, this workflow takes 3x to execute, as opposed to 1x for the same workflow performed asynchronously. This isn't always a negative trade-off — if each service's input must be the output of the previous service, then synchronous communication is the best way to implement it.

One of the most common challenges in microservice architectures is responsiveness: because microservices architecture encourages architects to create fine-grained services with data fidelity, they often

suffer from poor responsiveness because of data isolation. Thus, architects can tweak the performance of microservices that need orchestration for complex workflows by switching to asynchronous communication.

Reusability

In general, asynchronous communication yields better reusability; synchronous communication forces each call to block and wait, making it only reusable in specific situations.

Scalability

Similar to responsiveness, synchronous calls prevent high degrees of scalability because each call blocks while waiting for a result.

Throughput

Blocking calls (using synchronous communication) degrades throughput, whereas operating asynchronously (communicating in parallel) improves it.

Atomicity (Data Consistency and Integrity)

Another common but quite complex intersection with Orchestration is atomicity (the composite of data consistency and integrity), because of the commonality of workflows that need to collaborate with atomic behavior, shown in Table 27-4.

Table 27-4. The trade-offs depending on the level of consistency in the architecture

Capability	Atomic	Eventually Consistent
UNCHANGED		
Domain Alignment	★★★★★	★★★★★
IMPROVED		
Data Consistency	★★★★★	★★
Data Integrity	★★★★★	★★
Debuggability	★★★	★★
DEGRADED		
Adaptability	★★	★★★★★
Availability	★★	★★★★★
Elasticity	★★	★★★★★
Error Handling	★★	★★★★★
Evolvability	★	★★★★★
Fault Tolerance	★	★★★★★

Capability	Atomic	Eventually Consistent
Responsiveness	★★	★★★★★
Reusability	★★★	★★★★★
Scalability	★★	★★★★★
Throughput	★★	★★★★★

The following breakdown shows the impact of atomicity on each of the capabilities found in Table 27-4.

Unchanged

The following capability remains unchanged with regard to this influence:

Domain Alignment

> Domain alignment is largely unaffected by the data consistency strategy the architect chooses.

Improved

Architects tend to find the following capabilities (all involving data) improve when focusing on atomicity:

Data Consistency

> Because the architect chose orchestration plus atomicity, data consistency is fantastic due to this combination.

Data Integrity

> Data integrity is also highly improved in this architecture, although it still requires a bit more coordinating to verify both consistency and integrity.

Debuggability

Debuggability becomes slightly easier when adding atomicity to a distributed workflow because of the finite time completion inherent in atomic workflows. Atomicity requires architects to verify data consistency and integrity by the end of the workflow. Contrast that determinism with eventual consistency, where the proper resolution of the workflow is less deterministic.

Degraded

The capabilities degrade when architects focus on atomicity. The number and severity of these degraded capabilities illustrate data behind the common conventional wisdom in distributed architectures to avoid atomicity whenever possible.

Adaptability

Adaptability is degraded when architects introduce atomicity to orchestration: domain services involved in atomic workflows are harder to adapt because of the atomic coupling.

Availability

Availability is also lessened when architects introduce atomicity because now domain services must rely on more workflow infrastructure, namely the orchestrator, which can become both an availability and responsiveness bottleneck.

Elasticity

Elasticity is degraded because architects must build more coordination logic into domain services to handle atomicity, including compensating updates.

Error Handling

Error handling is degraded because adding atomicity to any workflow adds a notoriously difficult error-handling scenario to the workflow. Orchestration is better in this regard than Choreography in that it offers an automatic place for error handling, but atomicity still complicates it.

Evolvability

It's harder to evolve domain services that have workflow logic around data consistency inside them.

Fault Tolerance

The scope of fault tolerance goes up when architects add atomicity to Orchestration because of the required coordination and availability of the orchestrator.

Responsiveness

Adding atomicity to an orchestrated workflow adds more responsibility to the orchestrator (see the Compensating Updates pattern in Chapter 11), sometimes requiring it to issue extra commands to domain services to restore state. Thus, workflows that also add atomicity will also be less responsive.

Reusability

Reusability is also degraded because adding coordination logic for atomicity to domain services makes them less reusable.

Scalability

Scalability is degraded when combining atomicity and Orchestration because atomicity requires coordination, which adds extra communication overhead to the workflow.

Throughput

Throughput is also degraded when combining atomicity and Orchestration because atomicity requires coordination.

Atomicity, which is a common domain-level request (often without realizing how difficult it is to achieve), provides numerous challenges in distributed architectures and degrades many desirable capabilities.

Architecture Quantum

As defined in "Architecture Quantum", architects must consider five different aspects of coupling and architectural scope as they relate to communication patterns like Orchestration: static, dynamic, temporal, contract, and atomic coupling:

Static coupling

> *Static coupling* refers to the architecture's dependencies and "plumbing." In this case, the orchestrator has a static-coupling dependency on each of the orchestrated services, but the services don't necessarily have a similar dependency on the orchestrator. For example, a `PaymentService` might participate in either an orchestrated or choreographed workflow, depending on the circumstance, thus changing the static coupling. In general, the Orchestration pattern increases static coupling through introducing more dependencies.

Dynamic coupling

> *Dynamic coupling* describes the coupling required when services communicate; this is the essence of the Orchestration pattern. The orchestrator manages the communication between members of the workflow, dynamically coupling them when using synchronous communication.

Temporal coupling

> *Temporal coupling* doesn't really apply in Orchestration, which has no specific time-based dependencies.

Contract coupling

When using Orchestration, the orchestrator maintains the workflow state and coordinates the contracts passed between services. Often, architects need to add extra information to the contracts within the workflow, increasing the coupling between the individual services.

One of the key benefits of Orchestration aptly fits its metaphorical name. In a musical orchestra, when someone looks at the score for an individual instrument, it's often sparse to the point of being incomprehensible. Only when all the instruments perform together does the overall score make sense. Similarly, when using Orchestration, the individual services can be simpler, relying on the orchestrator for extra context.

Atomic coupling

Atomic coupling, the composite of data consistency and integrity, occurs when architects must build pseudotransactional workflows. Orchestration makes this easier because all such workflows must coordinate consistency and integrity, and having a well-defined orchestrator component provides a default location. We discuss the implications of this influence in "Atomicity (Data Consistency and Integrity)".

None of the above aspects are particularly good or bad, but they do introduce important changes to the architecture that architects need to think about.

How to Govern This Pattern

The primary aspects of this pattern for which architects should build governance are communication and coupling. There are several areas to watch out for.

Intraservice Communication

In Chapter 1, we provided an example of a fitness function to govern intraservice communication. Here it is again in Example 27-2.

Example 27-2. Defining a governance rule to control communication between the orchestrator and domain services

```
DEFINE SYSTEM Order Placement as OrderPlacement
DEFINE SERVICE Orchestrator as OrderOrchestrator
DEFINE SERVICE Placement as OrderPlacement
DEFINE SERVICE Payment as Payment
DEFINE SERVICE Inventory as Inventory

FOREACH SERVICE s IN {Placement, Payment, Inventory} DO
    ASSERT(s only communicates with Orchestrator)
END
```

The code in Example 27-2 defines a rule that the domain services must *only* communicate with the orchestrator. Because the orchestrator maintains the state for the workflow, intraservice communication would render that state inaccurate.

Granularity

A common antipattern we call Grains of Sand (see Chapter 21) involves having a workflow that's too fine-grained, which can lead to problems with responsiveness and latency. To govern this possibility, an architect should compare the amount of communication overhead within the architecture to the system's actual behavior, and establish reasonable thresholds.

Related Patterns, Antipatterns, and Pitfalls

Orchestration relates to the following patterns:

Choreography pattern (see Chapter 9)

The other primary communication pattern in distributed architectures.

Compensating Updates pattern (see Chapter 11)

When an orchestrated workflow uses eventual consistency, architects often use the Compensating Updates pattern to implement pseudoatomic behavior.

Chapter 28. Overcentralization Antipattern

Centralizing things is good, right? Sure — until it becomes not so good. Overcentralization is a classic example of an antipattern because it starts with a good motivation — centralizing to facilitate reusability — but sometimes leads to coupling nightmares. Fortunately, architects have a good metric to help avoid the slide from pattern into antipattern.

The overall metadata about the Overcentralization antipattern appears in Table 28-1.

Table 28-1. Metadata for the Overcentralization antipattern

CATEGORY	METADATA
CONTEXT:	Data, Integration Architecture
IMPROVED:	Data Consistency, Data Integrity, Workflow
DEGRADED:	Agility, Availability, Deployability, Domain Alignment, Elasticity, Fault Tolerance, Maintainability, Modularity, Reusability, Scalability, Simplicity, Testability

When Should I Use Global Coupling?

Architects often identify good candidates for reuse within systems, like technical capabilities (such as translating between file formats) or domain abstractions (consolidating everything we know about the `Customer` in one place). Some architects are constantly on the lookout for reusable code.

However, too much consolidation can lead to bad outcomes, for both technical and domain concepts.

Global coupling takes on many forms. Some of the most important are:

Enterprise Service Bus

> A common tool in the service-oriented architecture (SOA) era was the *Enterprise Service Bus* (ESB), a combination of orchestrator and integration hub with plug-ins for a host of additional capabilities. In fact, the driving philosophy behind SOA was to glue tiny building blocks together using the ESB, which offered services such as pseudotransactional coordination (what we refer to as *atomicity* in Chapter 37). This automatically implements much of the Compensating Updates pattern (see Chapter 11).
>
> The problem with the ESB was that it coupled everything in the system to everything else. Changes to even small components rippled throughout the system, potentially affecting every workflow that used that component. Additionally, the strong emphasis on fine granularity meant that workflows spread out across much of the architecture, making it difficult to identify all the constituent parts.

Master data management

> Large companies have complex data relationships, and it seems like a good idea to consolidate data in a central location. For example, in a large integration architecture, several different workflows need access to `Customer` information, so architects might create a central location for that data. However, this causes two problems. First, because so many systems use the shared data, every time the schema or some other aspect changes, all those coupled systems must also change. Second, because this datastore must store everything the organization knows about the entity (`Customer` in this case), it becomes complex. When a team needs a subset of the data, they must deal with the complexity of the entire thing.

A good bit of advice from Domain-Driven Design (DDD) is to *avoid* creating global data structures for these two reasons. Its advice is to keep data structures localized instead, keeping them as simple as possible, and deal with integration through communication rather than consolidation.

Broadly scoped (accidental) orchestration in microservices

A common bit of advice for architects building microservices that need orchestration is to keep the scope of the orchestrator as narrow as possible. This is a reaction to the problems with the ESB, which included a global orchestrator. However, architects sometimes inadvertently implement overly broad coupling by orchestration. For example, a common error-handling solution in Choreography (see the Choreography pattern in Chapter 9) relies on an error-handling service. The choreographed services are highly decoupled…*except* for the error handler. This is a narrow but common use of Orchestration to address one of the problems that tend to occur in Choreography. However, once the architect has created an error handler for a particular workflow, they might be tempted to expand that to other choreographed workflows that need error handling. They have some common use cases, after all. All the architect is really doing here is slipping into too much coupling, creating rippling side effects. Like localized data in the DDD case, error handling should be narrowly scoped in most distributed architectures.

Why Is This an Antipattern?

This antipattern occurs because of a classic misinterpretation of our First Law of Software Architecture: everything in software architecture is a trade-off. *Reusability* is a classic example of this law. Effective reusability has *two* facets, and everyone gets the first one correct: *abstraction*. That is the purpose of reusability: to find some piece to abstract for use in multiple contexts. However, many architects miss the second important facet: low volatility. *Volatility* refers to how often the code changes (also known as "churn"). If a team has chosen a reuse target because of a found abstraction

but the target has high volatility, it will cause overall churn in the system. Every time the volatile target changes, the side effects will "ripple out" to other parts of the architecture.

In fact, when architects ignore abstraction and focus on volatility, they see that the most effective reusability comes from *technical* abstraction rather than *domain* abstraction. Technical abstraction includes tools, frameworks, libraries, and anything else in a project that requires no domain knowledge. Teams are accustomed to reusing these types of artifacts, most of which have well-understood versioning and revision strategies.

Domain concepts, on the other hand, involve the application's behavior: the problem the team is writing software to solve. Because the problem domain is the focus of development, it tends to change the quickest. Thus, when trying to achieve domain-level reusability, architects should focus on measuring volatility, which is covered next.

How to Avoid This Antipattern

Fortunately, architects can avoid this antipattern by paying attention to both abstraction and volatility. Incidentally, this split is why most effective reuse involves technical artifacts like framework and libraries: they change less frequently than domain concepts. Domain concepts represent the motivation for writing the software, so in most projects they change frequently.

In fact, *volatility-based decomposition* is an architecture methodology based around volatility as a design element. It analyzes volatility as a first-class design concern, segregating behavior and capabilities based on their volatility.

Architects don't have to embrace an entire framework to take advantage of volatility as a design criterion. In our book, *Software Architecture: The Hard Parts*, we introduce tools that help architects iterate on the granularity of their microservices design, which we split into *disintegrators* (tools that increase granularity) and *integrators* (tools that bundle services back together). One of our disintegrators is volatility; we leverage this

metacapability of code to help determine granularity. As observed above, targets that exhibit good abstraction and low volatility make good reuse targets.

Trade-Offs

Like all antipatterns, the goal that leads to this one seems perfectly sensible: keep all the *X* in one place. However, architects know that reusability requires both abstraction and low volatility (which many potential reuse targets fail).

Pros

Consolidating disparate information seems like a good idea, and from a purely reusability and single representation standpoint, it is. For the purposes of data consistency, data integrity, orchestration workflow reusability, analytics, and hosts of other examples, single source of truth is extremely important and is often insisted upon by business stakeholders without realizing the host of negative trade-offs involved.

Cons

Everything in software architecture is a trade-off, including reusability via consolidation. As we stated above, architects must take multiple factors into account, including all five types of coupling (static, dynamic, contract, temporal, and atomic) and volatility.

Comparison Chart

The many capabilities impacted by this antipattern are listed in Table 28-2.

Table 28-2. Comparison chart for the Overcentralization antipattern

Capability	Using Antipattern	Avoiding Antipattern
IMPROVED		
Data Consistency	★★★★★	★★★
Data Integrity	★★★★★	★★★
Workflow	★★★★★	★★★★
DEGRADED		
Agility	★★	★★★★★
Availability	★★	★★★★★
Deployability	★★	★★★★★
Domain Alignment	★	★★★★★
Elasticity	★	★★★★★
Fault Tolerance	★★	★★★★★
Maintainability	★★★	★★★★★
Modularity	★	★★★★★
Reusability	★★	★★★★★

Capability	Using Antipattern	Avoiding Antipattern
Scalability	★★	★★★★★
Simplicity	★★	★★★★★
Testability	★★★★	★★★★★

This antipattern impacts a number of capabilities, listed as follows.

Improved

The following capabilities improve when using this antipattern:

Data Consistency

Data consistency is generally improved with this antipattern, and many architects must assess the cost of this advantage in relation to the many negative aspects. Data centralization is valued (whether that is quantified or not) so highly in many organizations that they will endure extraordinary pain and inconvenience to achieve it.

Data Integrity

Like data consistency, data integrity is one of the few capabilities that this antipattern improves, and is often the motivation for utilizing it.

Workflow

While an architect's ability to build workflows is impacted when something centralized is involved, it doesn't change much: workflows require coordination, and the negative aspect this antipattern brings to that coordination has a broader scope and is potentially more brittle.

Degraded

The following capabilities are degraded when using this antipattern:

Agility

Agility is crushed with this antipattern and gets worse the more it's applied. Agility is a composite that covers deployability, testability, and a host of other factors, all degraded by having too much centralization.

Availability

When architects build centralized resources, that resource must always be available for the architecture to be successful. The more centralized something is, the more coupling, volatility, and a host of other problems negatively affect availability. Consequently, architectures that rely on this antipattern (such as SOA in the early 2000s, which relied on a centralized ESB) can use clustering and federation to protect the availability vulnerability of the single source of failure.

Deployability

The more teams increase coupling to centralized things, the harder it is for them to be agile in terms of deployability. For example, if a system uses a centralized master datastore, they must coordinate any changes to it with each deployment, adding friction and coupling to the deployment process.

Domain Alignment

DDD encourages architects to isolate domain concerns together, but if part of that concern is centralized, it creates coupling points and breaks bounded contexts.

Elasticity

Operational capabilities like elasticity suffer greatly when architects choose to create global coupling points, and the degree of damage correlates to both the centrality of the coupling and how operationally critical it is. For example, in the case of master datastores, a single data

coupling point creates update problems as well as potential operational problems like elasticity.

Fault Tolerance

Fault tolerance is degraded because teams must worry about the availability of every centralized coupling point and how it will affect overall fault tolerance. Like availability, many architects that "lightly" apply this antipattern can use clustering, federation, and other DevOps techniques to mitigate the risk of single points of failure.

Maintainability

Systems that spread coupling across a wide area tend toward degraded maintainability. For example, when using a centralized data store, each time the central store changes, all coupling points that use it must iterate to check to see if they're affected.

Modularity

Like domain alignment, this antipattern harms modularity because it encourages architects to break natural modular boundaries, like bounded context, and instead create artificial ones, such as consolidated data.

Reusability

Although reusability is the very motivation for centralizing, it's ultimately degraded by this antipattern. Systems that rely on centralized behavior or data (such as master data or global orchestration via an ESB) struggle with effective reusability.

This is ironic because one of the primary architectural styles of the early 2000s was SOA, which encouraged global orchestration via an ESB. The goal of the architecture was enterprise reusability and integration ease, so architects designed fine-grained components to represent business entities and then global orchestration logic to "stitch" them together. However, they quickly discovered that, from a practical standpoint, it's virtually impossible to create the correct level of

granularity for proper reuse, and architects could not make issues like transactions and other forms of coupling generic enough to handle globally. The failure of this kind of global reusability led to microservices.

Scalability

Like elasticity, any centralized coupling point makes a distributed architecture more difficult to scale because the central coupling point must scale equally to the rest of the system.

Simplicity

Creating centralized coupling points complicates all the parts of the system that touch them, driving up the overall complexity of the system.

Testability

Testability is degraded with this antipattern because teams must test a broader scope that includes what's centralized.

Influences

This antipattern is not influenced by any outside factors, save for business coworkers who see the benefits but don't yet understand the trade-offs.

Am I in This Antipattern?

This antipattern creates a gradual slippery slope toward its negative impact. As mentioned above, the goal of centralizing information is laudable and makes sense…purely from a logical partitioning standpoint. However, software architecture exists in time, not as a static snapshot, and the physical ramifications of too much centralization can introduce friction.

The symptoms of that friction are measurable: anytime a team uses a centralized resource of any kind, they should measure both how truly

reusable it is (without variations, which often lead to versioning explosions) and how volatile it is.

Volatility is one of the key measures for architects to determine whether they've fallen into this antipattern. For example, for centralized data, an architect should track how often their schemas change and how much of a rippling side effect that has. For infrastructure coupling points like ESBs, architects should look at the volatility of the workflow orchestration code kept at the global level, and how much and often it changes.

Architects should set a limit on how much they'll allow some centralized resource or tool to impact other parts of the architecture, and start building interfaces to act as anticorruption layers between broadly shared entities.

Related Patterns, Antipatterns, and Pitfalls

The following patterns are related to the Overcentralization antipattern:

Nexus Architecture pattern (see Chapter 26)

> The Nexus Architecture pattern is the disciplined counterpart to this antipattern. Both centralize cross-cutting concerns, but Nexus does so only at well-defined join points and favors low-volatility, technical concerns — exactly the discipline that overcentralization lacks. When centralization is governed this way, it becomes a pattern rather than an antipattern.

Compensating Updates pattern (see Chapter 11)

> ESBs, a common form of overcentralization, historically bundled pseudotransactional coordination that automated much of the Compensating Updates pattern. Architects who decompose a centralized bus into more loosely coupled services lose that built-in behavior and must implement compensating updates themselves.

Choreography pattern (see Chapter 9)

Overcentralization frequently creeps into choreographed architectures through a shared error handler that gradually expands into a de facto global orchestrator. Keeping error handling narrowly scoped preserves the decoupling that makes Choreography attractive in the first place.

Orchestration pattern (see Chapter 27)

Broadly scoped Orchestration is one of the most common on-ramps to this antipattern. The remedy is not to abandon Orchestration but to keep each orchestrator's scope as narrow as the workflow it coordinates requires.

Chapter 29. Read-Through/Write-Through Cache Pattern

Architects often use caching within their architectures to improve responsiveness. However, introducing a cache creates a consistency challenge: the cache can easily become stale or inconsistent with the datastore. Traditionally, this burden falls on the application layer, which must know what gets cached, for how long, when to invalidate entries, and how to maintain consistency, as exemplified by the Cache Aside pattern (Chapter 8). The Read-Through/Write-Through Cache pattern takes a different approach: it delegates this synchronization complexity entirely to the cache layer, simplifying application logic at the cost of reduced flexibility.

Table 29-1 summarizes the metadata for the Read-Through/Write-Through Cache pattern.

Table 29-1. Metadata for the Read-Through/Write-Through Cache pattern

CATEGORY	METADATA
CONTEXT:	Caching, Data, Microservices, Monolithic Architectures
IMPROVED:	Data Consistency, Simplicity
DEGRADED:	Flexibility

When Using Caching, How Do I Optimize Data Consistency?

The Read-Through/Write-Through Cache pattern is a popular caching pattern that's useful in both monolithic and distributed architectures. It's particularly appropriate for read-heavy workloads where consistency is far more important than write throughput.

Applying the Pattern

The Read-Through/Write-Through Cache pattern shifts the responsibility of updating the primary data source from the application to the cache. This means that the cache acts as an intermediary between the application and the datastore. That is, as far as the application is concerned, the *cache* is its primary data source. In this pattern, the cache is in the critical path, sitting *between* the application and the data source. Contrast this to some other caching patterns, like the Cache Aside pattern in Chapter 8, where the cache sits *to the side* and isn't in the critical path. Some cache implementations use a "loader" mechanism that allows the cache to read from the datastore and use the datastore as a persistence mechanism on writes. Some cache implementations natively support durable datastores as backends, while others require adapters. These adapters, which the application team implements, typically include a *loader* to populate cache entries from the datastore on cache misses, and a *writer* (or "store" component) to persist changes to the datastore when the cache is written to or updated.

Implementing the Pattern

Consider the sequence of events that occur for both reads and writes (or updates), one at a time.

Reads

Sequentially, the read side of the Read-Through/Write-Through Cache pattern is easy to understand. Figure 29-1 illustrates the happy path, where the target item exists in the cache.

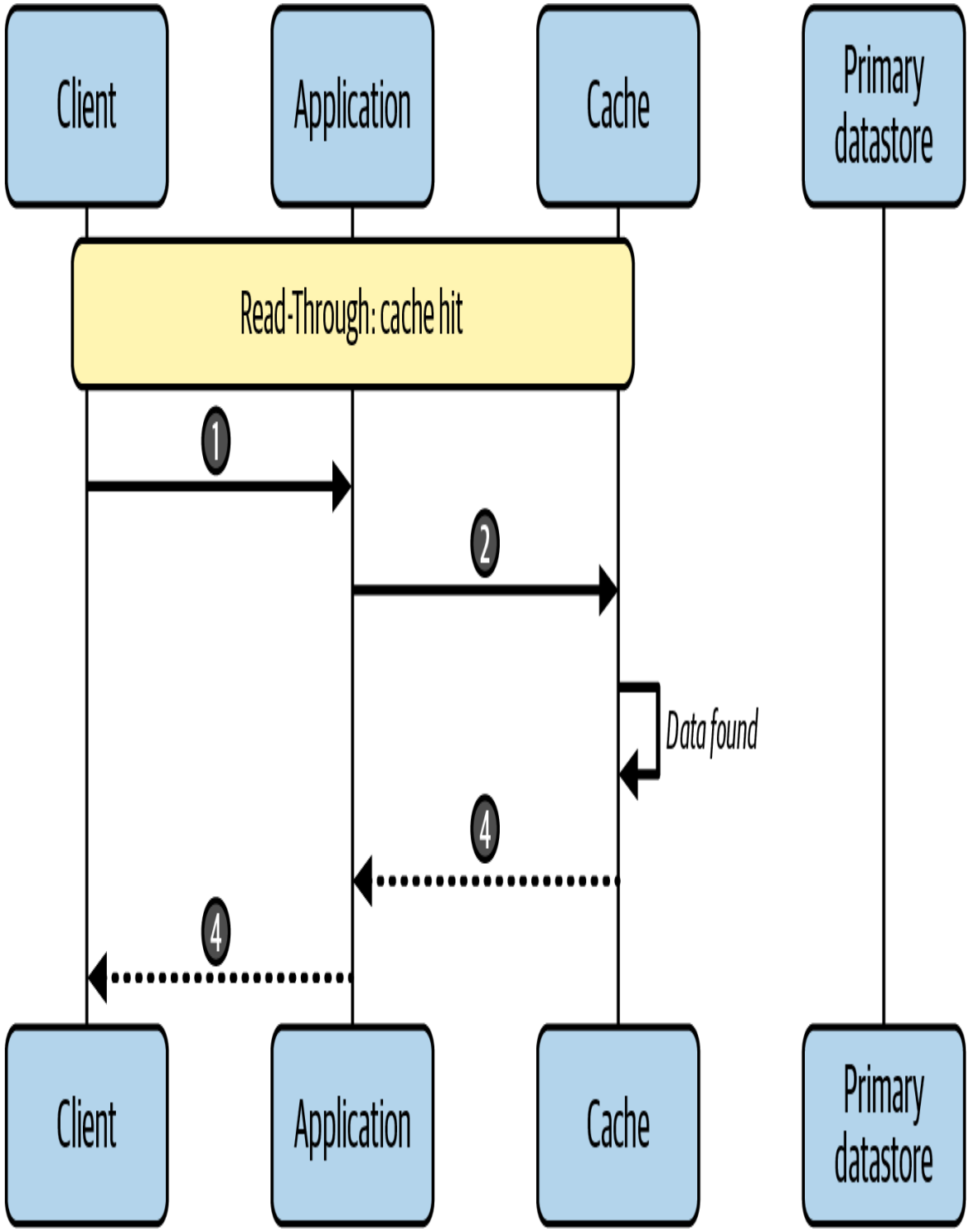


Figure 29-1. Read-Through/Write-Through Cache pattern read sequence diagram for a cache hit

As depicted in Figure 29-1, the sequence of events is as follows:

1. The client requests data from the application.
2. The application checks the cache for the requested data.
3. The cache looks up the requested key and returns it to the application.
4. The application returns the requested data to the client.

Figure 29-2 illustrates what happens on a cache miss.

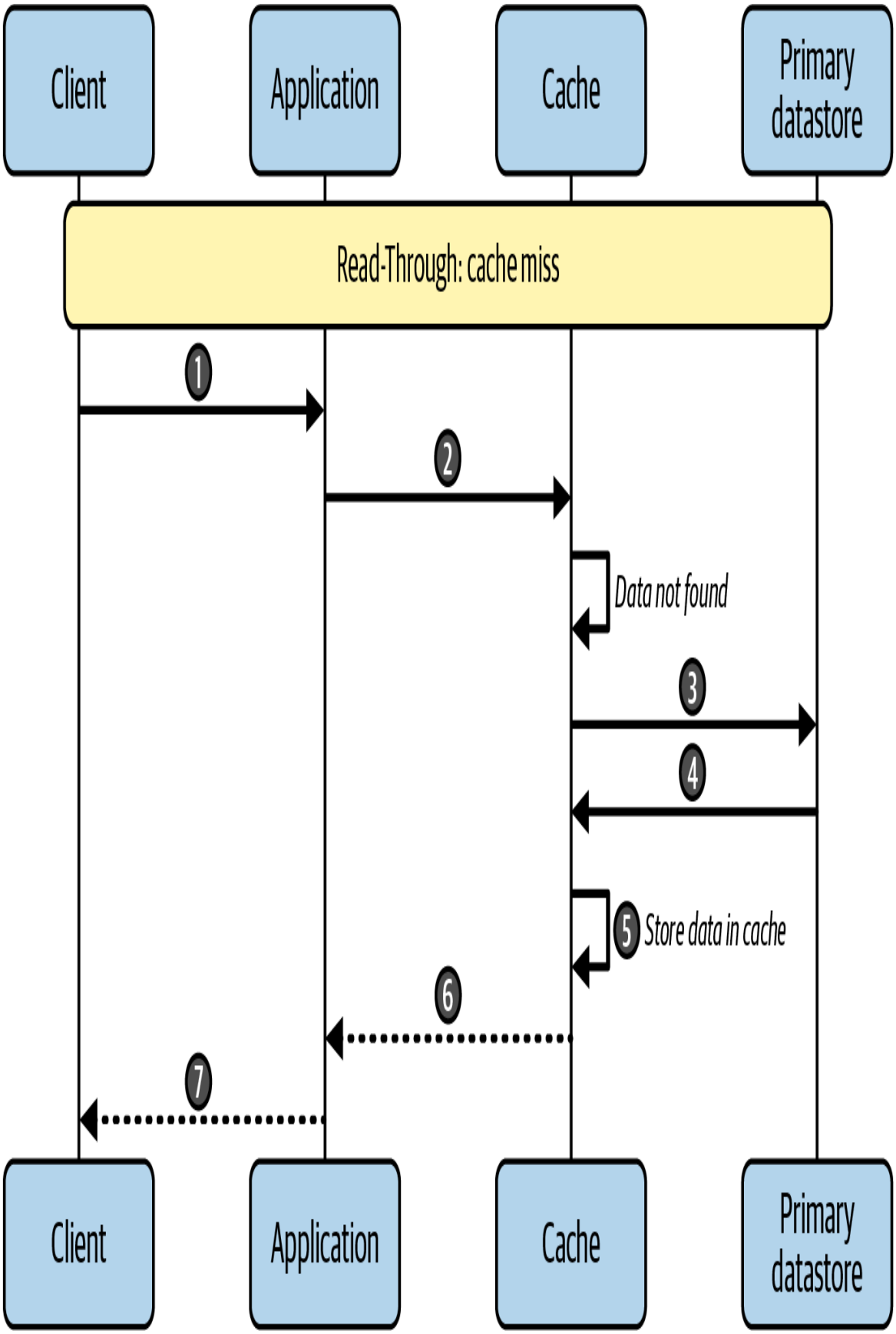

Client
Application
Cache
Primary datastore
Read-Through: cache miss
1
2
Data not found
3
4
5 Store data in cache
6
7
Client
Application
Cache
Primary datastore

Figure 29-2. Read-Through/Write-Through Cache pattern read-sequence diagram for a cache miss

Elaborating on Figure 29-2:

1. The client requests data from the application.
2. The application checks the cache for the requested data.
3. The data is *not* found in the cache, so the cache blocks while it attempts to look up the data in the primary datastore. It might use a preconfigured loader that was provided to the cache upon startup.
4. If the requested item is found in the datastore, the datastore returns it to the cache.
5. The cache stores a copy of the retrieved data in the cache for future requests.
6. The cache returns the data to the application.
7. The application returns the requested data to the client.

Several things are worth highlighting here. The Read-Through/Write-Through Cache pattern starts off with an empty cache, which is only populated on a cache miss. In this regard, its default behavior is to lazy-load items into the cache, very much like the Cache Aside pattern in Chapter 8. What differs in this pattern is the party responsible for populating the cache on a cache miss. In the Read-Through/Write-Through Cache pattern, it's the cache's job to reach for the primary datastore, retrieve the missing data, cache it for future requests, and then respond to the application request. As depicted in Figure 29-2, all this happens *synchronously* and in sequence.

Writes/Updates

When the application receives new or updated data, it simply writes the data to the cache, offloading the responsibility of data persistence to the cache.

Let's start with the happy path, where the write to the primary datastore succeeds, as illustrated in Figure 29-3.

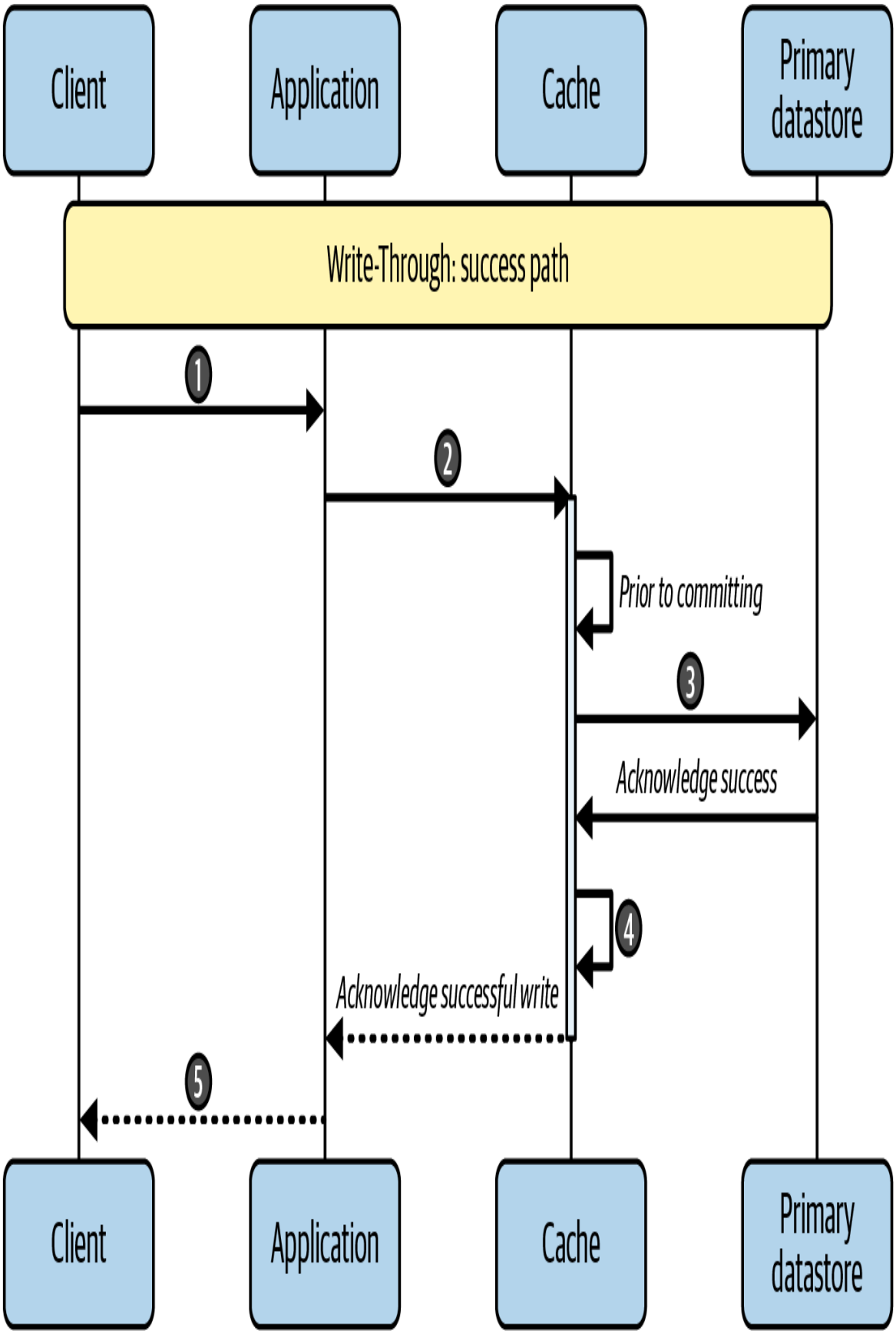

Client
Application
Cache
Primary datastore
Write-Through: success path
1
2
Prior to committing
3
Acknowledge success
4
Acknowledge successful write
5
Client
Application
Cache
Primary datastore

Figure 29-3. Read-Through/Write-Through Cache pattern write/update success path sequence diagram

The sequence of events is as follows:

1. The client sends new or updated data to the application.
2. The application writes the new data or updates existing data to the cache.
3. The cache, *before* committing, *synchronously* writes the data to the primary datastore using a writer or store component.
4. The cache commits its changes and acknowledges a successful write to the application.
5. The application acknowledges to the client that the update was successful (for example, with a `200 OK` HTTP status code).

The write to the primary datastore doesn't always succeed. A write may fail for a variety of reasons: network or infrastructure failures, constraint violations, transactional failures, or maybe even a validation check failure.

Figure 29-4 illustrates what happens when a write to the primary datastore fails.

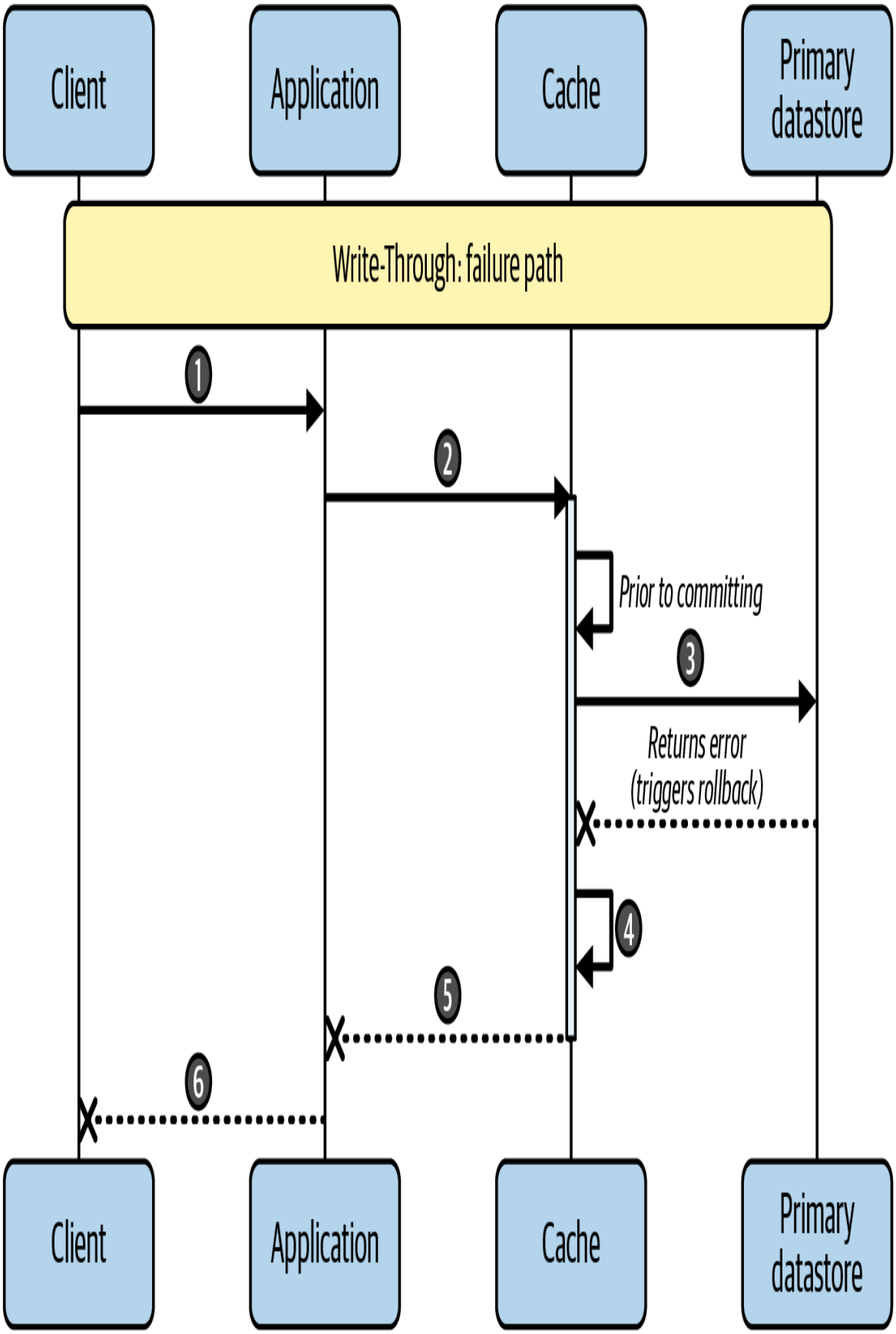

Client
Application
Cache
Primary datastore
Write-Through: failure path
1
2
Prior to committing
3
Returns error
(triggers rollback)
4
5
6
Client
Application
Cache
Primary datastore

Figure 29-4. Read-Through/Write-Through Cache pattern write/update failure path sequence diagram

The steps are:

1. The client sends new or updated data to the application.
2. The application writes the new data or updates existing data to the cache.
3. The cache synchronously writes the data to the primary datastore using a writer or store component.
4. If the write or update to the primary datastore fails, the cache does not commit its changes; instead, it rolls back to the previous value.
5. The cache returns an error code to the application.
6. The application reports the error (perhaps as a `5xx` error) to the client.

Things of Note

Now that we've examined the sequence of events for both reads and writes, it's worth examining a few unique aspects of the Read-Through/Write-Through Cache pattern.

Loader and store implementation

Some caches don't natively support a persistent backend and will need a loader and datastore implementation. This implementation will need logic to read and write to the primary data source, most likely using a query language like SQL. It might also need logic upon reading to convert data into the format the cache expects. The cache will probably also need proper error handling, in case writes need to be retried. Failures should propagate back to the cache, and transactional support will be needed to ensure that writes to the cache itself are rolled back correctly. The key detail here is that *all of these operations need to happen synchronously*.

Cache in the critical path

A defining feature of the Read-Through/Write-Through Cache pattern is that the cache sits in the critical path between the application and primary data source. In other words, from the application's perspective, the cache *is*

the datastore. The application team provides a loader and store implementation, but doesn't need to implement additional logic for the application to interact with both the cache and the primary datastore.

Although the application treats the cache as its primary datastore, it's impossible to have a scenario where the cache has been updated but the datastore has not. That is, the primary data source is *always* the source of truth. It's easy to verify this implementation detail using an integration test, as shown in Example 29-1.

Example 29-1. Integration test to verify that the cache is in the critical path

```
TEST "Ensure cache is rolled back if primary datastore write/update
fails"
  // Instantiate
  CREATE dataStoreMock
  CREATE readWriteThroughCache with dataStoreMock injected
  CREATE serviceInstance with readWriteThroughCache injected

  // Setup: Prime cache with initial value
  dataStoreMock.saveData("testKey", "initialValue") RETURNS SUCCESS
  serviceInstance.writeData("testKey", "initialValue")

  // Verify initial state
  ASSERT readWriteThroughCache.get("testKey") EQUALS "initialValue"
  ASSERT dataStoreMock.get("testKey") EQUALS "initialValue"

  // Configure mock to fail on next write
  THROW (new DatabaseException())
      WHEN dataStoreMock.saveData("testKey", "updatedValue")

  // Attempt write that will fail at datastore
  RESULT = serviceInstance.writeData("testKey", "updatedValue")

  // Assert: write fails, dataStore is not updated, cache is not
updated
  ASSERT RESULT IS FAILURE
  ASSERT dataStoreMock.saveData WAS CALLED WITH "testKey" and
"updatedValue"
  // Cache rolled back
  ASSERT readWriteThroughCache.get("testKey") EQUALS "initialValue"
  // Datastore unchanged
  ASSERT dataStoreMock.get("testKey") EQUALS "initialValue"
```

In the purest implementation of this pattern, the application *cannot* circumvent the cache. For example, if the cache were to become temporarily unavailable, the application *could not* access the primary datastore directly. This too can be verified using an integration test, as shown in Example 29-2.

Example 29-2. Integration test to verify that the cache cannot be bypassed

```
TEST "Application cannot bypass cache when cache is unavailable"
  // Instantiate
  CREATE dataStoreMock
  CREATE readWriteThroughCache (unavailable)
  CREATE serviceInstance with readWriteThroughCache injected

  // Setup: Configure cache to be unavailable
  THROW (new CacheUnavailableException("cache down"))
      WHEN readWriteThroughCache.get("missKey")
  THROW (new CacheUnavailableException("cache down"))
      WHEN readWriteThroughCache.put("anyKey", "anyValue")

  // Setup: Ensure datastore exists but MUST NOT be touched by the
application
  dataStoreMock.get("missKey") RETURNS "should-not-be-accessed"
  dataStoreMock.saveData("anyKey", "anyValue") RETURNS SUCCESS

  // Act: attempt a read while cache is down
  READ_RESULT = serviceInstance.readData("missKey")

  // Assert: read fails, datastore NOT accessed directly
  ASSERT READ_RESULT IS FAILURE
  ASSERT dataStoreMock.get WAS NOT CALLED

  // Act: attempt a write while cache is down
  WRITE_RESULT = serviceInstance.writeData("anyKey", "anyValue")

  // Assert: write fails, datastore NOT accessed directly
  ASSERT WRITE_RESULT IS FAILURE
  ASSERT dataStoreMock.saveData WAS NOT CALLED
```

Again, in this implementation, the cache sits in the critical path, so any performance degradation or failure will directly affect the system's overall reliability and availability. Before assessing this impact, however, architects should factor some implementation details into their testing strategies.

First, cache performance issues may stem from the cache itself, usually either from resource exhaustion or operational concerns, such as high network latency or quorum delays in distributed caches. Issues can also stem from the backing datastore if it exhibits performance issues or simply isn't available. It might be leveraging fault-tolerance techniques like circuit breakers and/or retry logic, or resiliency techniques like replication. Finally, the application must implement specific logic to respond appropriately in the case of a catastrophic failure.

While the canonical pattern *never* bypasses the cache, production implementations often allow the application to circumvent the cache if it's not available. This approach comes with its own implementation concerns and trade-offs. For example, perhaps the application only bypasses the cache (if not available) on reads, with the understanding that this could potentially put a larger load on the primary datastore and affect overall responsiveness and read throughput. On a write, however, bypassing the cache isn't allowed, because this pattern's primary goal is to guarantee consistency between the cache and the primary datastore.

Architects must evaluate all of these factors when assessing system capabilities such as reliability, availability, and latency. Chaos engineering is the most effective approach for measuring and governing these capabilities in production-like conditions.

Write-Behind Cache Pattern Variant

The Read-Through/Write-Through Cache pattern prioritizes consistency by enforcing atomicity guarantees across both the cache and the primary datastore over write throughput and responsiveness. However, in some scenarios, the architect might value write throughput and responsiveness more than eventual consistency. The Write-Behind Caching pattern variation, also known as Write-Back Caching, addresses this exact need by decoupling cache updates from primary datastore persistence.

Recall that in the Read-Through/Write-Through Cache pattern, the writes to the cache and the primary datastore are both blocking operations. This

affords strong consistency between the cache and the primary datastore — either the writes both succeed or they both roll back. In the Write-Behind Cache pattern, data is written only to the cache, which immediately acknowledges the write back to the application and queues up the write to the primary datastore for later. Figure 29-5 illustrates the sequence of events for a write for the Write-Behind Cache pattern.

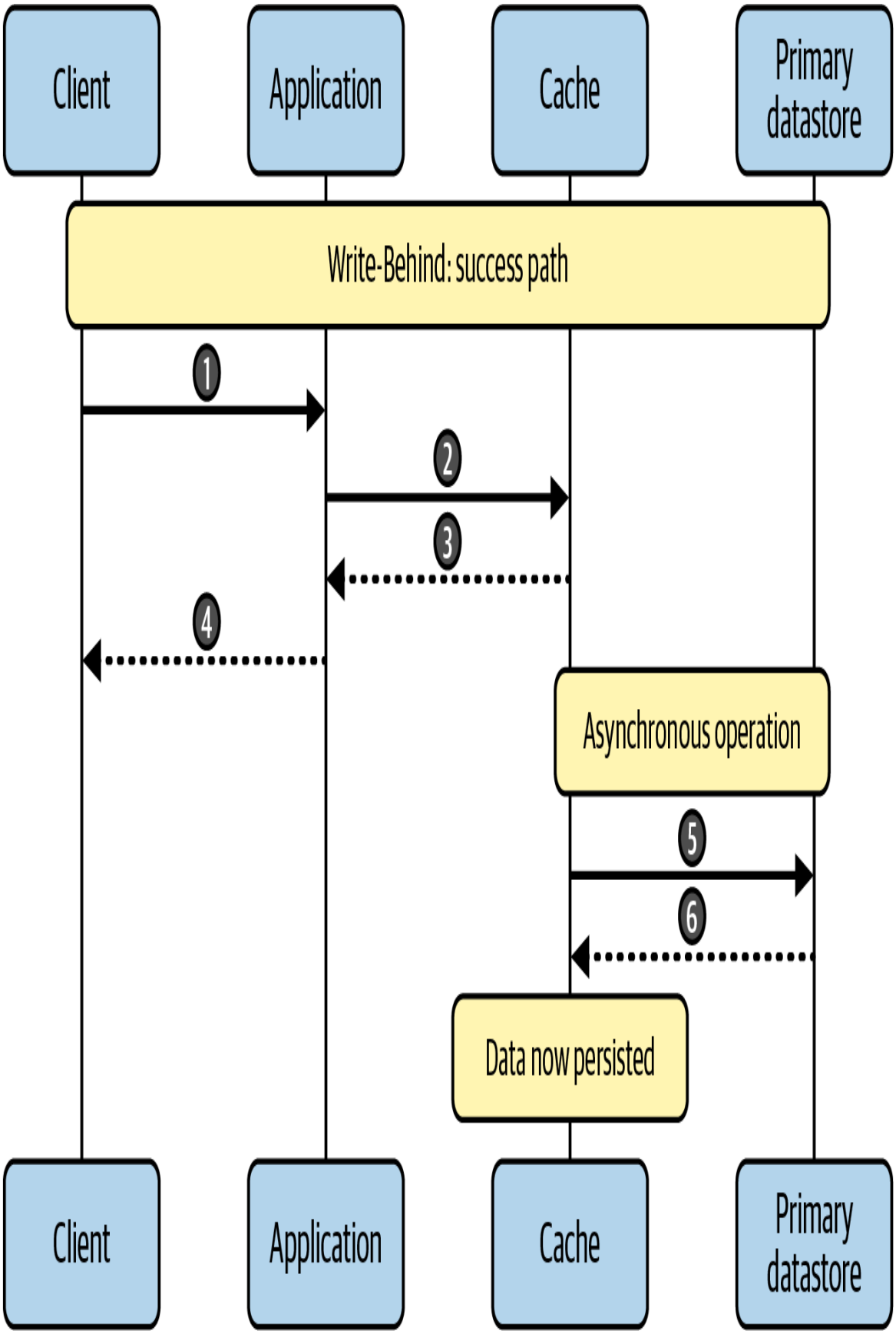

Client
Application
Cache
Primary datastore
Write-Behind: success path
1
2
3
4
Asynchronous operation
5
6
Data now persisted
Client
Application
Cache
Primary datastore

Figure 29-5. Write-Behind Cache pattern write/update success path sequence diagram

The steps are:

1. The client sends new or updated data to the application.
2. The application writes the new data or updates existing data in the cache.
3. The cache *immediately* acknowledges write success to the application.
4. The application acknowledges to the client that the update was successful (for example, with a `200 OK` HTTP status code).
5. The cache asynchronously (based on some criteria) writes to the primary datastore.
6. The primary datastore acknowledges a successful write.

The trigger for the cache to write to the primary datastore can be something as simple as a preconfigured delay or a batch threshold.

This variant flips the script on the Read-Through/Write-Through Cache pattern. Here, the authoritative source is no longer the primary data source but rather the *cache*, since the cache can temporarily hold data that hasn't yet been written to the persistent store. This is the trade-off: significantly improved write throughput and responsiveness for eventual consistency. Architects should carefully consider what should happen if the asynchronous write to the primary datastore fails. Should the cache try again after a backoff period, or is it OK to just lose the data? How should the system recover from this scenario? These scenarios can make all the difference in whether this pattern applies to a particular use case.

On the read side, however, Write-Behind Cache behaves exactly like the Read-Through/Write-Through Cache pattern.

This variant is extremely useful in write-heavy workloads where strong consistency isn't a mandate, such as logging or metrics collection.

Trade-Offs

Choosing a caching strategy like the Read-Through/Write-Through Cache pattern involves many trade-offs.

Pros

The advantages of applying this pattern include:

Data consistency

> The Read-Through/Write-Through Cache pattern shines when it comes to data consistency. It's often a great choice for architects looking to improve the read throughput of systems that need to ensure data integrity. Systems that provide reference data, like product catalogs or news websites, tend to be read-heavy and can benefit from this pattern. Because the cache and the primary datastore are updated together, this pattern minimizes the window of inconsistency between the two.

Simplicity

> Since the application isn't directly managing the interaction between the cache and the primary datastore, the application logic is significantly simpler. Things like error propagation and retry logic can sometimes be reduced to configuring the cache. This consolidation of caching logic into the cache layer itself reduces the surface area for bugs and simplifies testing.

Cons

The disadvantages of applying this pattern include:

Flexibility

> This pattern wrestles a lot of control away from the application over what data is cached and how. Imagine a system wanting to only cache data for a select type of user, set different TTLs for different kinds of data, or perhaps even use a different data model in the cache than in the primary datastore. Such flexibility is usually unavailable in this pattern.

Comparison Chart

When using caching, Table 29-2 is a comparison between using this pattern versus manually keeping the cache in sync by implementing specific application logic.

Table 29-2. Comparison chart for the Read-Through/Write-Through Cache pattern

Capability	Application-Managed Cache	Using Read-Through/Write-Through
IMPROVED		
Data Consistency	★	★★★
Simplicity	★	★★★
DEGRADED		
Flexibility	★★★★★	★★★

The following sections describe each of the capabilities found in Table 29-2.

Improved

The capabilities that are improved when applying this pattern include:

Data Consistency

> Caches inherently introduce the risk of data consistency issues in architectures — simply because yet another copy of the data can become stale. However, in the Read-Through/Write-Through Cache

pattern, the cache acts as the application's primary datastore. This means data consistency is significantly improved as compared to other caching implementations, where the application is responsible for keeping the cache up to date.

Simplicity

Many cache implementations can natively support a backing store, freeing the application logic from having to manage this complexity. However, simply introducing a cache presents some challenges, such as handling an additional dependency and another moving part within the architecture, configuring the cache, managing time-to-live (TTL), and choosing eviction policies.

Degraded

The capabilities that are degraded when applying this pattern include:

Flexibility

Managing the primary datastore is now the cache's concern. While this reduces the application's complexity, it also reduces flexibility and control around what items should be cached and for how long (varying TTLs).

Influences

This pattern is heavily influenced by the caching topologies used for the cache implementation. Different caching topologies bring their own sets of trade-offs, which architects should consider when making topological choices:

Single in-memory cache

In this topology, the cache runs in the same process as the application. Reads on a cache hit will be extremely fast. On a cache miss, a write, or an update, latency will be dictated by the time it takes to make a round

trip to the primary data source, assuming there's a network boundary. Since the cache is in memory, the application has a cold cache on restarts, so the consequent high cache miss rates will increase latency.

If the architecture achieves higher throughput by scaling application instances horizontally, then the architect must consider the *kind* of data being cached. Cache updates only affect the in-memory instance of the cache, so different instances can quickly exhibit inconsistencies. This pattern works well if the data being cached is static, request scoped, or the result of a computation. However, for data that's dynamic or changes quickly, architects must employ additional mechanisms, like sticky sessions, that may cancel out some of the benefits gained by scaling horizontally.

Finally, from a cost perspective, every instance having its own cache leads to a larger memory footprint and thus higher costs.

Client-server cache (remote cache)

In the Read-Through/Write-Through Cache topology, as far as the application is concerned, the cache *is* the primary data source. That makes this pattern much better suited to work with the client-server cache than the in-memory cache.

In a single-instance setup, on startup the application cache will be cold, increasing cache misses and resulting in higher latency. However, on single-instance restarts or if the application scales out horizontally, replicas will be presented with a warm cache. This helps with the availability, scalability, and elasticity of the system as a whole.

For multi-instance setups, all instances share the same cache. Not only does this mean not duplicating cached data (keeping costs low), it makes updating or invalidating the cache a lot easier. All it takes is for one instance to update the cache and the others simply benefit. Since all instances share the cache, the probability of instances reporting stale data is also lower, improving consistency.

On a cache miss, however, latency will be higher than with the single in-memory cache. This is because it takes the application multiple hops to get a chance to respond to the client. It goes first to the cache, only to get a miss, followed by another (blocking) call from the cache to the primary datastore. Updates take longer as well, since the path to update the cache is the same as to populate the cache on a miss.

Another downside is the centralization of the cache itself: any performance degradation in the cache or its backing data source will ripple across the architecture. Adding to this, the cache can become a single point of failure for the entire system.

Replicated in-memory cache

When combined with a replicated in-memory cache, the Read-Through/Write-Through Cache gets the benefits of both a single in-memory cache and a client-server cache, but with significant trade-offs.

Latency is low on a cache hit, since the cache is in memory. On a cache miss, write, or update, the only latency cost is the time it takes to interact with the primary datastore, usually over a network boundary.

With automatic cache replication, having a prepopulated cache is helpful for application restarts and creating new instances while horizontally scaling. Leveraging the cache's automatic replication improves consistency. This means any kind of data (whether static or fast changing) can be cached. Architects have no reason to reach for techniques like sticky sessions, since any instance can serve any request. This improves the architecture's availability, scalability, and elasticity, but increases costs, since every instance has a full copy of the cache.

Consistency guarantees, however, are a function of the kind of cache selected and the cache's architecture. While some replicated in-memory caches offer strong consistency (using techniques like leader selection), this comes at the cost of latency at write time (because all writes must go through the leader). The cache must write to the primary datastore

and replicate to all other nodes, both of which are synchronous operations in this pattern.

On the flip side, some architects opt for eventual consistency in caches to improve latency and throughput, despite its complexities and the challenges of maintaining consistency. In such cases, reads remain fast. Write operations, however, are followed by asynchronous replication to other nodes — introducing a window where other instances might be serving up stale data. Usually the most effective strategy is to leverage an aggressive TTL on cache entries, particularly for dynamic data (with the trade-off being a higher rate of cache misses). Other strategies resolve conflicts through versioning or a last-write-wins rule — all of which have significant trade-offs for throughput.

Overall, using the Read-Through/Write-Through Cache pattern with a replicated in-memory cache will use more bandwidth and incur more network overhead due to cross-instance chatter. The memory footprint per instance will be higher, as will the cost per instance.

Operationally, all of the complexities of managing a distributed system emerge when attempting to use the Read-Through/Write-Through Cache pattern with a replicated in-memory cache.

Caching Topology Comparison

Table 29-3 compares how the Read-Through/Write-Through Cache pattern performs across three different caching topologies.

Table 29-3. Comparison chart for the Read-Through/Write-Through Cache p caching topologies

Capability	Using Read-Through/Write-Through	In-Memory	Client-Server	Re In
IMPROVED				
Data Consistency	★★★	★★	★★★★	★
Simplicity	★★★	★★★	★★★★	★
DEGRADED				
Flexibility	★★★	★★★	★★	★

Improved

The following are capabilities that improve across caching topologies when using this pattern:

Data Consistency

> Of the three caching topologies, the client-server topology outshines the other two, since all writes or updates affect the shared cache. The single in-memory topology suffers because, while this pattern updates the primary datastore, the cache state is not shared across replicas. The replicated in-memory cache varies. Some architectures offer strong consistency as opposed to eventual consistency, with trade-offs.

Simplicity

Many caching technologies are embodied within the cache itself (simplifying its application logic) to natively support a persistent backend, failover logic, retries, and exception handling logic. Many single in-memory caching implementations require the engineering team to provide a loader and store, reducing this topology's simplicity. Simplicity also suffers a bit for replicated in-memory caching, since it introduces a distributed system that attempts to achieve either strong or eventual consistency.

Degraded

The following capability degrades across caching topologies when using this pattern:

Flexibility

Making the caching technology responsible for managing its own persistent datastore reduces its flexibility around what kinds of data it can cache. Of course, the engineering team can improve that flexibility by providing a loader and store implementation, as they do for many single in-memory implementations.

How to Govern This Pattern

In the Read-Through/Write-Through Cache pattern, the cache sits squarely *between* the application and the data source. In the purest implementation of this pattern (discussed under "Things of Note"), the system is expected to "fail fast," in that the cache can never be bypassed. This constraint has implications for the system's overall reliability. Indeed, *any* degradation in the cache's responsiveness or throughput has implications for the entire system. This is why the architecture must establish clear service-level indicators (SLIs) and service-level objectives (SLOs) for things like cache availability and latency. Furthermore, when the cache is unavailable, the entire system is unavailable. Architects should use chaos-engineering

techniques to enforce and validate these behaviors, and ensure that the system meets its expected SLIs and SLOs.

The Write-Behind variant suggests that the application *may* bypass the cache in certain scenarios — perhaps on reads, but not on writes. This reduces the entire system's overall responsiveness and read throughput to that of the primary datastore, with implications for the SLIs and SLOs, so it should be part of the testing strategy.

Related Patterns, Antipatterns, and Pitfalls

The following patterns are related to the Read-Through/Write-Through Cache pattern:

Cache Aside pattern (see Chapter 8)

> When choosing a caching pattern, architects must consider workload patterns, the kind of data being cached, and a multitude of other factors. The Cache Aside pattern (along with its variants) is an alternative caching pattern in which the cache sits *aside* from the critical path, allowing the application to bypass the cache if needed.

Event Sourcing pattern (see Chapter 16)

> The Read-Through/Write-Through Cache pattern and its Write-Behind variant have direct applications in Event Sourcing architectures, particularly for snapshot management.
>
> In Event Sourcing, a *snapshot* functions as a cached aggregate state that follows the Read-Through pattern. When loading an aggregate, the system attempts to load a snapshot (cache hit). If one exists, only events after the snapshot need to be replayed. If no snapshot exists (cache miss), the system must replay all events from the beginning and may create a snapshot for future reads. The Write-Behind variant is commonly used to create snapshots, which are generated asynchronously based on event count thresholds or access patterns,

rather than synchronously on every write. This improves write throughput.

Command Query Responsibility Segregation (CQRS) pattern (see Chapter 10)

CQRS is frequently used alongside both caching patterns and Event Sourcing. In CQRS architectures, the read side often employs caching strategies to optimize query performance, while the write side focuses on command processing. The separation of read and write models in CQRS naturally aligns with caching patterns, where the cache serves as an optimized read model that is distinct from the authoritative write-side datastore.

Chapter 30. Service Sharding Pattern

When creating distributed architectures like microservices, architects are commonly advised to avoid cross-service transactions, which can greatly increase the system's complexity. In fact, a number of patterns and techniques address this concern by finding ways to make workflows transactional (for example, see the Compensating Updates pattern in Chapter 11). However, another way to avoid the complexity of cross-service transactions is by applying *service sharding*: collapsing the transactional context back down to the service boundary.

The metadata for the Service Sharding pattern appears in Table 30-1.

Table 30-1. Metadata for the Service Sharding pattern

CATEGORY	METADATA
CONTEXT:	Distributed Architecture, Event-Driven Architecture, Message-Based Architectures, Microservices, Service Granularity, Workflow
IMPROVED:	Data Consistency, Data Integrity, Domain Alignment, Elasticity, Scalability, Simplicity, Testability
DEGRADED:	Adaptability, Evolvability

How Should I Implement Atomicity in My Distributed Architectures?

One of microservices' key design philosophies is decoupling as much as possible, because that high degree of isolation allows teams to make changes to one part of the architecture without affecting other parts. We cover five different types of coupling in this book, defined via the architecture quantum in "Architecture Quantum" and summarized here:

Static

Covers the dependencies and infrastructure (the "plumbing") the system needs to run.

Dynamic

Communication coupling between services or event processors.

Contract

When multiple services share the same contract. If one service requires the contract to change, it must change in the other one as well, which is a form of coupling.

Temporal

Mostly exemplified by startup dependencies between services. This is common in caching scenarios.

Atomic

Atomicity refers to the ACID principle (covered in "ACID Versus BASE") of unified workflows, which ensures that in a multiservice process, either all or none of the data changes occur, depending on the overall success of the workflow. In other words, the workflow is *atomic* from a data consistency standpoint.

Atomicity is the focus of this particular pattern. No matter how much architects try to avoid workflows that include atomicity, sometimes they have no choice because of other trade-offs.

The Service Sharding pattern attempts to avoid workflow-level coupling for atomicity by modifying the granularity and responsibilities of services. This isn't always possible, but it's a worthy goal for architects who want to avoid the headaches of cross-service transactions.

WHY AVOID CROSS-SERVICE TRANSACTIONS?

One of the most difficult problems in microservices involves atomicity across services. In fact, the entire second part of our book, *Software Architecture: The Hard Parts*, covers the interlocking forces that come into play when architects deal with atomicity: communication (synchronous/asynchronous), consistency (atomic/eventual), and coordination (orchestration/choreography). Each of these forces influences the others, so architects must consider them as a holistic set, rather than individually. In that book, we analyzed all eight possible ways these forces can combine when building atomic workflows, each of which has a distinct set of trade-offs. It's crucial to consider all three forces when changing one, because they're conjoined.

Applying the Pattern

This pattern aligns the architect's design with the underlying philosophy of microservices, so it's almost always desirable. However, it isn't always obvious *how* to reorient a workflow to refocus on atomicity, so let's look at an example.

Consider a company doing high-volume credit card processing for a particular type of card. Multiple "branded" cards exist for each card type, each offering a bewildering variety of discounts and other perks: for example, a customer with a hotel-branded credit card can opt for discounts on hotel rooms or rental cars. Each branded card has a unique set of charging criteria to apply to each transaction. Since this system needs to scale up to handle a high volume of transactions, the architect decides to

implement credit card processing using the Choreography pattern (see Chapter 9), as shown in Figure 30-1.

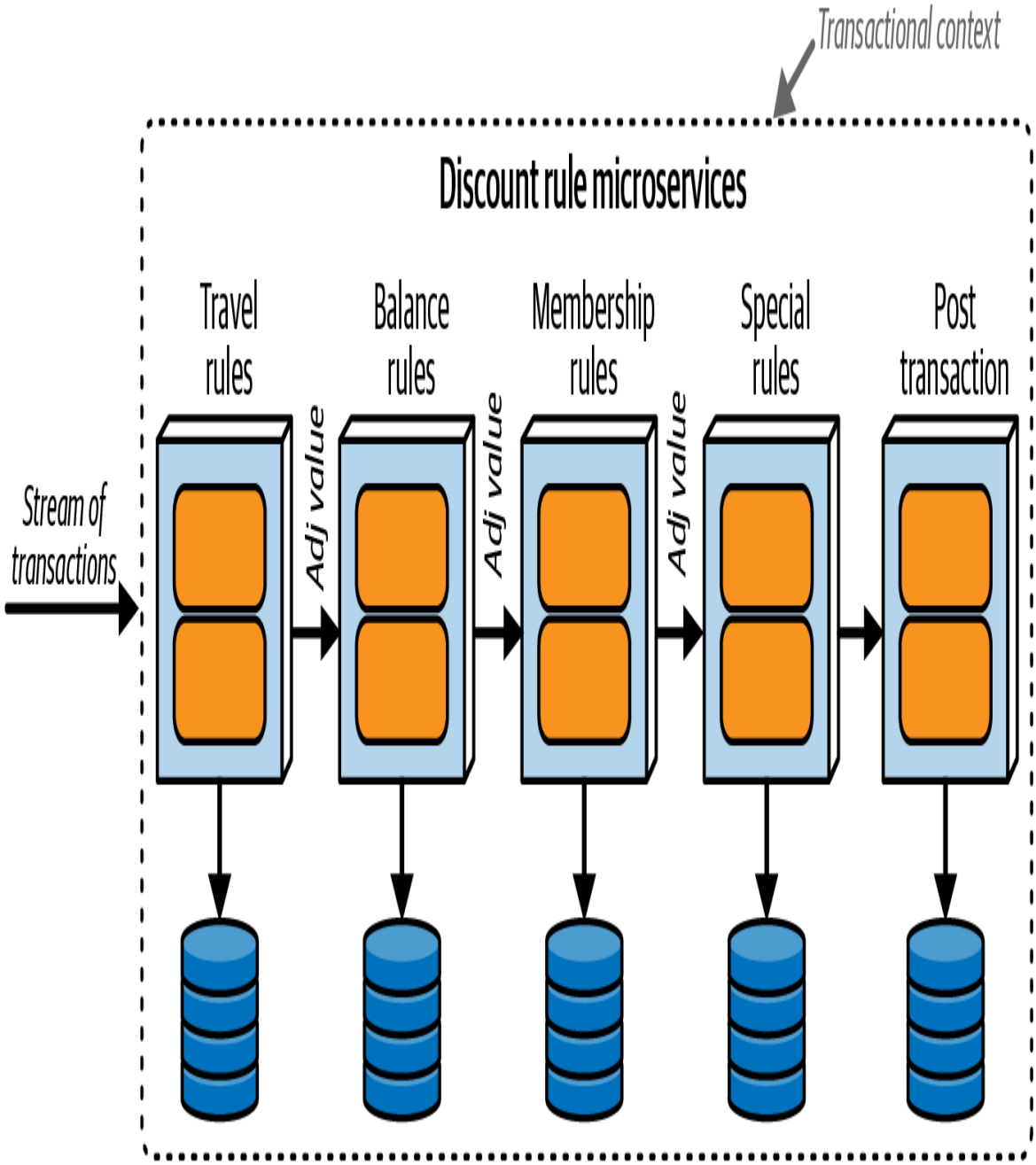


Figure 30-1. The Choreography solution for high-volume credit card processing

Credit card transactions stream into the workflow, where each microservice applies the discount for that particular card and customer. By the end of the workflow, the system has applied all relevant discounts to calculate the final transaction amount.

In Figure 30-1, the dotted line delineates the transactional boundary for this workflow. If one discount fails to be applied for some reason, the architect wants the whole transaction to fail rather than partially applying the rules (following the definition of atomicity). Consider the error case shown in Figure 30-2.

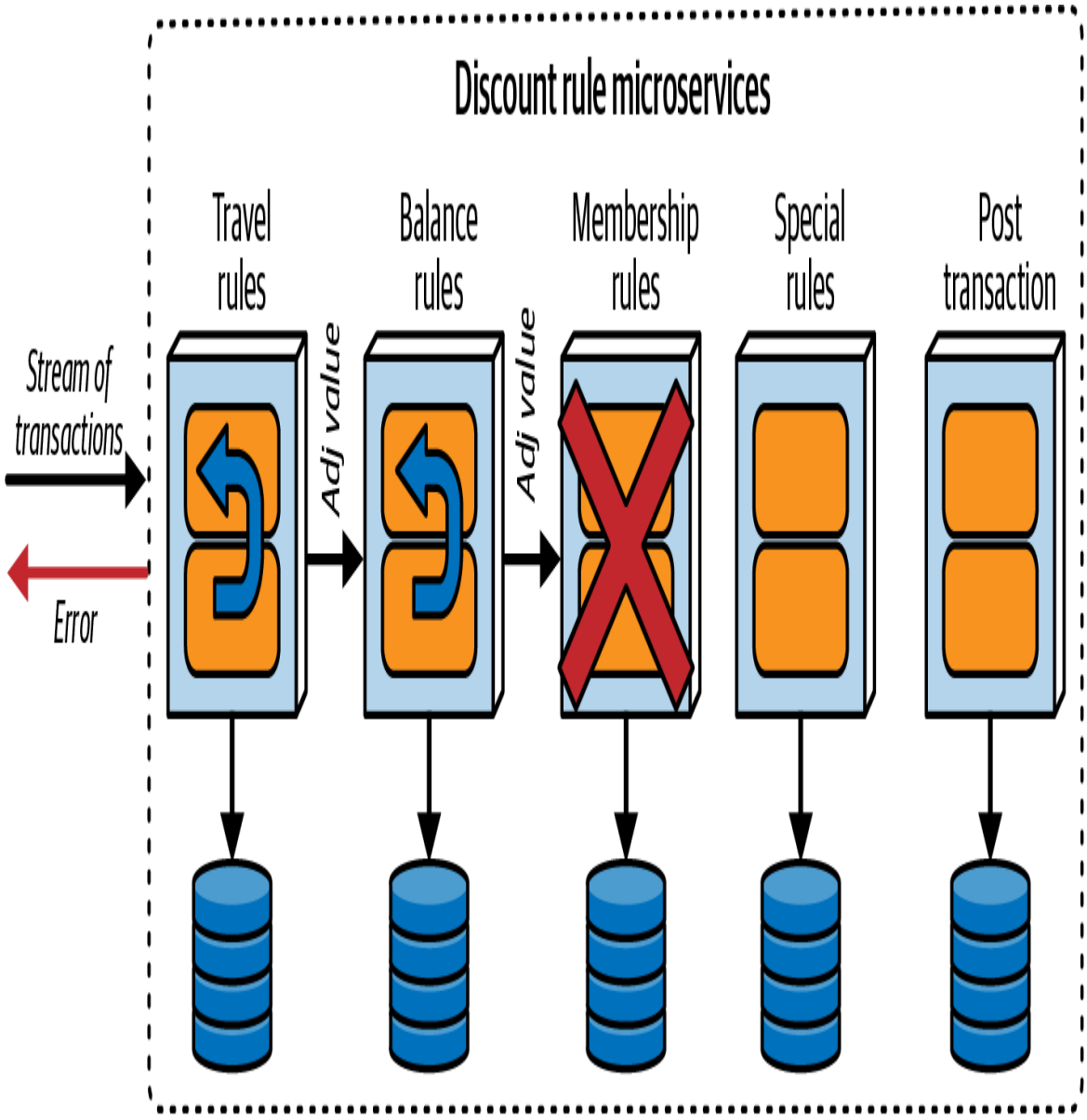


Figure 30-2. When one rule fails, all previously applied rules roll back

In Figure 30-2, when the workflow attempts to apply `Membership Rules`, it fails for some reason. The architect wants to roll back the

changes to both `Travel Rules` and `Balance Rules` to restore everything to the previous state.

In *Software Architecture: The Hard Parts*, we analyzed eight different types of *sagas*, or transactional workflows, each with a different combination of what we called *primal forces* (communication, consistency, and coordination), naming each for a whimsical metaphor. This particular workflow is an example of either a "Phone Tag saga" or a "Horror Story saga," which we determined to be the two most difficult transactional workflows. Both variations use atomic consistency and choreography for coordination. The Phone Tag saga uses synchronous communication. In that case, the architect has no real options — each request blocks and waits for the return of the next service in line. The flip case, the Horror Story, uses asynchronous communication and has the opposite problem: interactions can happen arbitrarily. Additionally, the requirement that the entire workflow be atomic adds a great deal of complexity to the choreographed services, since they must handle error states, boundary conditions, and so on (see Chapter 9 for a complete discussion).

The architect would like to use asynchronous communication and choreography to achieve the required scalability. However, the atomicity requirement adds tremendous complexity to the services and workflow.

Architects can't always implement the Service Sharding pattern, because it requires some luck as to how their problem domain works. Fundamentally, making a workflow transactional is a form of *technical partitioning*: wrapping an atomic context around several services. This pattern encourages architects to instead use *domain partitioning*, pulling the atomic context back into the service boundary.

TECHNICAL VERSUS DOMAIN PARTITIONING

In our book, *Fundamentals of Software Architecture* (now in its second edition), we distinguish architecture styles as partitioned either *technically* or by *domain*, depending on the architect's motivation for how they separate components. For example, the traditional layered architecture separates components into layers, such as `Presentation`, `Persistence`, `Business Rules`, and so on. This architecture is technically partitioned because the architect chose to separate the parts by their technical responsibilities, like presentation.

The alternative approach, exemplified by Domain-Driven Design (DDD), separates components into workflows or bounded contexts rather than technical behavior. For example, components in this architecture reflect domain concerns, such as `CatalogCheckout` or `ShipToCustomer`.

Generally, in microservices, which is a domain-centric architecture, teams should try to find domain-partitioned solutions (like this pattern).

Consider the alternative service design shown in Figure 30-3. The architect designs each service to represent a bank as a microkernel (an architecture with an unchanging core and plug-ins for customizability). The team builds the custom rules for each bank as plug-ins for the bank service, and the team creates an instance of a microservice for each bank.

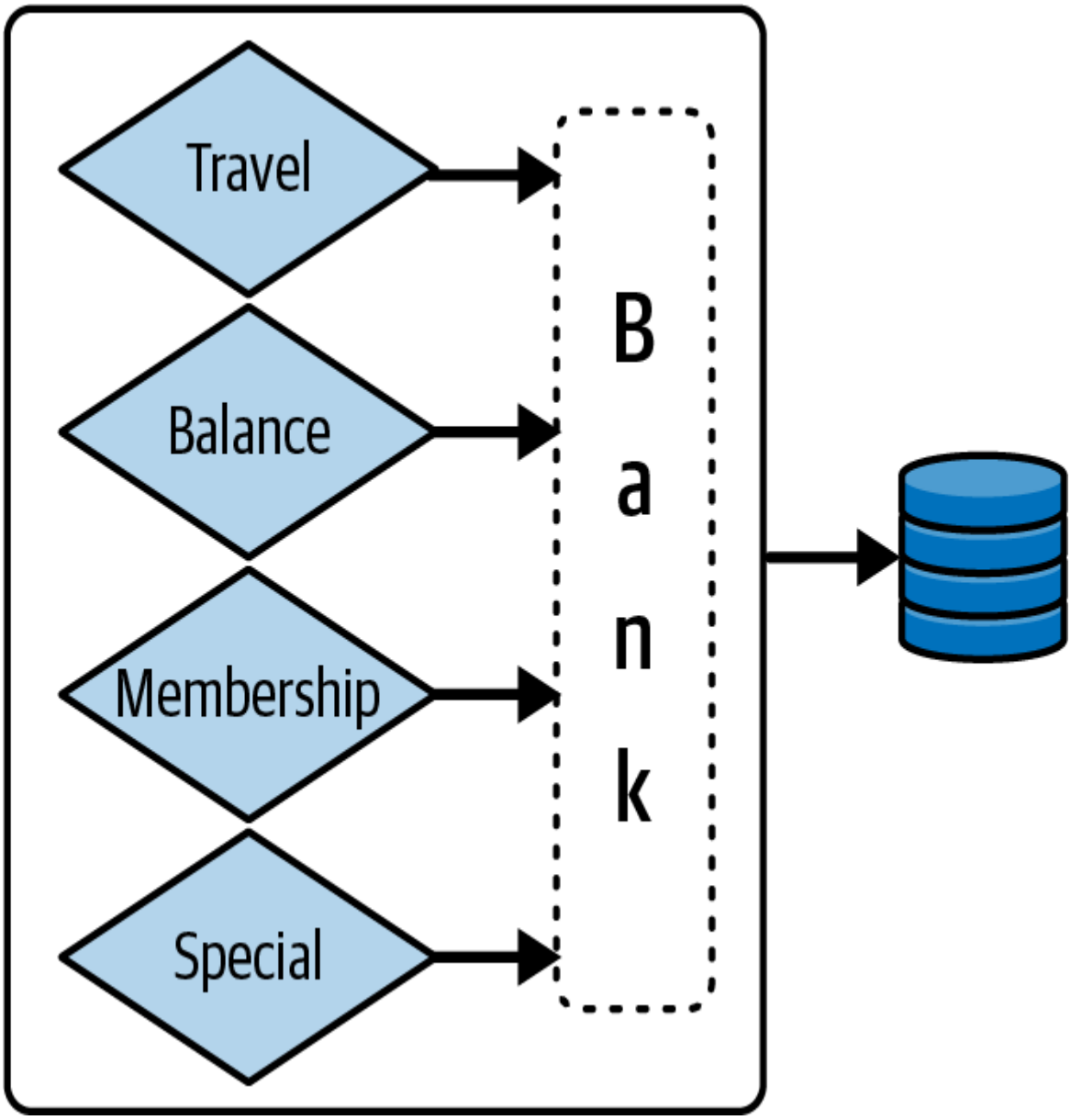


Figure 30-3. Each bank service implements a microkernel, with plug-ins for varying discount rules

In Figure 30-4, transactions flow through a service that dispatches them based on their bank brand. Each bank brand has a service that includes all the custom discount rules for that organization. Because each service "knows" all the rules that apply to that bank, it can apply all the discounts (or not) within the transactional context of a single service rather than stretching them across several services. Now, the transactional context has shrunk back to the service boundary. If an error occurs while applying a

discount, the team can easily roll back the previous discounts within the same service context.

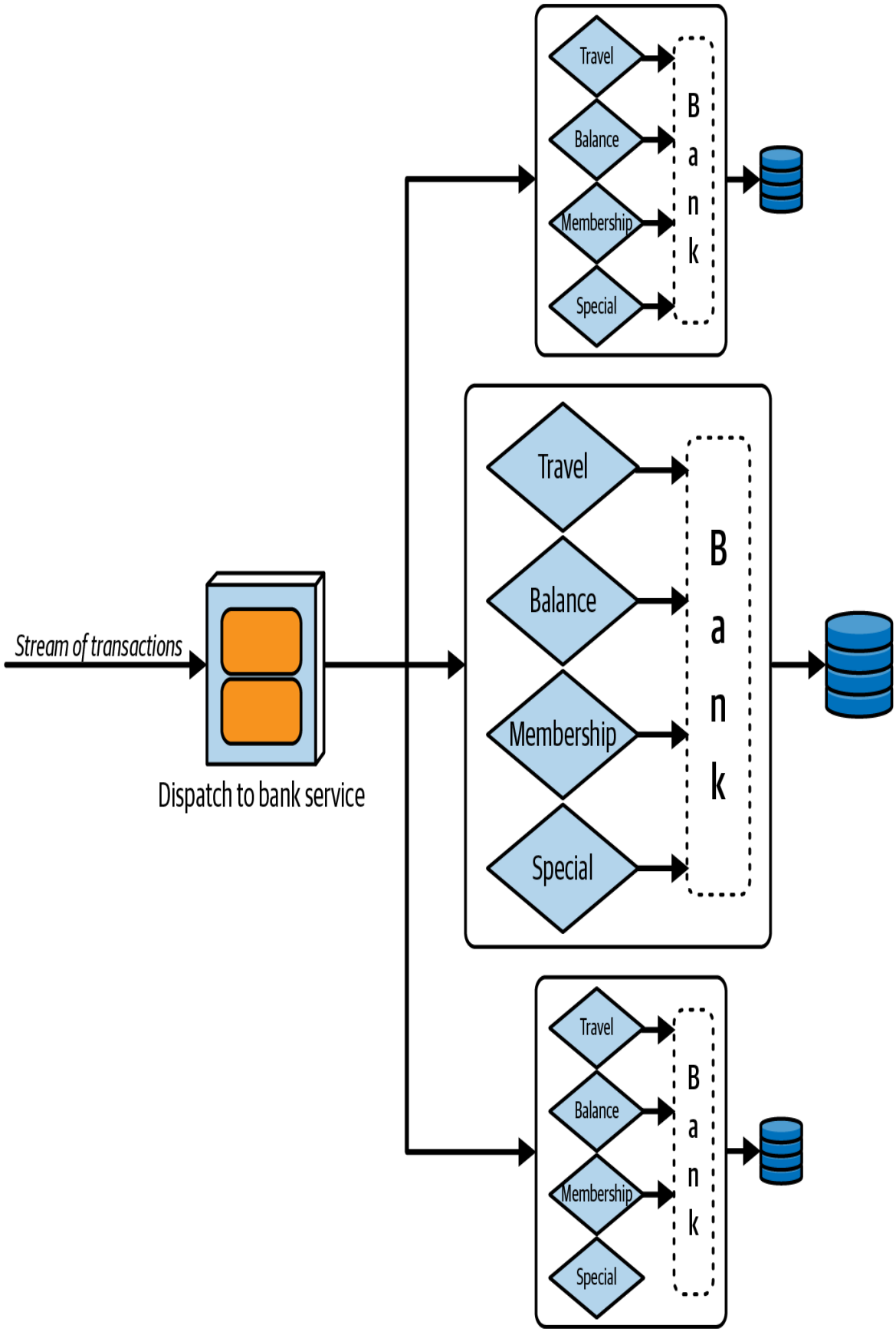

Travel
Balance
Membership
Special
Bank
Travel
Balance
Membership
Special
Bank
Stream of transactions
Dispatch to bank service
Travel
Balance
Membership
Special
Bank

Figure 30-4. Sharding the atomicity to the service level eliminates the cross-transactional workflow

The key question that enabled this design is: is any one bank's transactional volume greater than what a single microservice instance can handle? If not, then this architecture allows a service for each bank, with each one owning its transaction context, greatly simplifying the architecture.

Trade-Offs

The trade-offs for this pattern are mostly great…*if* architects can find a way to take advantage of them by finding the correct atomic context.

Pros

This pattern offers significant advantages:

Better alignment with microservices

> The microservices design philosophy encourages restricting transactions to service boundaries, so this pattern aligns perfectly.

Better operational architectural capabilities

> Because atomicity doesn't extend into the workflow, operational architectural capabilities (such as responsiveness, scalability, elasticity, and others) universally improve. One key factor that negatively affects these capabilities is that teams must coordinate to handle cross-service atomicity.

Cons

The biggest problem with this pattern is the lack of guarantee that an architect can apply it, since they must find just the right partitioning to meet their goals:

Abstraction

Architects often try to isolate microservices as much as possible, yet many business workflows require atomic coordination across services. In those cases, architects can apply this pattern, but it may have negative side effects, such as breaking bounded contexts or building mini-monoliths within a microservices architecture.

Comparison Chart

This pattern has many impacts, mostly good, when architects can apply it. These are shown in Table 30-2.

Table 30-2. Comparison chart for the Service Sharding pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Data Consistency	★★	★★★★★
Data Integrity	★★★	★★★★★
Domain Alignment	★	★★★★★
Elasticity	★★	★★★★★
Scalability	★★	★★★★★
Simplicity	★★★	★★★★★
Testability	★★	★★★★★
DEGRADED		
Adaptability	★★★★★	★★★
Evolvability	★★★★★	★★★★

The following sections describe each of the capabilities found in Table 30-2.

Improved

The following capabilities are improved when using this pattern:

Data Consistency

This pattern greatly simplifies data consistency (which is about the synchronization of data) within a distributed architecture, which is its key advantage. Aligning atomic behavior with the preferred topology of the architecture means that typical choices for microservices work, without elaborate workarounds. Data consistency (rather than data integrity) is the main focus of this pattern, which is why its impact is slightly larger.

Data Integrity

Data integrity is about data correctness, and this pattern is designed to ameliorate many of the problems inherent in trying to create transactional workflows. If the architect can shrink the transactional scope back to the service level, the need for saga transactions goes away. This pattern leverages database mechanisms to ensure correctness; data never suffers from network effects if it's self-contained within the service.

Domain Alignment

Keeping the atomic context to the service boundary is the preferred abstraction for microservices. Building workflows that spread atomic context across several services leads to *technical abstraction*, the opposite of the microservices architectural style.

Elasticity

One of the hindering factors for elasticity occurs when teams must handle data coordination to handle data atomicity (data consistency and integrity): architectures that require lots of coordination can't scale as well as architectures where each service is transactionally stateless, which aligns the topology with the microservices philosophy.

Scalability

Similar to elasticity, coordination for the purposes of keeping data correct and in sync harms scalability; isolating transactional context to services allows the architecture to scale more easily. Because the architect no longer needs to scale the entire workflow, they can apply microservices practices such as redundancy and service-level scaling.

Simplicity

Collapsing the atomic context of a workflow to a single service context removes the need for error handling and cross-service coordination in unexpected circumstances.

Testability

It's much easier to test designs that don't have elaborate workflows to handle compensating updates and other patterns (see the Compensating Updates pattern in Chapter 11 for details).

Degraded

The following capabilities are degraded when using this pattern:

Adaptability

Because each service is completely self-contained, it's hard to adapt the service without replacing it. This pattern collapses what would normally be a workflow into a single service boundary, making changes self-contained.

Evolvability

Evolvability is slightly less impacted than adaptability: it's more common when evolving a highly distributed architecture like microservices to replace services with new features, allowing DevOps practices such as A/B testing, feature toggles, and so on. Thus, it's more natural to wholly replace fine-grained services when evolving the architecture, less so when adapting it to new behavior.

Influences

This pattern changes the topology and some of the service interaction capabilities of the distributed architecture.

Service Granularity

In our book, *Software Architecture: The Hard Parts*, we describe techniques called *disintegrators* (which make services smaller) and *integrators* (which make services larger). One of the integrators is *service atomicity*: if an architect has broken their services down to be very small and granular (which may tend toward the Grains of Sand antipattern, discussed in Chapter 21), they discover that they need to create cross-service transactional behavior. One of the solutions to that problem, rather than implementing an atomic workflow, is to collapse the atomic behavior back into a single service, as suggested by this pattern.

Architecture Quantum

As with many of the structural patterns in distributed architecture like microservices, this pattern impacts several of the coupling capabilities of the architecture quantum:

Static coupling

> Static coupling refers to dependencies and infrastructure coupling; this pattern attempts to remove all external static coupling.

Dynamic coupling

> *Dynamic coupling* refers to the impact of synchronous versus asynchronous communication on the pattern. The influence, in this case, is toward removing communication completely: with this pattern, the architect no longer has to decide which kind of communication to use for the atomic workflow because the workflow disappears into the service boundary. As we mentioned above, architects must consider communication, consistency, and coordination when evaluating atomic

workflows. This pattern greatly simplifies the architect's decision process by eliminating all three concerns: it involves no communication, transactions are atomic within the service boundary, and no orchestration or choreography is needed.

Temporal coupling

This type of coupling might occur if the sharded service is using an in-memory cache to improve operational capabilities, combining both static coupling and (briefly) dynamic coupling (at startup).

Contract coupling

When designing a multiservice workflow, the architect must decide what information should pass between the services, which requires a contract, usually expressed by some protocol like REST or gRPC. (For our purposes in this book, a *contract* represents the data and metadata passed when two services exchange information.) This pattern reduces potential contract coupling. When the architect can eliminate the workflow, they don't need a contract (or versioning, lifecycle, or other related concerns).

Atomic coupling

This pattern explicitly eliminates this type of coupling by collapsing atomic concerns into the service boundary, aligning it with the microservices design philosophy.

In many ways, this pattern exists to make some of the trickier problems in microservices go away.

How to Govern This Pattern

Architects' primary concern when governing this pattern relates to its core purpose: atomicity (data consistency and integrity). When choosing this pattern, they must ensure that the atomic context doesn't extend past the

service boundary. Because they have consolidated behavior into a service granularity that is typically larger than nominal, extending the atomic context to another service defeats the point of this pattern and makes the result worse.

For example, say an architect decides to use this pattern to isolate atomicity, making for a larger-than-ideal service. Over time, the system needs to incorporate more behaviors, involving the larger service in an atomic workflow. Now the architect has the problems of both a larger context *and* distributed atomicity, which is worse than just creating an atomic workflow from the beginning.

We recommend using this pattern judiciously, and eagerly refactoring service granularity as workflows evolve.

Related Patterns, Antipatterns, and Pitfalls

This pattern deals with atomicity and service granularity, relating it to patterns with similar scope:

Compensating Updates pattern (see Chapter 11)

> Workflows that use the Compensating Updates pattern may restructure their solution toward this pattern to solve update complexity.

Orchestration pattern (see Chapter 27)

> Architects can sometimes avoid using the Orchestration pattern when using this pattern.

Choreography pattern (see Chapter 9)

> Architects can sometimes avoid using the Choreography pattern when using this pattern.

Grains of Sand pitfall (see Chapter 21)

> Service Sharding is often the remedy when services have been decomposed too finely — the Grains of Sand pitfall — and that

excessive granularity forces transactions to span multiple services. Collapsing the atomic context back into a single, coarser-grained service trades some granularity for a simpler transactional boundary, pulling the design back from the pitfall.

Chapter 31. Stamp Coupling Antipattern

When services communicate with one another, they must agree on what kind of data to pass back and forth between them: in other words, their contract. A *contract* specifies what attributes (data) are sent to message consumers, and in some cases the data types for each attribute. This chapter describes how certain practices in creating contracts between services or systems can lead to the Stamp Coupling antipattern.

Because this chapter deals with an antipattern, the Improved and Degraded metadata categories in Table 31-1 refer to the capabilities that are correspondingly improved and degraded when *adopting* this antipattern.

Table 31-1. Metadata for the Stamp Coupling antipattern

CATEGORY	METADATA
CONTEXT:	Contracts, Event-Driven Architecture, Message-Based Architectures, Microservices
IMPROVED:	Adaptability, Evolvability, Simplicity
DEGRADED:	Bandwidth Usage

How Specific Should My Contract Be?

Architects are faced with myriad decisions when creating contracts between services. They first need to determine what data the receiving service needs and what data is available from the sending service. Then they have to

decide on the format of the contract (JSON, XML, Object, and so on), and also whether the contract will be loose or strict. A *loose* contract consists of an agreed-upon collection of name-value pairs without an associated schema definition. A *strict* contract, on the other hand, has an associated schema exactly specifying the contract attributes, the associated data types, and sometimes even the validation rules. The following schema is an example of a typical JSON strict contract:

```
{
 "$schema": "http://json-schema...",
 "properties": {
 "order_id": {"type": "string"},
 "customer_id": {"type": "string"},
 "item_id": {"type": "string"},
 "qty": {"type": "number", "minimum": 1},
 ...
}
```

Another decision architects face is how specific the contract should be about the data it includes. This can be challenging if the data requirements for the receiving service are not known yet, or the data requirements for the receiving service continuously change. This decision gets even more daunting in event-driven architectures, where multiple message consumers are likely, each with its own data requirements.

In response to these challenges, architects sometimes choose to simply include *all* of the data they have available in the sending service, letting message consumers pick and choose what data they need to process. This simplifies the architecture because the sending service doesn't have to be concerned about maintaining separate contracts for each receiving service. This, in turn, reduces coordination and friction between the services.

To illustrate this technique, suppose a help desk associate at a major worldwide company creates a trouble ticket for a customer having issues with one of its software products. Processing the incoming trouble ticket requires the coordination of four services, as shown in Figure 31-1.

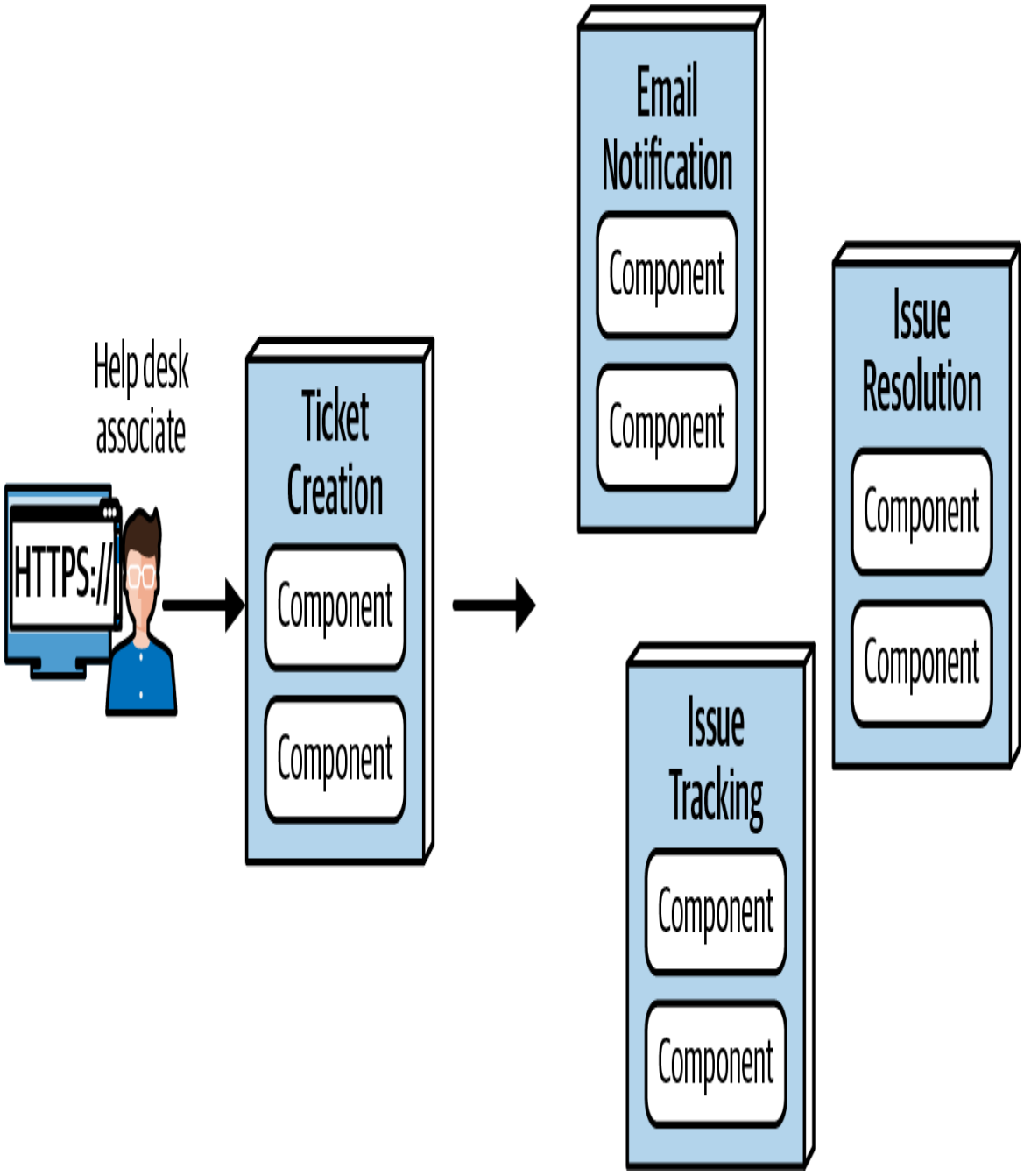


Figure 31-1. The `Ticket Creation` service must coordinate with three other services to process an incoming trouble ticket

The `Ticket Creation` service receives and validates all of the trouble-ticket information, which contains 50 attributes from the user interface with a total average size of 4 MB (including attachments), and inserts this data into its database. It must then coordinate with three other services, each of which owns its own database, to start processing the trouble ticket:

`Email Notification` *service*

Emails the customer confirming that the company has received the trouble ticket.

`Issue Resolution` *service*

Assigns the trouble ticket to a product specialist based on the software product involved and makes all relevant information available to the product specialist to resolve the issue.

`Issue Tracking` *service*

This auditing-based service records all trouble tickets entered for each software product. The reporting and analytics teams and management use it to track the number of issues associated with each product they support.

Table 31-2 shows the partial trouble ticket data needed by each of these services. The term *Information* on four of the data items refers to numerous attributes contained within that context. For example, the data relating to the single *Customer Company Information* data attribute contains numerous attributes, including business name and address, licensing information, and so on.

Table 31-2. Trouble ticket data needed by each service

DATA	Email Notification	Issue Resolution	Issue Tracking
Ticket Number	√	√	√
Customer ID		√	
Customer Name		√	
Customer Email	√	√	
Customer Phone		√	
Customer Address			
Software Product Information	√	√	√
Problem Description	√	√	
Problem Category		√	√
Ticket Status			
Control Code			
Help Desk Call Log			

DATA	Email Notification	Issue Resolution	Issue Tracking
System Config Information		√	
Supporting Documentation		√	
Size	1 K	2,200 K	1 K

Since these services have no dependency on each other and can be performed in any order, the architect decides to simplify the workflow by having the `Ticket Creation` service publish a single event containing all of the ticket information, as shown in Figure 31-2, using a strict contract (JSON schema) for the event payload.

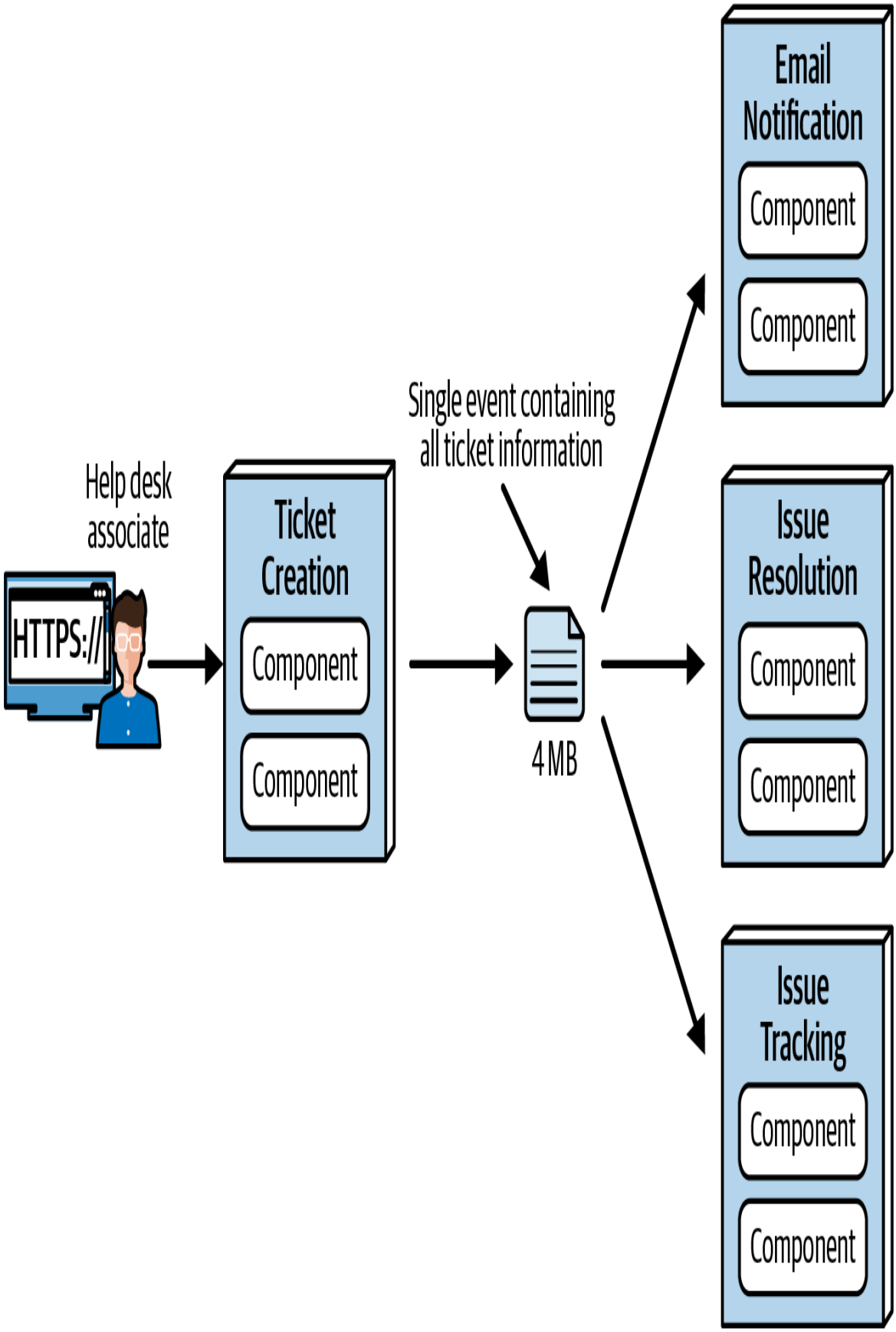

Email Notification
Component
Component
Single event containing all ticket information
Help desk associate
Ticket Creation
Component
Component
HTTPS://
4 MB
Issue Resolution
Component
Component
Issue Tracking
Component
Component

Figure 31-2. The `Ticket Creation` service publishes a single event containing all of the trouble ticket information

By sending only a single event containing all the data, the `Ticket Creation` service doesn't have to worry about maintaining separate contracts — the services can simply respond to the single event, pulling off whatever data they need for their processing. While this technique seems straightforward, efficient, and simple, it can lead to a common antipattern called Stamp Coupling.

Why Is This an Antipattern?

Stamp coupling refers to a form of coupling where a group of services all share the same composite data structure, but use only a few attributes from it. In this case, a change to the composite data structure for data that some services don't need might require an unnecessary change to that receiving service. Stamp coupling gets its name from the late 1960s, when COBOL data records were so static that companies actually made rubber stamps of them. When business analysts created a new specification or requirement, a technical analyst would locate the COBOL record containing the required data field, find the corresponding rubber stamp, and literally stamp the specification with it.

Two issues that can cause the practice of sending extraneous contract data to become an antipattern are *contract management* and *network bandwidth usage*. The following sections describe each of these issues.

Contract Management

In our prior example, the `Ticket Creation` service published an event using a strict contract containing *all* of the trouble ticket information (50 attributes). Each of the three responding services must thus adhere to this contract:

```
{
 "$schema": "http://json-schema...",
```

```
 "properties": {
 "ticket_num": {"type": "number"},
 "product_id": {"type": "string"},
 "customer_id": {"type": "string"},
 "control_num": {"type": "number"},
 ...
}
```

Now suppose the business wants to change the `control_num` attribute, representing an internal control number used for auditing purposes, from a numeric value to a string value containing a Universally Unique Identifier (UUID). The developer would need to modify the `Ticket Creation` service and the corresponding trouble ticket schema to reflect this attribute change, as follows:

```
{
 "$schema": "http://json-schema...",
 "properties": {
 "ticket_num": {"type": "number"},
 "product_id": {"type": "string"},
 "customer_id": {"type": "string"},
 "control_id": {"type": "string"}, ❶
 ...
}
```

❶ The developer renamed the former `control_num` attribute to `control_id` and changed the data type to a `string` value.

Even though the `Email Notification`, `Issue Resolution`, and `Issue Tracking` services don't need or care about this attribute, they still have to change to adopt the new schema definition. Contract versioning can sometimes help defer this kind of change, but eventually the three services will need to change to accommodate the new contract.

It's mostly strict contracts — those containing a schema — that are negatively impacted by this antipattern, but schema-less loose contracts can be affected as well, particularly when changing or removing attributes from the payload of a message. For example, suppose a developer modifies a loose-contract payload created by the `Ticket Creation` service by

splitting up the single `custName` attribute into separate `custFirstName` and `custLastName` attributes, as follows:

```
// loose-contract payload before change
{
  "ticketNum": "98765",
  "custId": "123",
  "custName": "Clark Kent",
  ...
}

// loose-contract payload after change
{
  "ticketNum": "98765",
  "custId": "123",
  "custFirstName": "Clark",
  "custLastName": "Kent",
  ...
}
```

The `Email Notification` and `Issue Tracking` services won't break as a result of this change because neither service uses or cares about the customer's name. However, the `Issue Resolution` service *will* break because it needs this information and would thus need to change as a result of the payload modification.

We discuss techniques that help address the contract change issues associated with this antipattern in "How to Avoid This Antipattern".

Network Bandwidth Usage

Communication between services usually means passing data back and forth, consuming network bandwidth. Sending large amounts of extraneous information in the payload of the message can significantly increase bandwidth usage, which can become an issue for high-volume and large data payloads.

In our prior example, the `Ticket Creation` service published an event containing all of the data for the trouble ticket entered by the help desk associate — 50 attributes with an average size of 4 MB for the entire

payload. Given this is a global company with hundreds of software products, the volume of trouble tickets ranges anywhere from 1,000 to close to 10,000 trouble tickets per day. Table 31-3 shows the bandwidth usage and associated costs (assuming an average cloud bandwidth cost of $0.03 per GB) for trouble ticket creation events published by the `Ticket Creation` service during a typical eight-hour day.

Table 31-3. Bandwidth usage and costs for sending all of the ticket data in the event payload

Volume/day	Bandwidth/day	Cost/day
1,000	4 GB	$0.12
10,000	40 GB	$1.2
100,000	400 GB	$12.0
1,000,000	4,000 GB	$120.0

For low volumes of trouble tickets, this is unlikely to be an issue. However, if the volume increases, it can quickly become a problem in both bandwidth usage and cost, particularly for cloud-based deployments. Keep in mind that the cost and bandwidth usage figures in Table 31-3 are for a *single* event — the one published from the `Ticket Creation` service. For systems that trigger hundreds of different types of events, the aggregated costs and bandwidth usage of running the entire system can rise significantly.

Now compare the same results, but this time assuming that the `Ticket Creation` service sends *only* the information needed by each service, either in a single event or in separate messages for each service. Based on the size calculations shown in Table 31-2, this would add up to 2,200 K for the `Issue Resolution` service, 1 KB for the `Email`

`Notification` service, and 1 KB for the `Issue Tracking` service, as shown in Table 31-4.

Table 31-4. Bandwidth usage and cost for sending only necessary ticket data in the event payload

Volume/day	Bandwidth/day	Cost/day
1,000	2.2 GB	$0.066
10,000	22.2 GB	$0.66
100,000	222 GB	$6.6
1,000,000	2,222 GB	$66.0

As Tables 31-3 and 31-4 illustrate, removing large volumes of extraneous attributes and sending only the data each service needs significantly reduces both cost and bandwidth usage.

How to Avoid This Antipattern

Fortunately, it isn't hard to avoid this antipattern. The following sections describe each of these techniques.

Consumer-Driven Contracts

One way to avoid this antipattern is to send only the data each receiving service needs. In our example, this would mean sending three separate messages (one for each service), as shown in Figure 31-3, rather than a single consolidated event.

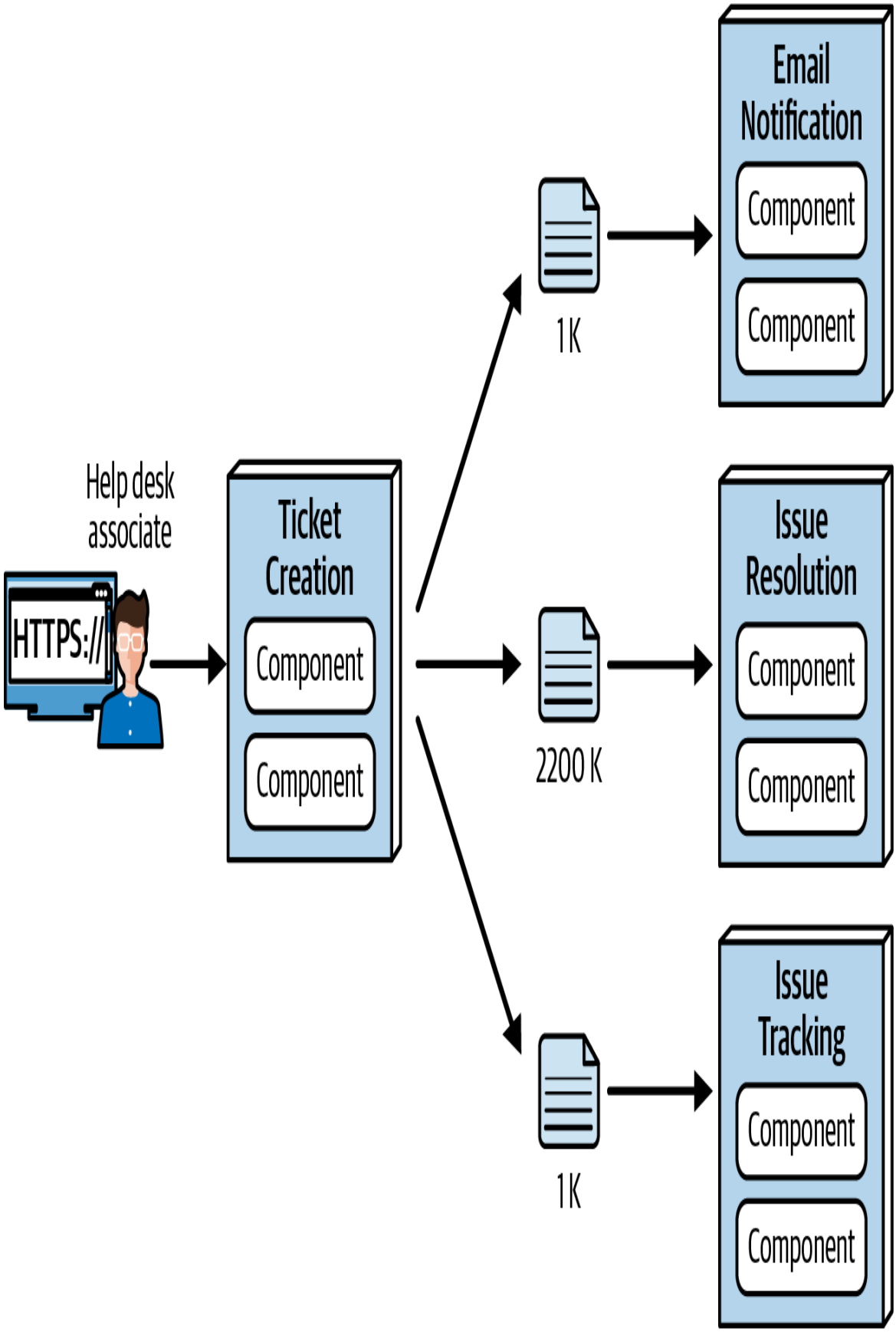

Email Notification
Component
Component
1K
Help desk associate
HTTPS://
Ticket Creation
Component
Component
Issue Resolution
Component
Component
2200 K
Issue Tracking
Component
Component
1K

Figure 31-3. The `Ticket Creation` service sends a separate message to each service with only the data each needs

This technique, which is a form of the *consumer-driven contract*, avoids the Stamp Coupling antipattern (and hence reduces both bandwidth usage and cost). However, it does introduce additional complexity into the architecture, because the `Ticket Creation` service must now orchestrate the ticket process, sending information to three separate queues or RESTful endpoints.

When using this technique, calculate the aggregated bandwidth usage of each message to ensure that it's less than for a single consolidated event. In Figure 31-3, changing communication from a single event to three separate messages lowered the overall bandwidth usage from 4 MB down to 2.2 MB per ticket. However, assume each of the services needs significantly more ticket data, as shown in Table 31-5.

Table 31-5. Comparison of aggregated payload size for small and large payloads

Service	Old payload size	New payload size
`Issue Resolution`	2.2 MB	3.5 MB
`Email Notification`	1 KB	2.1 MB
`Issue Tracking`	1 KB	1.4 MB
TOTAL	2.202 MB	7 MB

Using this example, from a bandwidth perspective, it would be better to send a single event containing the 3.5 MB of attributes for the `Issue Resolution` service (assuming the payload also includes data that other services need). However, remember that stamp coupling is also about

contract management: a change to the larger contract would likely require changes in the `Email Notification` and `Issue Tracking` services, even though those services don't need the data being changed.

URL Filtering for Query Requests

For query requests, architects can use *URL filtering* to address the bandwidth usage issues associated with stamp coupling. This technique involves a service that is requesting data from another service or endpoint using a consolidated large contract. That service specifies the fields it wants through URL parameters. Using our example, suppose that, when marking the ticket as resolved, the `Ticket Resolution` service needs to get the customer name from the API gateway using a `GET /app/v1/customer` API call. Unfortunately, this call returns *all* of the customer information — another example of the Stamp Coupling antipattern. To resolve this problem, the `Ticket Resolution` service could issue the following API request instead:

```
GET /app/v1/customer?cust_id=123&fields=cust_name
```

The responding service attached to this API endpoint would require additional programming logic to interrogate the query parameters to determine which fields it should include in the response. It might also require setting all other fields to `NULL` in the case of a strict contract, or not including any other attributes in the case of a loose contract.

The URL filtering technique, while addressing the bandwidth usage issues associated with stamp coupling, does not address the contract management issues involved with using strict contracts. For example, a change to the consolidated strict contract would require a corresponding change to the requesting service, regardless of whether the fields are set to `NULL` or not.

Loose Contracts

Another technique that helps address contract management with stamp coupling is to use loose contracts — those without an associated schema — in conjunction with consumer-driven contracts. Briefly stated, a *consumer-driven contract* is one where each consumer defines their own specific contract (typically a loose one). Adding attributes to a loose contract usually doesn't break other calling services, but removing an attribute or changing one typically does if the calling services are using those attributes. In these cases, calling services using a loose contract from an owning service can create tests that the owning service runs to ensure that a change in the contract doesn't break any other services. Tools like Pact can help automate this testing process.

GraphQL

Architects can avoid this antipattern by using tools like GraphQL, which can help with bandwidth usage and, to some degree, contract management issues. With GraphQL, the owning service of a contract and the message consumers create what's called a *GraphQL spec*. This specification essentially creates separate consumer-driven contracts for each service, allowing each to request specifically what it wants without the need for URL filtering. GraphQL servers such as GraphQL.js, Apollo, and Relay do the work to resolve the GraphQL specs and deliver the corresponding payload to the calling service.

To illustrate how GraphQL works, suppose the `Customer Profile` service from our example system defines the following GraphQL spec for a customer profile named `Profile`:

```
type Profile {
    name: String ❶
    addr1: String
    addr2: String
    country: String
    ...
}
```

The `Ticket Resolution` service would then define its own specification for the customer profile information, with the same spec name (in this case `Profile`) but with only the attributes it's interested in:

```
type Profile {
    name: String ❶
}
```

❶ This is the only attribute the `Ticket Resolution` service is interested in.

When the `Ticket Resolution` service makes a request to the GraphQL server using this specification, it only receives the customer name, using less bandwidth than transferring all the customer profile information.

Key-Based Payloads

A final technique to avoid this antipattern is to leverage *key-based event payloads*, which contain only the keys of the data being inserted or deleted, or keys and specific changed attributes (old value and new value) for update actions. See Chapter 23 for a full description of this pattern and how it's used.

Trade-Offs

The following sections describe the pros and cons associated with the Stamp Coupling antipattern.

Pros

Even though stamp coupling is an antipattern, architects might decide the advantages it provides are worthwhile against the trade-offs. For relatively small payload sizes and low request volumes, the trade-offs of cost, bandwidth usage, and ease of contract maintenance are not overly

significant compared to the advantages of overall simplicity and evolvability. However, for larger payload sizes and higher message volumes, these trade-offs become significant. Architects should take both message volume and message size into account when considering whether to avoid this antipattern by creating specific contracts (and possibly separate messages) for each receiving service.

Another advantage when adopting this antipattern and sending as much data as possible in the message is that message producers don't need to be concerned about the specific data needs of the message consumers — only that those services get the information they require as part of the message payload. This means less coordination and communication between teams, avoiding the additional complexity of using consumer-driven contracts and sending information to multiple services. Instead, message producers can deliver a single consolidated event that all message consumers share.

Cons

Most antipatterns and pitfalls have serious disadvantages, which is why architects generally choose to avoid them. Contract management is one of those disadvantages. Per the definition of stamp coupling, defining extraneous attributes in a strict payload contract will (eventually) require a change to message consumers that don't need those attributes, if that part of the contract changes. This unnecessarily increases the coupling level between those services.

Another major consideration that teams should be aware of when adopting this antipattern is the amount of bandwidth used by message payloads. As demonstrated in this chapter, ignoring this antipattern can lead to a significant increase in cost, particularly for large payloads and high request volumes. This cost increase becomes even more significant in cloud-based environments.

Comparison Chart

The comparison chart in Table 31-6 shows the qualitative ratings for various capabilities when falling into and when avoiding this antipattern. In this chart, good attributes start at a five-star rating, and negative consequences reduce the five stars. Because the trade-offs associated with adopting this antipattern vary greatly depending on the size of the message payload and request volumes, we've chosen to average those in this chart. See "Influences" for a more detailed comparison chart based on these influences.

Table 31-6. Comparison chart for the Stamp Coupling antipattern

Capability	Using Antipattern	Avoiding Antipattern
IMPROVED		
Adaptability	★★★★★	★★★
Evolvability	★★★★★	★★★
Simplicity	★★★★★	★★★
DEGRADED		
Bandwidth Usage	★★	★★★★★

The following sections describe each of the capabilities found in Table 31-6.

Improved

The capabilities that are improved when applying this pattern include:

Adaptability

Passing all the data in the message payload allows services to change and adapt without having to modify any message producers, particularly when responding services suddenly require data elements in the message payload they didn't need before.

Evolvability

When the message payload contains all of the data, teams can add responding services without contract changes or changes to the corresponding message producers, making it easier to evolve the system.

Simplicity

When adopting this antipattern and sending as much data as possible in the message, message producers don't need to be concerned about the specific data needs of the message consumers — only that those services get the information they require as part of the message payload. This means less coordination and communication between teams, avoiding the additional complexity of using consumer-driven contracts and sending information to multiple services. Instead, message producers can deliver a single consolidated event that all message consumers share.

Degraded

The capabilities that are degraded when applying this pattern include:

Bandwidth Usage

As demonstrated in the prior section, network bandwidth usage can become an issue with stamp coupling, particularly for messages with large data payloads and high request volumes. For cloud-based deployments, higher bandwidth usage can also significantly raise costs associated with communication between services and systems when falling into this antipattern.

Influences

The primary influence on this antipattern is the event-driven architecture (EDA) style. Because EDA relies heavily on triggering events based on the outcome of a particular process, with multiple services reacting to those events, stamp coupling is much more prevalent here than with other architectural styles. The services reacting to an event likely have different data needs, so the service triggering the event must account for all data the services need.

The most common technique to avoid stamp coupling in EDA is to send specific *messages* to each receiving service rather than a single *event* (see "Events Versus Messages" in Chapter 37 to understand the difference between a message and event). However, this technique, as illustrated in Figure 31-3, increases coupling between the service triggering the event and the services reacting to the event. This is because the triggering service must have detailed knowledge about which services need to react to the triggered event and what their specific data needs are. In addition, the triggering service must act as an orchestrator (see Chapter 27) to coordinate and send information to multiple event channels.

By adopting this antipattern and sending only a single derived event containing an aggregate of all the information needed by the responding services, architects can significantly reduce service coupling and contract coordination, and keep within the tenets of the event-driven architectural style. For this reason, we strongly recommend paying particular attention to the trade-offs involved with this antipattern when using event-driven architecture, and performing stricter measurements (see "Am I in This Antipattern?") to determine whether to adopt it or not.

Another factor that influences this antipattern and its corresponding star ratings is the size of the payload and the request volume. As shown in Table 31-7, those capabilities that are improved don't change, but the one that is degraded — bandwidth usage — does. Notice for small payloads with low request volumes the bandwidth usage isn't overly significant.

However, as the payload size and request volume increase, bandwidth usage becomes a rather significant reason to avoid this antipattern.

Table 31-7. Comparison chart for the Stamp Coupling antipattern, based on influences

Capability	General Ratings	Small Payloads and Low Volumes	Large Payloads and High Volumes
IMPROVED			
Adaptability	★★★★★	★★★★★	★★★★★
Evolvability	★★★★★	★★★★★	★★★★★
Simplicity	★★★★★	★★★★★	★★★★★
DEGRADED			
Bandwidth Usage	★★★	★★★★	★

Am I in This Antipattern?

Architects can determine whether they're encountering the Stamp Coupling antipattern by paying attention to the payload size of messages and the volume of messages being sent, as well as by tracking the rate of change of the payload contracts. The following sections outline these governance techniques.

Payload Size and Message Volumes

Continually calculating and tracking the size of a message or event payload is an architect's first defense against falling victim to this antipattern. For large payloads in particular, pay attention to whether the responding service actually needs all of the data in the message payload and corresponding contract. It's important to continuously or periodically track the size of each message or request sent, as changes to data requirements may affect message size.

Architects can have the message producer periodically calculate the payload size and request volumes, and stream those metrics for observability using a continuous or triggered fitness function. Another option is to use the Wire Tap pattern periodically to intercept the request payload and calculate and stream the same metrics.

Rate of Contract Change

Since contract management is an issue associated with this antipattern, architects can measure and track a contract's rate of change and its corresponding impact on other services. The higher the change rate, the more of an issue this antipattern is, particularly if the contract change affects other services.

Related Patterns, Antipatterns, and Pitfalls

The Stamp Coupling antipattern is related to any pattern that involves communication and message payload contracts. As such, architects should pay close attention to the following communication-related patterns:

Contextual Events pattern (see Chapter 12)

> *Contextual events* are derived events that describe a specific outcome of a service processing an incoming event. Architects must decide on the type of payload contract contextual events use.

Data-Based Event Contract pattern (see Chapter 14)

This pattern describes sending all of the available data in an event payload to responding services. Architects should pay particular attention to ensure they're not encountering the Stamp Coupling antipattern when adopting this pattern.

Extensible Events pattern (see Chapter 18)

The Extensible Events pattern describes when a service triggers a derived event based on its processing outcome, even though no other services are responding to that event. It's a way of promoting architectural evolvability and overall agility. Since it's difficult to determine whether services will need this event in the future, architects should apply caution when sending *all* the data available in the extensible event due to increased bandwidth and cost considerations.

Key-Based Event Contract pattern (see Chapter 23)

A Key-Based Event Contract pattern is one where only the keys of the processed data are sent in the event payload. Architects using this pattern usually don't have to worry about Stamp Coupling, since only a minimal amount of data is passed in the event payload.

Chapter 32. Stovepipe Architecture Antipattern

In the old days, people kept warm and cooked using wood stoves. Because they required venting, people built *stovepipes* out of metal tubes, joints, and connectors to route smoke outdoors, often resulting in haphazard (and sometimes amusing) stovepipe configurations, like the one in Figure 32-1.

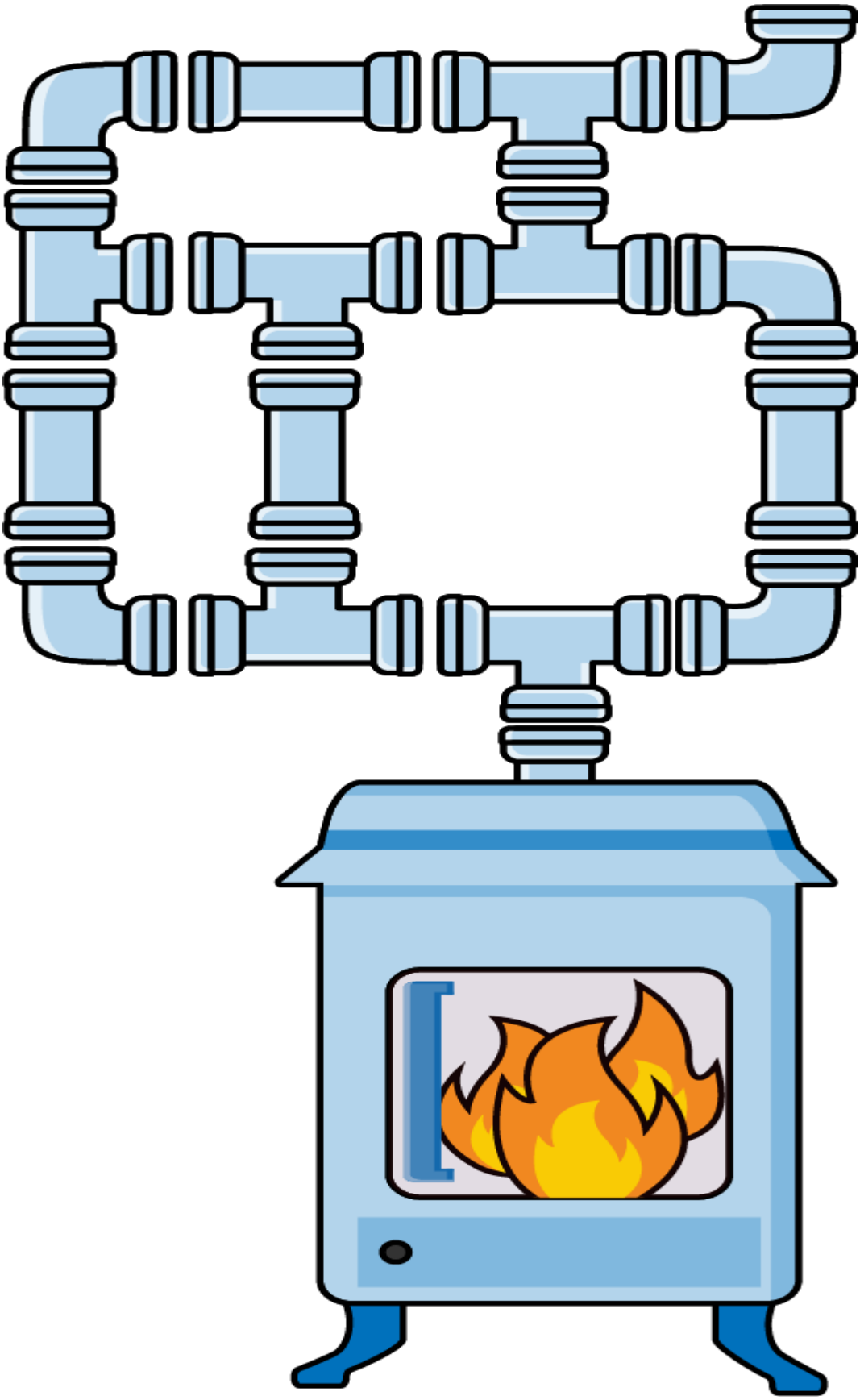


Figure 32-1. An example of an ad hoc and ill-fitting collection of stovepipe parts

Software architecture can sometimes resemble these ad hoc, bolt-on stovepipes, resulting in what we call the Stovepipe Architecture antipattern. This chapter describes how to avoid this antipattern by applying effective techniques to iterate on an architecture.

Table 32-1 summarizes the metadata for the Stovepipe Architecture antipattern. Because this chapter deals with an antipattern, not a pattern, the Improved metadata category refers to the capabilities that are improved when adopting this antipattern, and the Degraded metadata category refers to the capabilities that are degraded when adopting this antipattern.

Table 32-1. Metadata for the Stovepipe Architecture antipattern

CATEGORY	METADATA
CONTEXT:	Architectural Styles, Iterative Architecture, Monolithic Architectures
IMPROVED:	Cost, Initial Time to Market
DEGRADED:	Adaptability, Agility, Availability, Deployability, Domain Alignment, Elasticity, Evolvability, Fault Tolerance, Maintainability, Responsiveness, Scalability, Testability

How Do I Iterate on an Architecture?

All architectures change — sometimes significantly — through their lifespans to accommodate new and changed functionalities. However, due to time and budget constraints, architects often take shortcuts by simply "bolting on" new components or services to the architecture, creating what is commonly referred to as a "stovepipe" architecture. The Stovepipe

Architecture antipattern describes an ad hoc collection of semi-related ideas, concepts, structures, and components, comprising an unreliable, brittle architecture that's hard to change.

This chapter describes techniques for properly iterating on greenfield or brownfield architectures, thus avoiding this common antipattern.

Why Is This an Antipattern?

When creating a new system or evolving an existing one, most teams focus solely on its behavior. This is particularly true for systems largely developed through vibe coding or agentic AI. However, the behavior of a system is different from its capabilities. The system's *capabilities* are its architectural capabilities, such as scalability, responsiveness, performance, availability, maintainability, and so on — what some architects call *nonfunctional requirements*.

Say a team wants to design a truck (lorry), but they're new to this line of work — no one on the team has ever designed a truck before. They decide to design the truck through the practice of *emergent design*, which teaches that the team should start with designing something simple, such as a roller skate. They should then keep evolving that design as they learn more about the problem, until they finally arrive at the truck (see Figure 32-2).

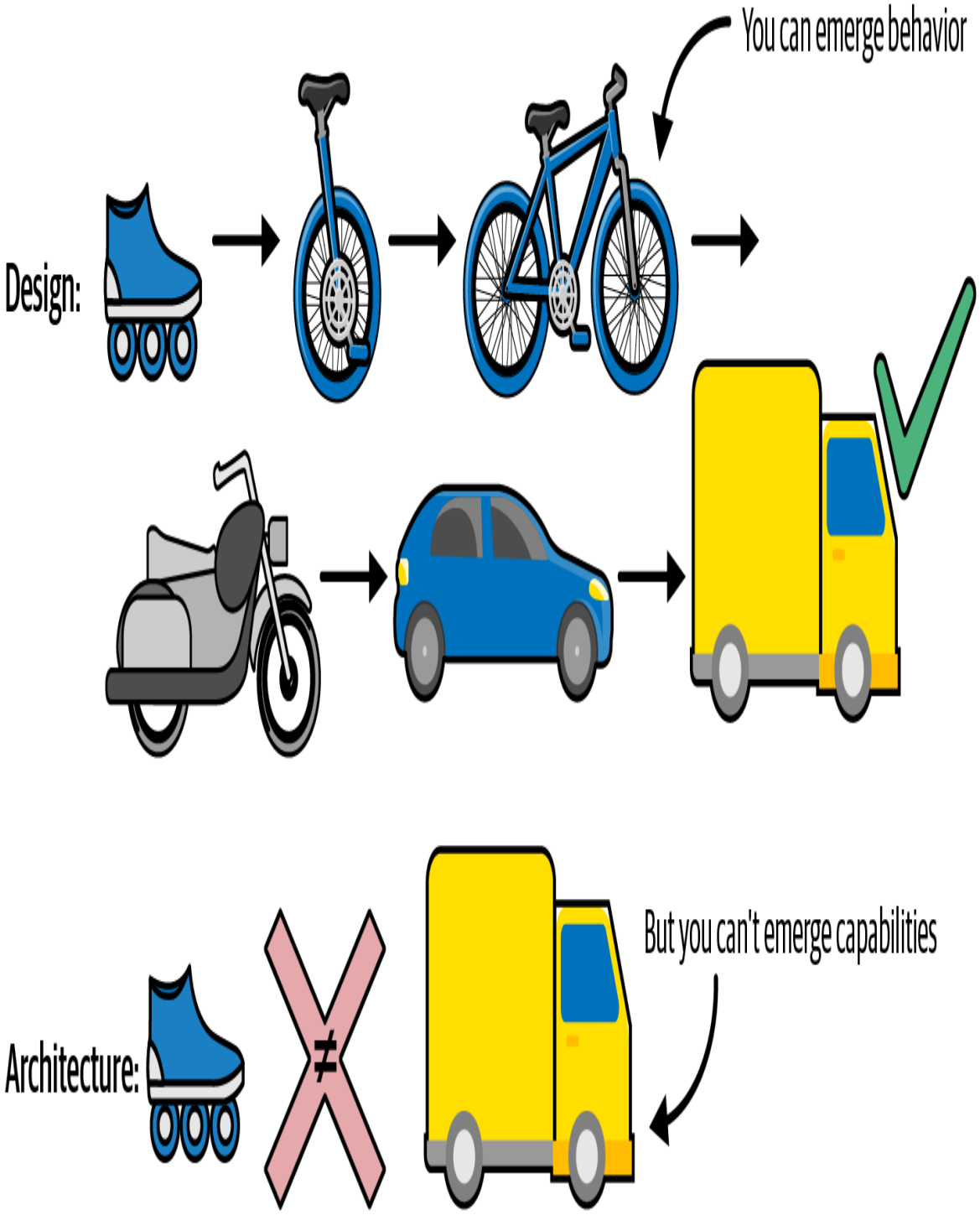


Figure 32-2. Emergent design is about starting out with a simple solution to learn about the problem domain, then iterating to eventually arrive at the goal

Emergent design is a proven practice that software development teams worldwide use to develop complex systems. Remember, however, that while design is concerned about a system's *behavior*, architecture is

concerned about its *capabilities*. Unfortunately, capabilities don't just emerge through iteration like behaviors do; they require planning. A building contractor can't construct a 10-story office building by starting out with a simple doghouse, because the two structures' *capabilities* are vastly different. For the same reason, architects can't start with a "roller skate" architecture and evolve it into a truck.

John Gall, author of the book *Systemantics* (Quadrangle, 1977), famously coined what is known as Gall's Law:

> *A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system.*

When applying Gall's Law to software architecture and capabilities, it's easy to see how Stovepipe Architecture quickly becomes an antipattern.

Let's say a team is developing a new system that must support 200,000 concurrent users with an average response time of 1,200 ms. While they can (and should) evolve its design, Gall's Law suggests that teams can't just bolt on the scalability and responsiveness requirements once the behavior they want has emerged (or even *during* emergent design). Instead, they should start with a simple system that can meet their scalability and responsiveness requirements, then increase its behavioral complexity as they continuously measure, track, and verify that the system can still scale and respond as required.

Simple systems, such as the youth-soccer-team registration website described in the related Architecture by Implication antipattern in Chapter 4, don't require an architecture, so the Stovepipe Architecture antipattern is rarely an issue. Cost, effort, and time to market are more important in such systems than more complex operational capabilities like scalability, availability, and fault tolerance. It's possible to bolt additional behavior onto these simple systems without invoking the negative trade-offs of this antipattern.

However, as also illustrated in Architecture by Implication, once the simple system grows and becomes more widely used, those more complex operational and structural capabilities become more important — and attempting to simply bolt them on would result in a Stovepipe Architecture. Consequently, according to Gall's Law, the team would be better off starting over with a simple working system that satisfies the architectural capabilities of the system.

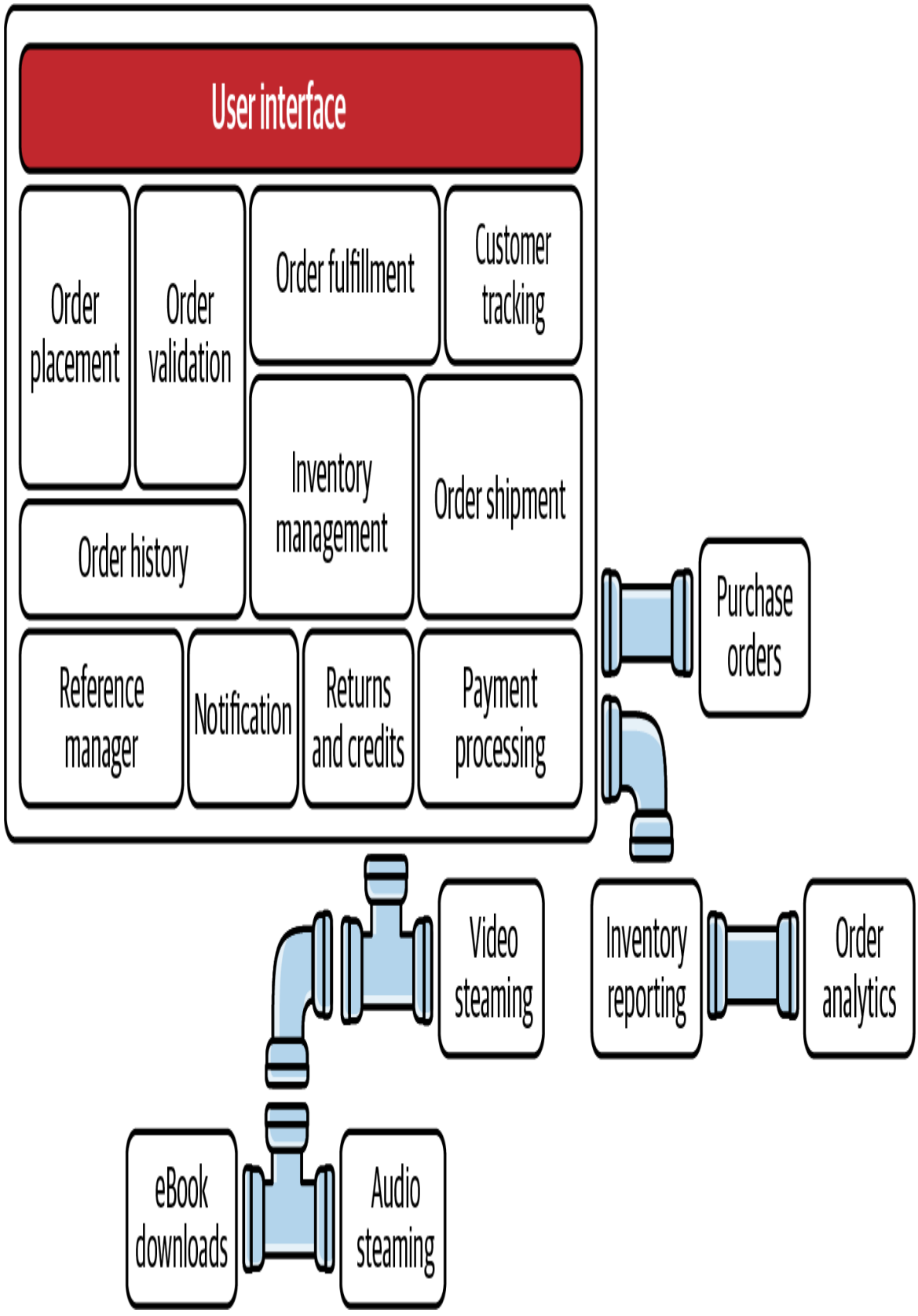

User interface
Order placement
Order validation
Order fulfillment
Customer tracking
Inventory management
Order shipment
Order history
Reference manager
Notification
Returns and credits
Payment processing
Purchase orders
Video steaming
Inventory reporting
Order analytics
eBook downloads
Audio steaming

Figure 32-3. A monolithic system can quickly become a Stovepipe Architecture as new behavior is haphazardly bolted on

To illustrate how a system can quickly become an ad hoc collection of ill-fitting parts, consider the ordering system illustrated in Figure 32-3. The initial monolithic architecture shown here is well structured to support the necessary architectural capabilities.

However, if developers introduce new behavior by simply bolting services onto the existing monolithic system (depicted in the diagram as boxes connected by stovepipes), the architecture quickly loses its clarity and definition, becoming a Stovepipe Architecture.

The Stovepipe Architecture antipattern can occur with distributed architectures as well. Consider the microservices architecture illustrated in Figure 32-4. In this diagram, a single user interface accesses six services (A through F) through separate screens within the user interface. Each microservice in this system is independent of the others, owns its own data in a bounded context, and supports all necessary architectural capabilities.

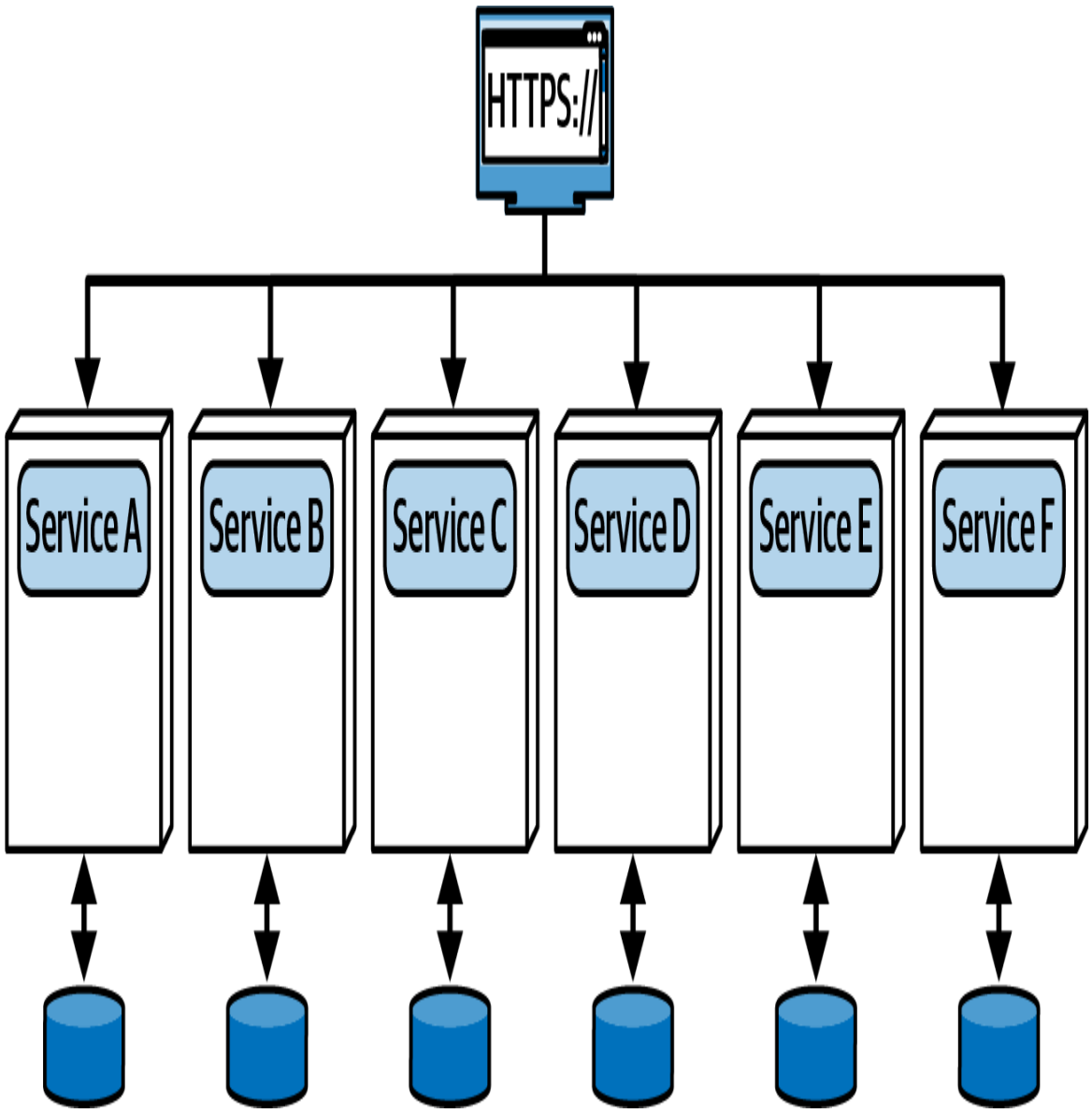


Figure 32-4. A well-structured microservices distributed architecture with independent services and data

The team decides to restructure the UI, consolidating those six separate screens into two screens. Now the UI only makes calls to `Service A` and `Service F`. However, the UI still needs the other services' functionalities, so those services must use interservice communication, as shown in Figure 32-5.

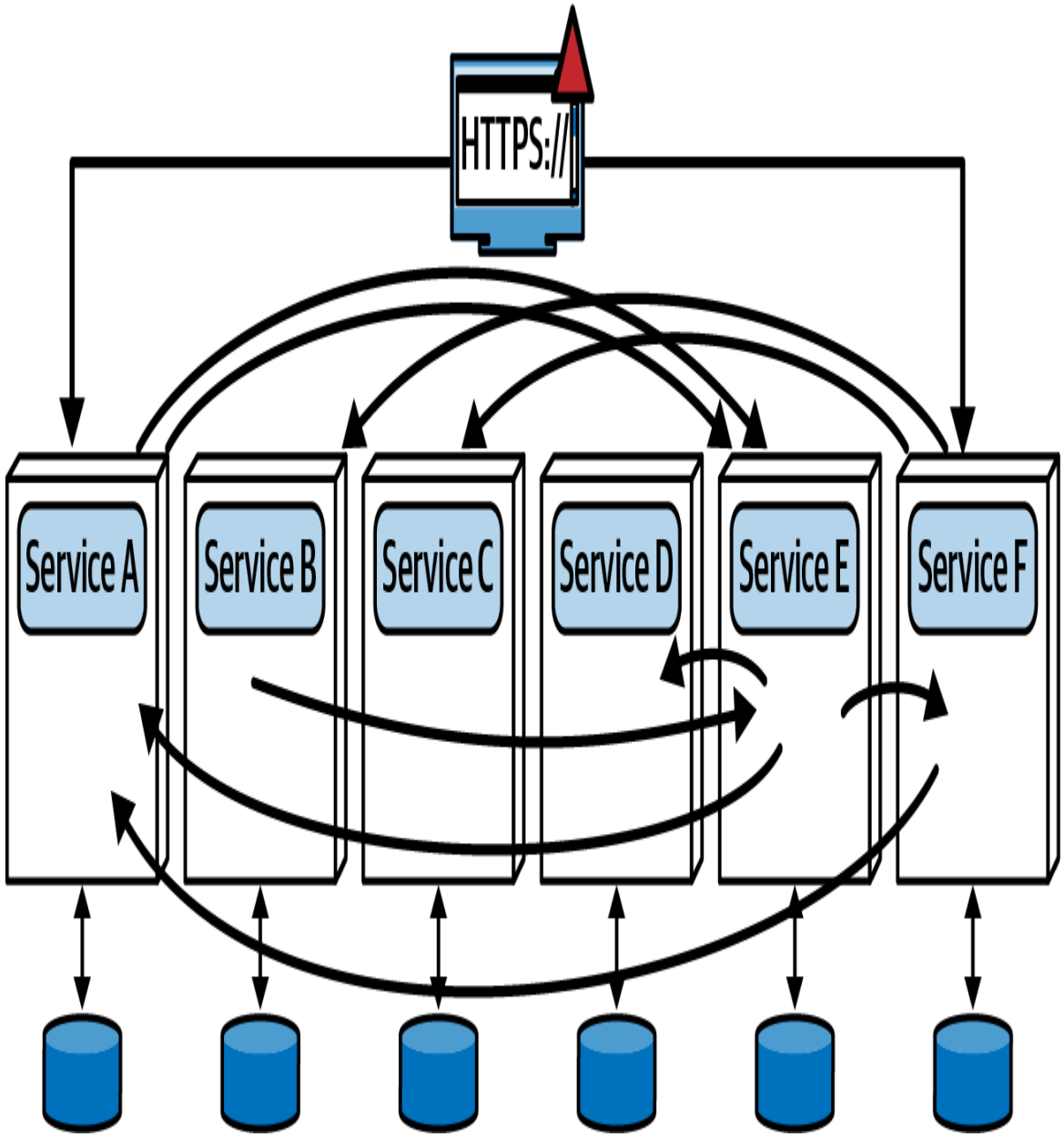


Figure 32-5. A change in the UI now requires services to communicate with each other, resulting in a Stovepipe Architecture

Alternatively, teams can follow the example illustrated in Figure 32-5 and use Orchestration (see Chapter 27) to coordinate the services. For example, assume a UI has two separate screens, with each accessing a separate microservice through two API endpoints, as shown in Figure 32-6. The team changes the UI to consolidate those screens into a single screen accessing a single API endpoint. Then they bolt on a microservices orchestrator to coordinate the two microservices without any thought for

how this will affect the architectures — thus falling into the Stovepipe Architecture antipattern.

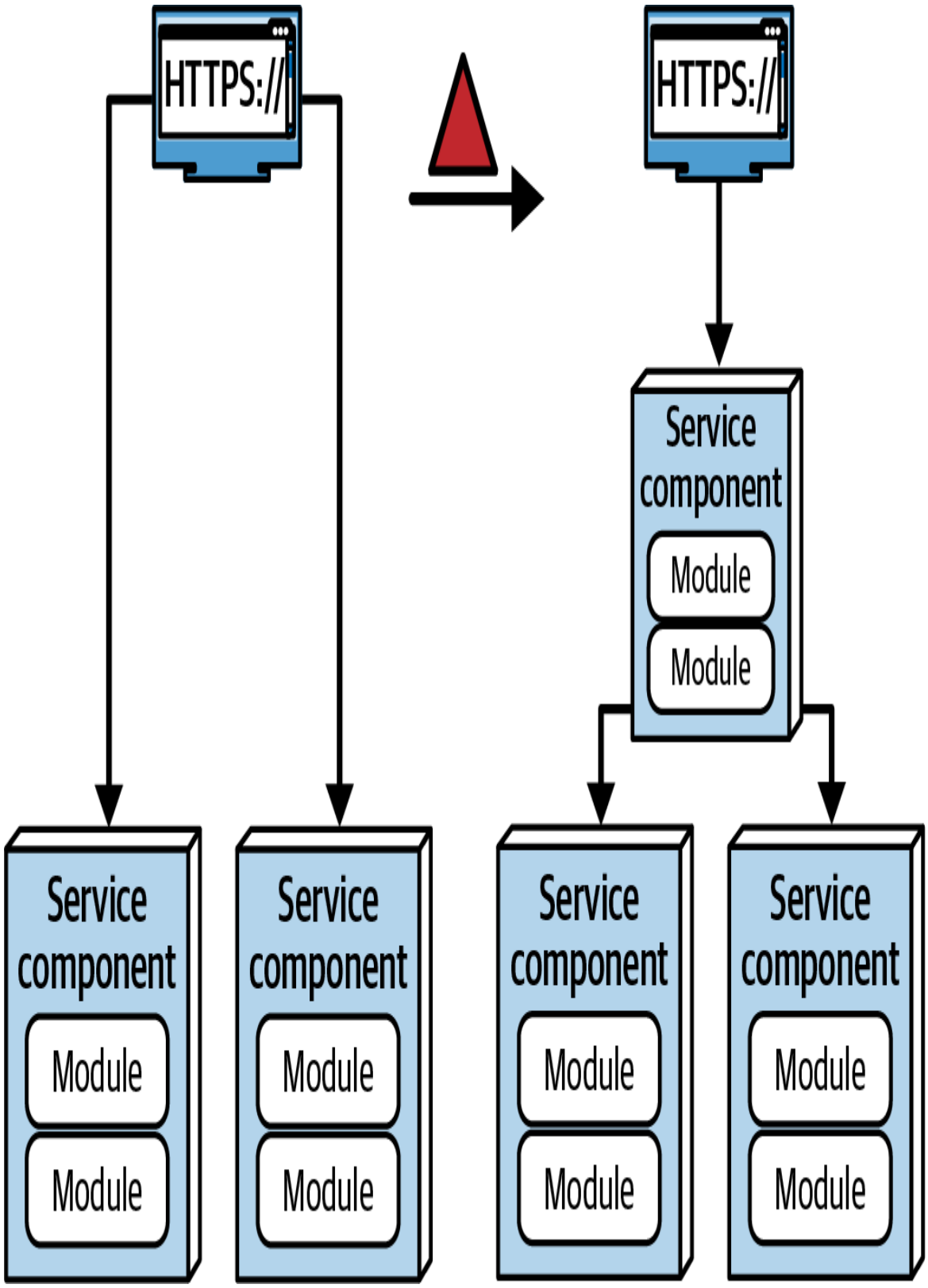


Figure 32-6. A change in the UI now requires services to be orchestrated, resulting in a Stovepipe Architecture

These scenarios, as well as a host of others described in the following sections, can all lead to this antipattern. The next section describes techniques for avoiding it by using iterative architecture to maintain a well-structured architecture as teams change the system's behavior.

How to Avoid This Antipattern

Architects can avoid the Stovepipe Architecture antipattern by leveraging *iterative architecture*: a set of techniques to *evolve* an architecture rather than simply bolting on additional components or services, thus preserving its capabilities as it changes and grows.

The techniques for iterating on an architecture are slightly different for greenfield and brownfield systems, as well as for iterating on a logical architecture versus a physical one. So the following section on greenfield systems focuses on iterating on a system's logical architecture (its components and their interactions), while the brownfield system section focuses on the physical architecture (services, UIs, and so on).

Greenfield Systems

Greenfield systems are systems built from the ground up, either as new systems or migrated from legacy systems. Typically, architects start by defining a *logical architecture*, which shows the system's architectural components — its building blocks — and how they interact. An architectural component is something the system *does* — its behavior — and is usually implemented through class files.

Each architectural component has a distinct and well-defined set of operations, role, and responsibility in the system. For example, in a logical architecture describing a typical ordering system, the `Payment` component is responsible for applying payment for a customer order on checkout. It consists of well-defined endpoints that allow a customer to pay for their order through a variety of means: a credit card, gift card, rewards points,

and so on. Developers then write source code to implement this architectural component.

Generally, a system's logical architecture is independent of its physical architecture. Creating a logical architecture is about what the system does and how its functional parts interact, rather than its physical structure of services, databases, user interfaces, and so on. Architects must properly identify components and decide where certain behaviors should reside in a logical architecture by continuously identifying and restructuring logical components. This works best as an iterative process, as illustrated in Figure 32-7.

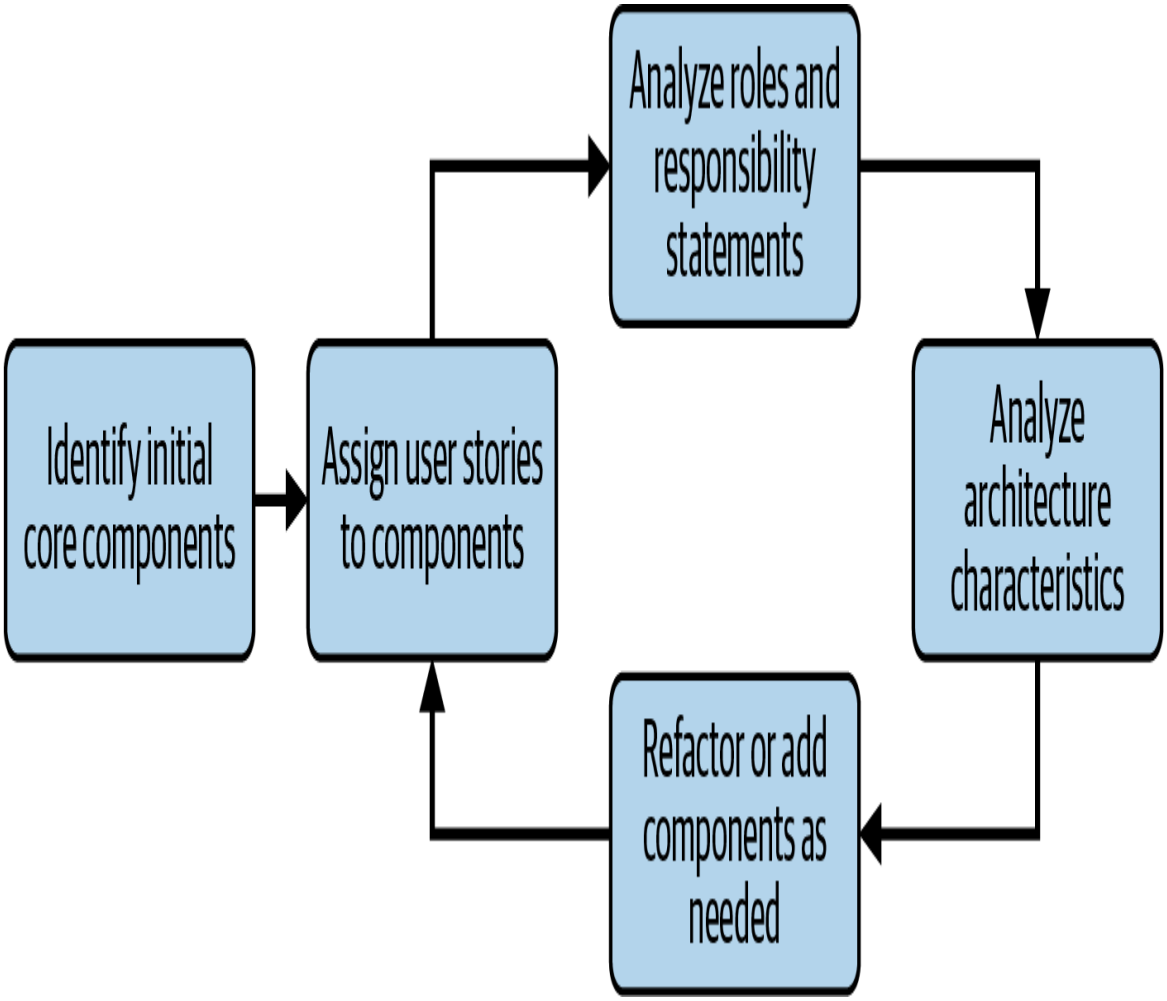


Figure 32-7. The component identification and refactoring cycle

When an architect develops a logical architecture, their first step is to identify its initial core components and assign user stories or requirements

to them. After this, they analyze the component's roles and responsibilities to make sure that the user stories or requirements assigned to it make sense. Then the architect looks at the architectural capabilities the system must support and determines whether they need to break apart or combine any components to support them. Finally, the architect might further refine the components based on this analysis. This feedback loop can help architects avoid creating a Stovepipe Architecture.

Consider the existing components of the logical architecture shown in Figure 32-8. When a customer makes a purchase, the `Order Placement` component interacts with the `Inventory Management` component to adjust the inventory for the items purchased, and with the `Order Fulfillment` component, which handles packing the item and preparing it for shipping. The `Order Fulfillment` component then interacts with the `Order Shipping` component to ship the order to the customer.

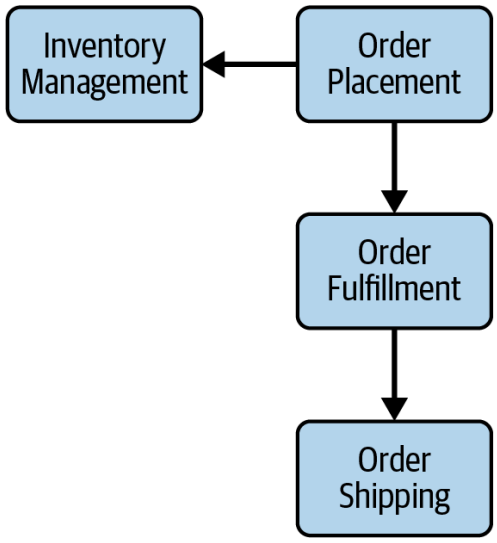


Figure 32-8. A logical architecture diagram showing the basic components of a typical ordering system

Now suppose the business team wants to add some behavior to email the customer every time the state of the order changes: when the order is successfully placed, when it has been packed and prepared, and when it ships. Without considering the architecture, the development team (with or without the help of an AI agent) decides to code this customer notification behavior into the existing `Order Placement` component, as illustrated

in Figure 32-9. However, that solution requires the `Order Fulfillment` and `Order Shipping` components to each make a call to the `Order Placement` component to send the notification email. Not only does this increase component coupling, it also introduces cyclical dependencies into the architecture and places an additional responsibility on the `Order Placement` component (notifying a customer). That responsibility doesn't match the `Order Placement` component's primary role and responsibility, which are to validate the order and perform the customer checkout behavior.

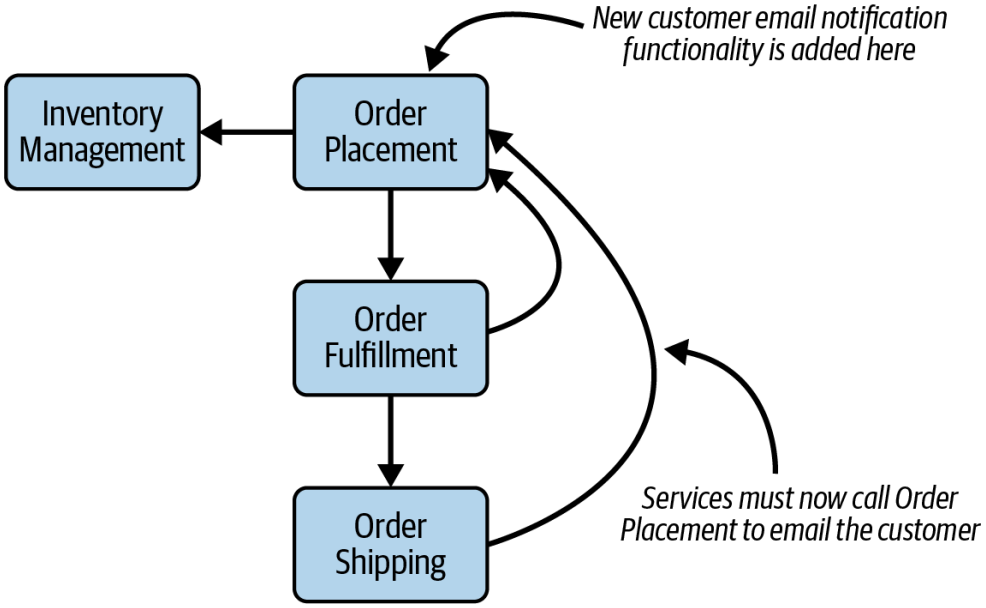


Figure 32-9. The customer notification behavior is located in the Order Placement component, resulting in increased coupling and overloaded components

Placing behavior in an unrelated component can lead to a Stovepipe Architecture, since the bolted-on behavior (in this case, notifying the customer) is added to an existing component haphazardly, causing that component to become too big as its role and responsibilities increase. This practice also overloads components as they start to become dumping grounds for added functionalities. The result is an unstructured, ad hoc architecture. A good example of this is when developers throw all sorts of unrelated utility functions (such as string manipulation, date transformation, calculations, and so on) into a `Utility` class file, which is typically imported into most classes anyway.

To avoid the Stovepipe Architecture antipattern, architects should first analyze a proposed change to determine its possible impact on the architecture and decide where the new behavior should go. Using the Role and Responsibility Model (discussed in detail later in this chapter) and the flowchart shown in Figure 32-7, an architect can quickly determine that the `Order Placement`, `Order Fulfillment`, and `Order Shipping` components, while they all need the customer notification behavior, shouldn't all be responsible for emailing the customer. Instead, the architect can add a component called `Customer Notification` that's responsible for emailing customers when their order's status changes. This architecture, shown in Figure 32-10, is an example of how architectures naturally evolve to accommodate new behavior.

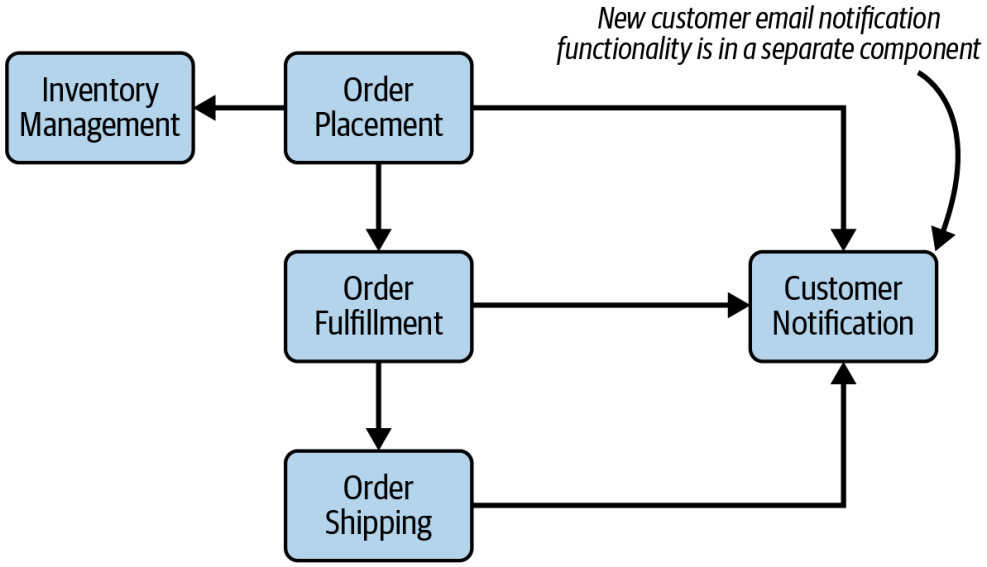


Figure 32-10. Creating a component for the new behavior decouples the system and preserves component responsibilities

By iterating on the architecture properly, architects can preserve the primary roles and responsibilities of existing components, reduce the coupling between them, and avoid this antipattern.

Brownfield Systems

A *brownfield system* is an existing system that will likely undergo change. The techniques of iterating on a logical architecture that we described in the

previous section apply to brownfield systems as well; this section focuses on iterating on the *physical* architecture. Unlike a logical architecture, a physical architecture consists of artifacts such as services, message brokers, gateways, user interfaces, databases, and so on. Just as with logical components, haphazardly adding or expanding services with new behavior to a physical architecture can lead to the Stovepipe Architecture antipattern.

This section outlines three common scenarios where this can happen and describes iterative architecture techniques to help avoid this antipattern.

When changes require service orchestration

In the previous example (Figure 32-6), the UI had two screens, each invoking a separate service. The development team then consolidated the two screens into one and added an orchestrator service to call each service. Because each service's behavior had previously been separately invoked by individual screens and corresponding API endpoints, maintaining two separate services made sense. However, because of the UI change, it no longer makes sense to keep these services separate. They're no longer invoked *individually*, but as a collective whole.

Adding unnecessary orchestration adds complexity to the architecture and, if those services are updating information, can also degrade data integrity and consistency. Because the services are coordinated as an atomic unit, the orchestration service must coordinate and manage a distributed transaction between them. If the first service commits its updates to the database and the second service fails, the orchestrator would have to issue a compensating update to reverse the updates applied by the first service. Iterating on the architecture makes room for the architect to realize early on that this orchestration is unnecessary and instead consolidate the individual services, as illustrated in Figure 32-11.

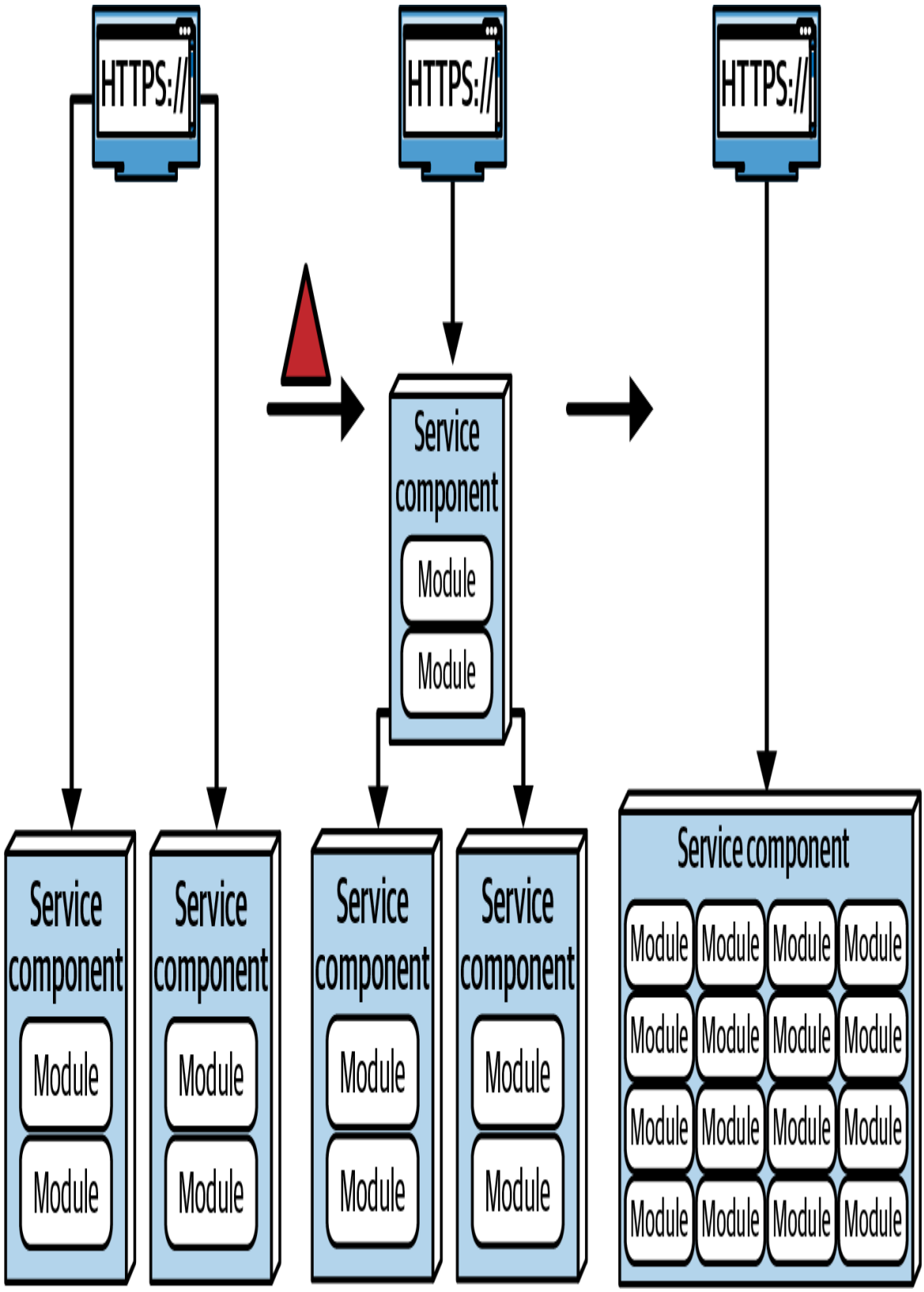


Figure 32-11. Removing unnecessary orchestration and consolidating the services improves the architecture

By consolidating the services instead of adding an orchestrator, the architect not only simplifies the architecture but also maintains data atomicity for single requests without having to use distributed transactions. Architects considering adding orchestration to coordinate services should check first to see if it makes more sense to consolidate the services instead. (See the Grains of Sand pitfall in Chapter 21 for more information about service granularity.)

When changes involve new data requirements

Adding or changing behavior often means that services require additional data they didn't need before. For monolithic databases, and when changes are contained within the services that access a single domain database, this typically isn't a problem. However, it is a problem for microservices that use dedicated data and for services that require data from a different domain database. Take, for example, Service A and Service B, the two independent microservices on the left side of the diagram in Figure 32-12. Each service owns its own data in a well-formed bounded context.

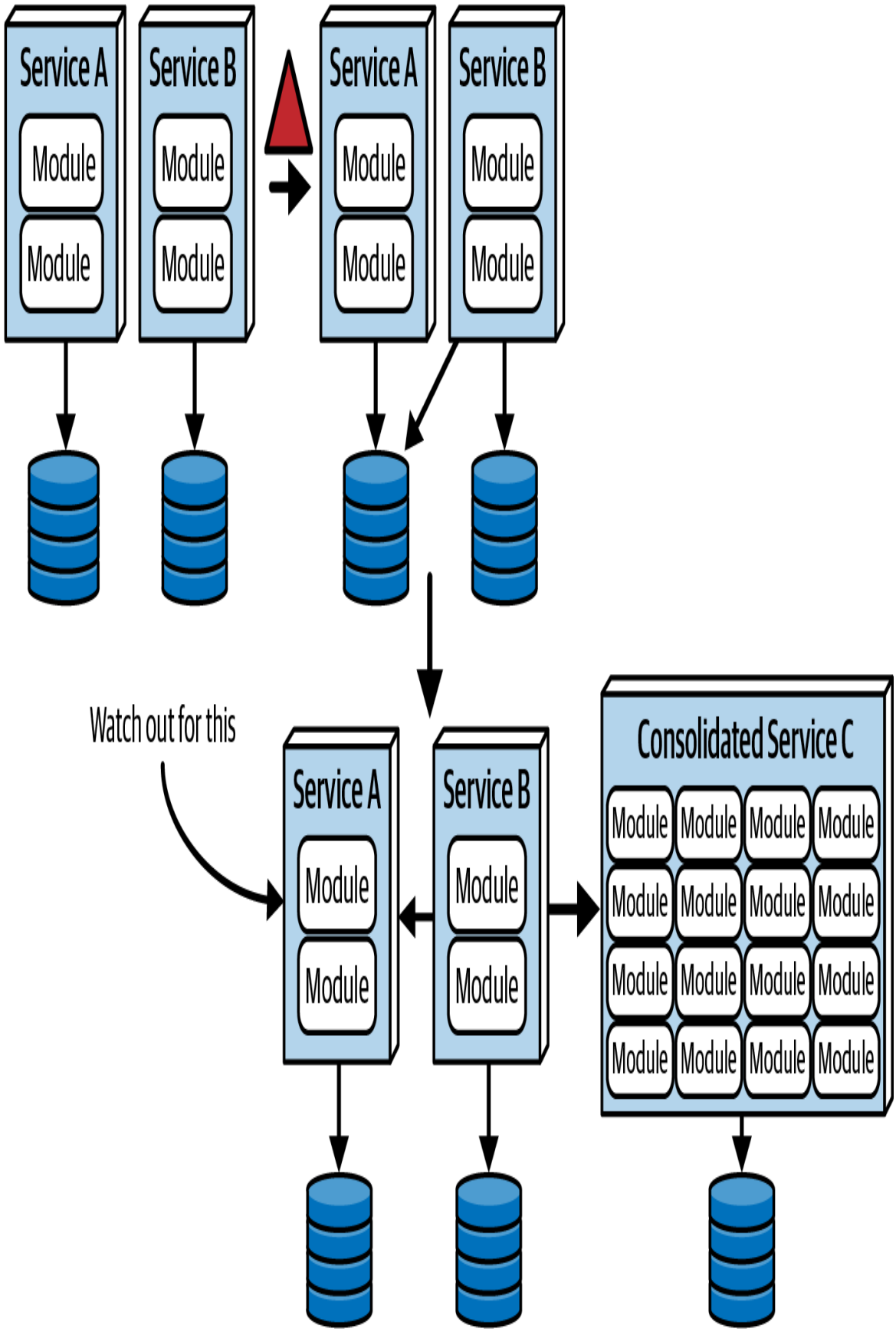

Service A
Module
Module
Service B
Module
Module
Service A
Module
Module
Service B
Module
Module
Watch out for this
Service A
Module
Module
Service B
Module
Module
Consolidated Service C
Module Module Module Module
Module Module Module Module
Module Module Module Module
Module Module Module Module

Figure 32-12. New behavior requires Service B to access data from Service A; iterating might consolidate these services

The team implements new behavior in Service B. Now it requires data from Service A, as shown in the top middle of the diagram. However, since the data associated with Service A is in a bounded context, Service B can't simply connect to the database and query the data it needs — it must *ask* Service A for the data (see the bottom middle of the diagram). To gain access to that data, Service B can make a synchronous RESTful call, use in-memory replicated caching, or even replicate the data in its own database. Regardless of the method, Service B is now coupled to — and hence dependent on — Service A. An occasional coupling due to data needs might be OK for some services, but too much of this creates a big ball of distributed mud in the form of a Stovepipe Architecture. To avoid this antipattern, architects can consolidate the two services into a single service, as shown at the right of the diagram, to improve scalability, responsiveness, and fault tolerance.

When adding behavior to existing services

A third way to produce a Stovepipe Architecture happens when a development team adds new behavior to an existing service, much like we described earlier in "Greenfield Systems". However, now we can discuss the implications for the physical architecture, particularly the services.

It's much easier for the team to bolt new behavior onto an existing service than to create a service for the new behavior. This practice can quickly create a Stovepipe Architecture — and here's why. Consider the example illustrated in Figure 32-13 where the team keeps just bolting more and more new behavior onto Service A. Over time, Service A grows as it takes on more and more responsibility. It becomes less maintainable, its testing scope and deployment risk increase, and its fault tolerance and elasticity decrease.

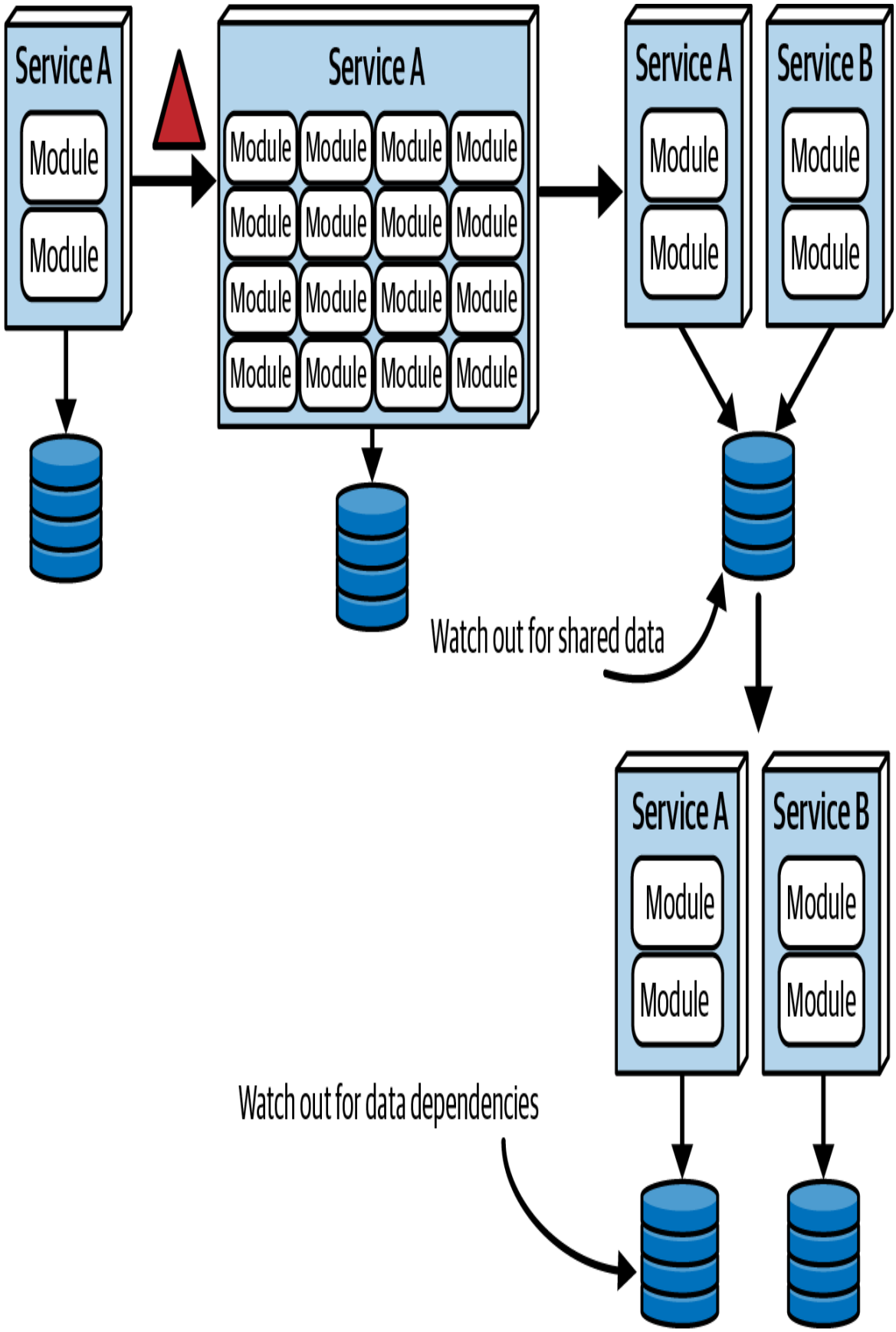

Service A
Module
Module
Service A
Module Module Module Module
Module Module Module Module
Module Module Module Module
Module Module Module Module
Service A
Module
Module
Service B
Module
Module
Watch out for shared data
Service A
Module
Module
Service B
Module
Module
Watch out for data dependencies

Figure 32-13. New behavior can increase a service's size, degrading its architectural capabilities

Iterating on the architecture when adding behavior is a good way to make sure no important architectural capabilities are degraded. If they are, the architect can break the service apart, as illustrated at the top right of Figure 32-13. However, each part of the split service will still share the same database (or database schema). This might be OK in some situations, but it does create a broader bounded context. This means that if the data structure changes, more services will need to change. It also might degrade fault tolerance and scalability in the database. A further iteration could be to break apart the data, too, as illustrated at the bottom right of the diagram. However, this should only be done if the data needs of each service are solely contained within their respective individual bounded contexts.

Repairing a monolithic stovepipe architecture

Many teams, upon learning about this antipattern, realize they have unknowingly adopted it. Fortunately, teams have a number of ways to repair a Stovepipe Architecture.

This antipattern often results from developers not paying attention to the system's internal structure, particularly in monolithic systems and larger services (such as the domain services typically found in service-based architectures). In such cases, we recommend the following refactorings, which we describe in Chapter 5 of our book *Software Architecture: The Hard Parts*:

Identify and size components

> Because services and monolithic systems are built from components, it's critical to not only identify the components within them, but to properly size them as well. This pattern is used to identify components that are either too big (doing too much) or too small (not doing enough). Components that are too large relative to other components are generally more coupled to other components, are harder to break into separate services, and lead to a less modular architecture.

Gather common and shared components

Finding common code is mostly a manual process, but some automation can be used to assist in this effort. Finding and isolating common and shared code helps turn that code from implicit coupling to explicit coupling.

Flatten components

This critical refactoring pattern basically moves all of the source code to the leaf nodes of the system's directory or namespace structure, thus establishing specific definable components in the system or service. Architects can now move the shared code identified in the prior pattern to shared components, creating explicit coupling between behavioral components and the shared components.

Determine component dependencies

The purpose of this pattern is to analyze the incoming and outgoing dependencies (coupling) between *components* and to untangle those dependencies.

Create component domains

This pattern is about identifying component domains — groups of components that perform some sort of related behavior in the system, such as order shipping logic, order fulfillment logic, payment logic, and so on.

By utilizing these structural refactoring patterns, architects can refactor and improve the structure of a system or service without affecting its behavior, and transform a Stovepipe Architecture into a well-structured one with improved maintainability, testability, deployability, reliability, adaptability, and evolvability capabilities.

Repairing a distributed Stovepipe Architecture

In distributed architectures, while the refactoring patterns we just described also apply, architects can use several other techniques to transform a physical Stovepipe Architecture, particularly with regard to services.

The first technique is to measure and analyze the number of interservice calls and refactor to reduce them. Interservice calls — especially synchronous ones — can significantly degrade most of the system's process and operational capabilities. Consolidating services or data can help.

Another technique is to apply the traditional Role and Responsibility Model, one of the core parts of the CRC object-oriented methodology, originally proposed by Ward Cunningham and Kent Beck. This model states that each component or service should have a specific and distinct role and responsibility. It's a great way to find services or components that are doing too much and are hence too big. A similar approach is the Single-Responsibility Principle (SRP), coined by Robert C. Martin, which states that a component or service should be responsible to one (and only one) actor. Applying these two concepts, architects can find opportunities to iterate and improve architectural capabilities.

Iterative Architecture and Residuality Theory

The iterative architecture techniques we describe in this section help teams avoid haphazardly bolting on added behavior by forcing them to pay attention to architectural concerns when applying changes.

Residuality Theory, created by software architect Barry O'Reilly, uses aspects of complexity theory and predictive analysis to minimize how much an architecture is changed on the basis of unknown future requirements. The less the architecture needs to change, the less likely its architects will end up adopting the Stovepipe Architecture antipattern.

Residuality Theory, greatly simplified, holds that:

1. The unordered systems representing the business domain contain unknown situations, called *stressors*, that will impose new requirements, forcing the architecture to change.

2. Stressors are associated with various states within the system, called *attractors*.
3. After an architect changes the architecture to address a stressor, what's left over in the architecture is called *residue*.

Residuality Theory states that if architects change the architecture to accommodate *some* potential stressors, other unknown stressors associated with the same attractors (and hence the same residue) typically won't require additional changes to the architecture.

Stressor analysis is a collaborative exercise used in residuality theory. Teams collectively brainstorm possible events that could affect the system — technical issues, but also business, environmental, and social situations, however ridiculous. Then they determine what the stressors they identified would break in the architecture and propose changes to address those potential breaks. Those proposed changes form *residues*. The more potential stressors they identify, the more residues they will come up with.

Architects often complain that Residuality Theory and stressor analysis can lead to overengineering, where teams implement residues solely based on "what-if" scenarios rather than actual requirements (the "known knowns"). The following are some techniques to avoid overengineering when applying Residuality Theory and stressor analysis:

- Record each stressor the team identifies during stressor analysis in a spreadsheet, and mark down which residue addresses the stressor. Then, for each stressor, try to gain team consensus in terms of the probability that the stressor might actually happen. We recommend a scale of *unlikely*, *maybe*, and *likely*.
- Look for residues that impact a critical or core area of the system and calculate how many stressors were associated with that residue. Then determine the overall collective probability of those stressors. If the stressors targeting a critical area of the architecture collectively have a medium-to-high level of probability, that residue would be a good candidate for the team to implement.

- Look for residues with a high number of associated stressors (core or not), and calculate the overall collective probability of those stressors. If a large number of stressors identified by the team target a particular area of the architecture, likely the team has identified an area that will undergo lots of change, making that residue another implementation candidate.

For more information about stressor analysis (and Residuality Theory in general), refer to Mark's video on this topic.

Trade-Offs

Antipatterns differ from pitfalls in that antipatterns *do* have trade-offs: some positive consequences can come from adopting them, whereas pitfalls have no upsides. The following sections outline the pros and cons of adopting this antipattern.

Pros

Time and budget pressures are usually what drive teams toward this antipattern. It's far cheaper and faster to ignore architectural concerns and just focus on adding behavior as quickly as possible. This is particularly true with practices like vibe coding. However, these benefits can be short-lived as maintenance and testing become harder.

Cons

A Stovepipe Architecture is unlikely to be able to support any operational or process-related capabilities as the focus is primarily on adding behavior wherever possible. For small, simple systems, the trade-offs of cost and time might outweigh the need for architectural capabilities, whereas for larger systems, the need for architectural capabilities typically far outweighs the cost and time considerations.

Teams adopting this antipattern will also find the system's overall reliability starts to degrade. Increased coupling, improper service responsibilities, and an unstructured architecture all lead to unexpected errors, particularly when modifying behavior. When a change in one area of the system breaks other, unrelated parts of the system, it's often a symptom of this antipattern.

Comparison Chart

The comparison chart in Table 32-2 shows how the qualitative ratings for various factors change when adopting or avoiding this antipattern. Architects should consider the system's context and capabilities (is it a small, simple website or a critical, complex business system?), as well as what critical architectural capabilities they need, when deciding whether or not to adopt this antipattern to gain its cost and time benefits. In this chart, good attributes start at a five-star rating, and negative consequences reduce the five stars.

Table 32-2. Comparison chart for the Stovepipe Architecture antipattern

Capability	Using Antipattern	Avoiding Antipattern
IMPROVED		
Cost	$	$$$
Initial Time to Market	★★★★★	★★★
DEGRADED		
Adaptability	★★	★★★★★
Agility	★	★★★★★
Availability	★	★★★★★
Deployability	★	★★★★★
Domain Alignment	★	★★★★★
Elasticity	★	★★★★★
Evolvability	★	★★★★★
Fault Tolerance	★	★★★★★
Maintainability	★	★★★★★
Responsiveness	★	★★★★★

Capability	Using Antipattern	Avoiding Antipattern
Scalability	★	★★★★★
Testability	★	★★★★★

The following sections describe each of the capabilities found in Table 32-2.

Improved

The capabilities that are improved when applying this pattern include:

Cost

> It costs money to analyze the architectural ramifications of potential changes, apply those changes, and iterate on the architecture, especially when using agentic AI and code generation — which is one reason teams simply bolt on new behaviors. However, those cost savings are generally short-lived, because as Stovepipe Architectures grow, they get harder and more expensive to maintain.

Initial Time to Market

> Teams can move faster and deliver behavior more quickly when they don't have to assess the architectural ramifications of each change. However, like cost savings, this benefit is usually short-lived.

Degraded

The capabilities that are degraded when applying this pattern include:

Adaptability

> Stovepipe Architectures don't adapt evenly to changes in technology, platform, languages, and other factors. Because they aren't starting from a clean, well-structured architecture, teams find that they can easily

adapt certain areas of the architecture, but other areas seem almost impossible to adapt. This inconsistency is pervasive in Stovepipe Architectures.

Agility

While *initial* time to market is improved by ignoring the architecture and simply bolting on behavior, over time teams will find it increasingly difficult to maintain, test, and deploy changes. Without a well-structured architecture in place, overall agility — the ability to respond quickly to change — will quickly degrade, significantly slowing teams down when applying changes to the system.

Availability

The ability of a system to be operational requires planning. Without a proper architecture in place, it's difficult for teams to ensure a reasonable and consistent level of availability for the system.

Deployability

Deploying software is a highly error-prone phase of the software development lifecycle, and typically carries with it high levels of risk that something will break. Rather than haphazardly bolting on behavior, having a well-defined architecture in place can significantly reduce the level of risk and ceremony involved with deploying software.

Domain Alignment

Bolted-on behavior tends to cross domain boundaries, hence degrading overall domain alignment. Having a well-structured architecture with well-defined domain boundaries helps in preserving domain alignment.

Elasticity

Elastic systems — those that have highly variable user load — require an architecture that is suited for this type of condition, with the focus being on the Mean Time To Start (MTTS) of services within that

system. Teams that ignore the architecture and bolt on additional behavior can significantly impact elasticity due to not paying attention to service granularity.

Evolvability

Simple, small systems can evolve to a certain degree using this antipattern, but larger, more complex systems require a well-defined architecture to evolve.

Fault Tolerance

This capability is about parts of the system being able to function while other parts fail, and also the amount of time it takes to recover from a fault (known as Mean Time To Recovery, or MTTR). This capability requires planning and an appropriate architecture. Services within a Stovepipe Architecture are typically interconnected in an ad hoc fashion, so that when one service goes down, all others are impacted.

Maintainability

As systems get larger and more complex, they naturally become harder to maintain. Teams will find that in unstructured systems without a well-defined architecture it's increasingly difficult to locate bugs, change existing behavior, and add behavior. By taking the architecture into account when adding or changing behavior, teams can better identify well-defined domains, components, and services with appropriate levels of coupling, allowing them to better understand and maintain the system.

Responsiveness

Stovepipe architectures can sometimes be responsive, but as those systems undergo change and become more complex, responsiveness usually degrades. Like most of the operational capabilities, responsiveness requires planning — and correspondingly, an appropriate and well-defined architecture.

Scalability

Teams that continue to bolt on additional behavior without regard to the architecture will find those systems rarely meet their scalability needs. Scalability requires up-front planning and an appropriate architecture.

Testability

Testability is about the ease of and completeness of testing. While Stovepipe Architectures may be *easy* to test, the *completeness* of testing quickly degrades as those systems become increasingly unstructured. Having modular systems with well-defined domains, components, and services with appropriate levels of coupling help achieve better completeness of testing.

Influences

This antipattern is about maintaining a well-defined and well-structured architecture instead of an ad hoc collection of components. As such, adopting this antipattern could influence aspects of infrastructure, data, team topologies, engineering practices, the enterprise, the business environment, and systems integration. Without a well-structured architecture, it would be very difficult to align a system with these areas.

Am I in This Antipattern?

Teams can identify whether they're in this antipattern by observing and analyzing both process-related and operational-related capabilities:

Process-related capabilities

Teams can measure and track the rate of change of the system, and the amount of time and effort involved in applying changes. If it takes too long to apply changes or add features, this might be a good indicator that the system lacks an appropriate structured architecture.

Teams can also measure and track the number of errors reported (and unreported). An increase in the number of errors over time is a good indicator that the system is unstructured and highly coupled, and would benefit from a well-defined architecture.

Operational-related capabilities

Teams can pay attention to and track complaints and errors related to things like scalability, responsiveness, fault tolerance, availability, and data integrity. All of these operational capabilities require planning and a well-defined architecture in place to achieve. As companies grow, these capabilities become increasingly important. Tracking these capabilities early and often allows teams to respond in a timely fashion to rewrite or refactor the system with an appropriate architecture to address these operational needs.

Related Patterns, Antipatterns, and Pitfalls

The following antipattern and pitfall are related to the Stovepipe Architecture antipattern:

Architecture by Implication antipattern (see Chapter 4)

Sometimes a Stovepipe Architecture implies a lack of proper architecture (or any architecture at all), so architects should be familiar with both of these antipatterns when creating or maintaining a system that requires specific architectural capabilities.

Grains of Sand pitfall (see Chapter 21)

The Grains of Sand pitfall is about service granularity and not making services too small. When applying changes and iterating on an architecture, architects must pay attention to the granularity of a service to resolve data needs, service coupling, and interservice communication issues.

Chapter 33. Supervisor Consumer Pattern

As companies grow and expand, their systems need to grow to support the increase in user load — in other words, they need to scale. *Scalability* is about a system being able to handle an increase in user load while maintaining a consistent response time for all users. The Supervisor Consumer pattern allows developers and architects to provide scalability within a service instance, allowing services to respond to multiple requests without having to create additional service instances.

Table 33-1 summarizes the metadata for the Supervisor Consumer pattern.

Table 33-1. Metadata for the Supervisor Consumer pattern

CATEGORY	METADATA
CONTEXT:	Event-Driven Architecture, High-Volume Processing, Message-Based Architectures, Microservices
IMPROVED:	Elasticity, Responsiveness, Scalability, Throughput
DEGRADED:	Message Processing Order, Simplicity

How Can I Improve Throughput and Scalability?

In message-based and event-driven architectures that use asynchronous communication, bottlenecks form when message *consumers* (services receiving the message) can't keep up with the volume of messages being

sent (we'll refer to messages and events interchangeably here and throughout the chapter). For example, in an event-driven architecture, a `Customer Notification` service might react to events triggered by many different services (such as when the customer places an order, when it's ready for shipment, when it's been shipped, and so on). Typically, a single instance of a `Customer Notification` service can only react to one event at a time, causing events to stack up. Another example in message-based systems is when a `Customer` service instance responds to `get_name` requests from other services needing the customer name. A single instance of a `Customer` service can usually only process one `get_name` request at a time, which can mean the requesting services have to wait too long for the `get_name` request to complete. This bottleneck is shown in Figure 33-1.

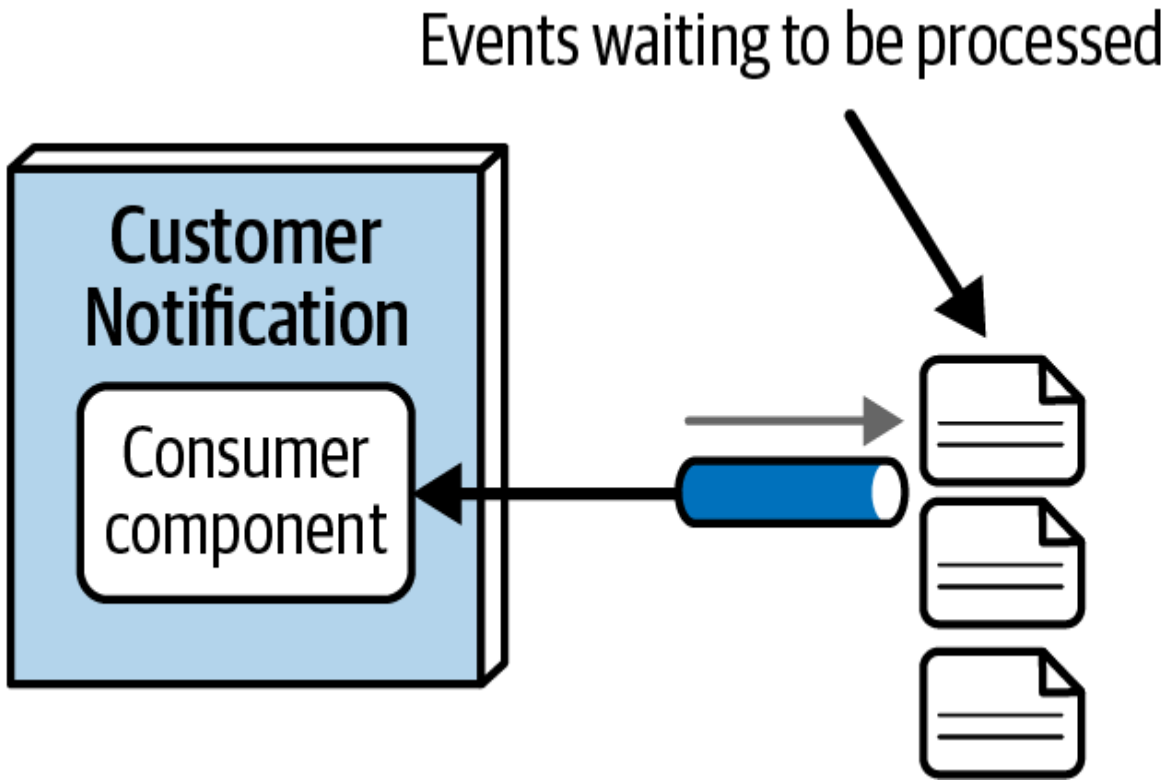


Figure 33-1. A bottleneck forms when messages are produced faster than they can be processed

Architects and developers often respond by creating additional instances of the message consumer service to handle the large number of pending messages, as illustrated in Figure 33-2. However, it's expensive to

instantiate new services, and it's not feasible to create hundreds of service instances when the message volume significantly increases.

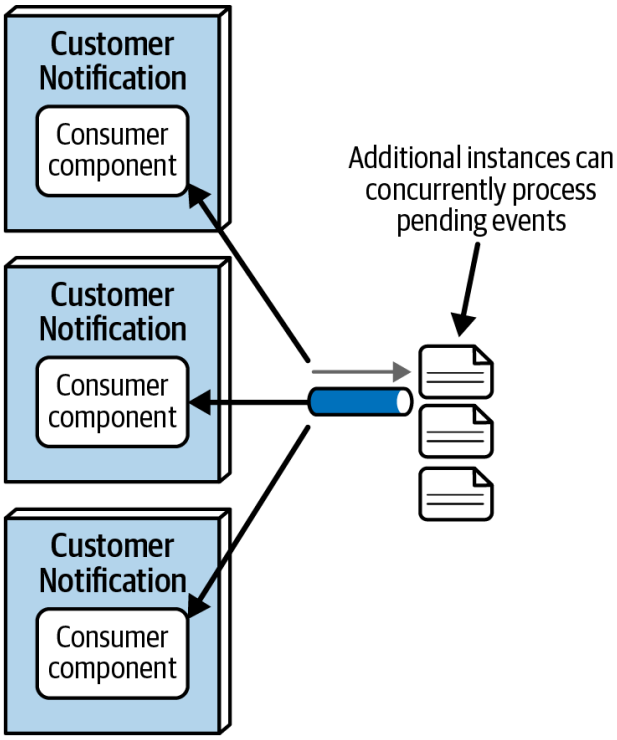


Figure 33-2. Multiple instances of a service can process requests concurrently, but not efficiently

Another technique is to utilize *multithreading* by wrapping the message consumer class in a thread and creating multiple message consumers within a single service instance. Using this technique, as shown in Figure 33-3, each service instance can process multiple requests simultaneously without having to create additional service instances. For example, if each `Customer` service instance creates 5 message consumer threads, then 2 instances can process a total of 10 `get_name` requests simultaneously.

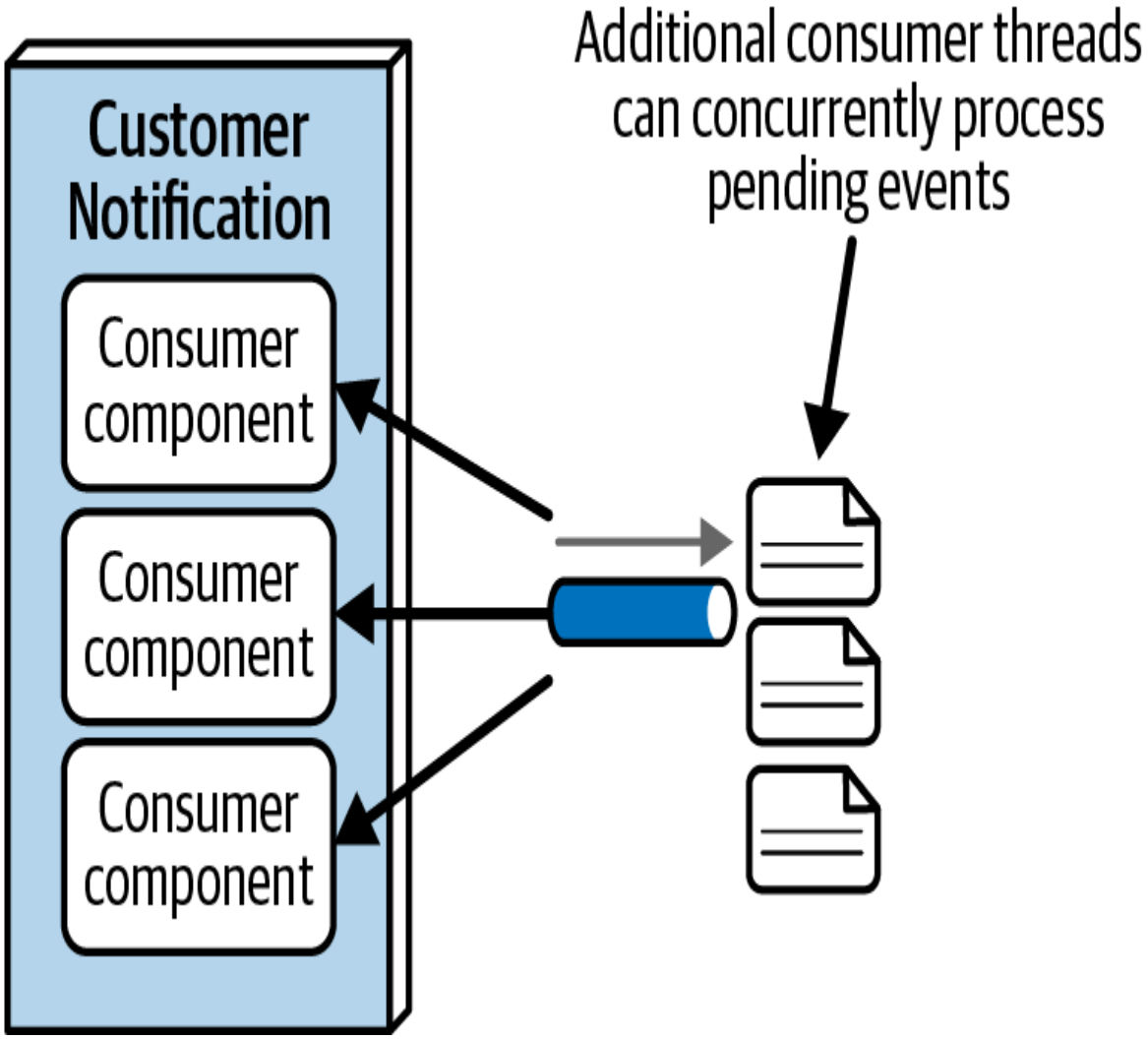


Figure 33-3. Increasing the number of consumers allows each service instance to process simultaneous requests

While this technique improves scalability without the need to add service instances, it can still produce a bottleneck. For example, in the prior example, if 100 `get_name` requests come into the request queue simultaneously, each of the 2 service instances can only process 10 concurrent requests, meaning 90 other requests will wait in the queue. Furthermore, in an idle state, those message consumers are unnecessarily consuming valuable memory and available threads within the service instance. The architect could certainly increase the number of message consumer threads, but the issue might still produce a bottleneck without enough consumer threads. This is even more expensive in an idle state.

The solution to this common problem is the Supervisor Consumer pattern, where each service instance continuously manages a *variable* number of message consumers, creating them when needed and destroying them when not needed. Using this pattern, architects can optimize thread and memory usage while ensuring that all enqueued messages or events are processed with a consistent response time.

Applying the Pattern

The Supervisor Consumer pattern has two components: a `Supervisor` component, which monitors the request queue, and a `Consumer` component, which dequeues and processes the next message from the request queue or event channel. This basic topology is illustrated in Figure 33-4.

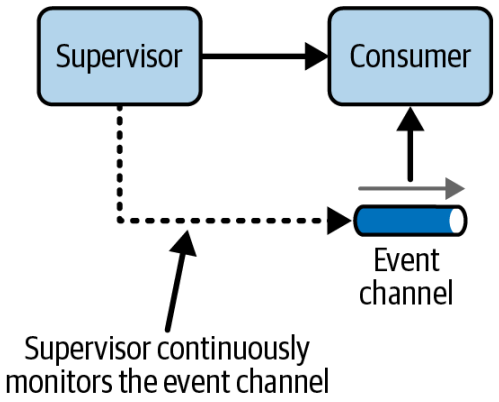


Figure 33-4. This pattern consists of a Supervisor component and a Consumer component

The `Supervisor` component is responsible for determining how many `Consumer` components to create dynamically, based on the number of pending messages in the queue or topic. If the pending messages outnumber the `Consumer` components, the `Supervisor` component will create additional `Consumer` components within the service instance to process the pending messages concurrently, as illustrated in Figure 33-5. The `Supervisor` component removes the `Consumer` components when they're no longer needed.

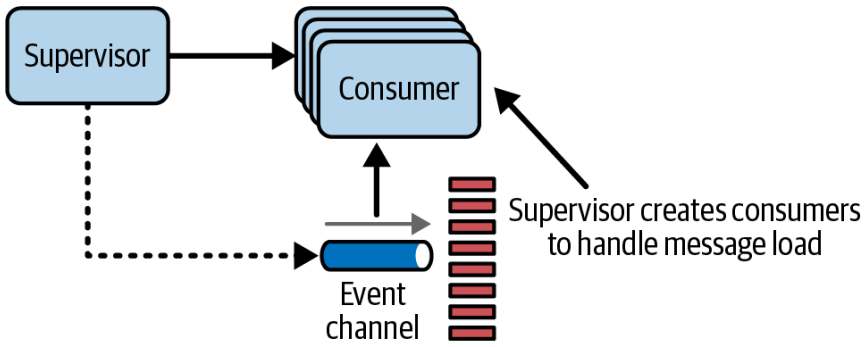


Figure 33-5. The Supervisor component dynamically creates Consumer components as demand increases

The `Supervisor` component uses the following basic algorithm to manage `Consumer` component threads:

1. Continuously monitor the number of pending messages in the queue or topic (called the *queue depth*), perhaps once a second.
2. Determine the number of `Consumer` components needed, based on the number of pending messages.
3. Apply a maximum threshold for the number of `Consumer` threads within the service instance.
4. Add or remove `Consumer` component threads as needed.

On startup (or in an idle state), the `Supervisor` component usually maintains a single `Consumer` component thread as the default, although this can easily be configured statically or dynamically. This is indicated with a `num_idle` attribute, as shown in the following pseudocode:

```
var list msg_consumers ❶
var num_idle_consumers = 1 ❷

connect_to_request_queue()
start_consumers(num_idle_consumers)
```

❶ The `Supervisor` component keeps track of the `Consumer` components it has created in this list.

❷ Developers can hardcode the number of default consumers or retrieve the value from an external configuration file. Another option is to use a

dynamic value that is calculated through basic machine-learning algorithms to optimize the startup or idle state of the service instance.

The `Supervisor` component periodically monitors the queue or topic to see how many messages are pending. How often it checks the queue depth largely depends on the throughput and frequency of messages, but generally most architects set this value to once a second:

```
var polling_freq_ms = 1000
while(true) {
   var queue_depth = get_pending_msg_count(queue)
   ...
   sleep(polling_freq_ms)
}
```

Based on the queue depth, the `Supervisor` component determines the number of `Consumer` component threads to create or remove, depending on factors including the number of pending messages, the maximum number of threads available in the service instance, and the number of service instances available. The `Supervisor` component generally uses the formula `queue_depth / num_service_instances`. For example, if 10 messages are waiting in the queue and 2 instances are available, each instance can create 5 `Consumer` component threads to handle the burst of requests:

```
var max_consumers = 500 ❶
var num_instances = get_instance_count() ❷
var consumers_needed = queue_depth / num_instances

if consumers_needed > max_consumers
   set consumers_needed = max_consumers

// create consumers
if consumers_needed > msg_consumers.size() ❸
   start_consumers(consumers_needed - msg_consumers.size())

// remove consumers
if consumers_needed < msg_consumers.size() ❹
```

```
    remove_consumers(msg_consumers.size() - consumers_needed)
}
```

❶ The maximum threshold for the number of `Consumer` threads is based on the number of available threads, the amount of available memory, and other constraints such as database connections.

❷ The service instance count can usually be determined by querying a configuration server or service orchestrator (such as Kubernetes).

❸ This line of code checks to see if more consumers are needed whenever the `Supervisor` component checks the queue depth (say, every second). If more `Consumer` components are needed, the `Supervisor` component creates more.

❹ This line of code removes `Consumer` components that are no longer needed.

The `Supervisor` component starts a `Consumer` component by creating a new `Consumer` class, adding it to the consumer list, and starting a new thread containing that `Consumer` class, as shown in the following pseudocode:

```
// startup consumers
for (consumer_count) {
    var consumer = new MessageConsumer
    msg_consumers.add(consumer)
    new thread(consumer).start()
}
```

To remove a `Consumer`, the `Supervisor` component invokes a `shutdown` operation, which causes it to wrap up its work and stop the thread:

```
// remove consumers
for (consumer_count) {
    var consumer = msg_consumers.get_next()
    consumer.shutdown()
    msg_consumers.remove(consumer)
}
```

This removes unnecessary `Consumer` classes from the consumer list and instructs them to shut down, freeing up threads and memory.

The pseudocode in Example 33-1 shows the basic algorithm for this pattern.

Example 33-1. Pseudocode describing the basic algorithm for the `Supervisor` component

```
var list msg_consumers
var num_idle_consumers = 1
var polling_freq_ms = 1000
var max_consumers = 500

connect_to_request_queue()
start_consumers(num_idle_consumers)

while(true) {
   var queue_depth = get_pending_msg_count(queue)
   var num_instances = get_instance_count()
   var consumers_needed = queue_depth / num_instances

   if consumers_needed > max_consumers
      set consumers_needed = max_consumers

   // create consumers
   if consumers_needed > msg_consumers.size()
      start_consumers(consumers_needed - msg_consumers.size())

   // remove consumers
   if consumers_needed < msg_consumers.size()
      remove_consumers(msg_consumers.size() - consumers_needed)

   sleep(polling_freq_ms)
}
```

Consistent Response Time

One way this pattern provides scalability is by ensuring a *consistent* response time for all requests made to a service instance. For example, suppose a `Customer` service instance responds to a `get_name` request from another service, asking it to query and return the customer name based on a customer ID. This takes 400 ms to complete. Without this pattern in place, additional `get_name` requests would stack up in the queue, significantly degrading responsiveness for other requests. However, with

this pattern in place, the `Supervisor` component creates as many `Consumer` components as it needs to guarantee a 400 ms response time for *all* the pending requests in the queue. This is illustrated in Figure 33-6.

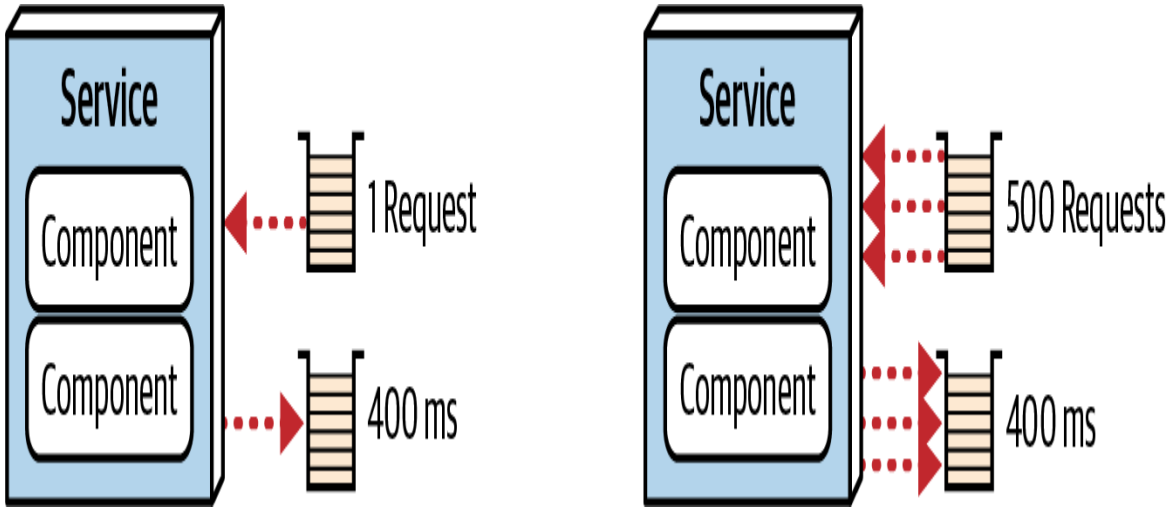


Figure 33-6. Each service instance can process multiple requests with a consistent response time

Macro Variant

We just described the *micro*-level use of this pattern — that is, programmatic scalability *within* a single service instance. In the *macro* variant, a `Super Supervisor` component creates additional service instances when the existing instances can't create enough `Consumer` component threads to handle all of the pending messages in the queue.

As illustrated in Figure 33-7, a `Supervisor` component communicates with a `Super Supervisor` component (usually a separate service) to inform it that the available threads within each service instance are insufficient to process the current messages or events. The `Super Supervisor` component creates an additional service instance to handle the extra pending messages. In the macro variant, the Supervisor Consumer pattern is essentially implemented *within* as well as *outside* each service instance, with the service instances acting like `Consumer` components.

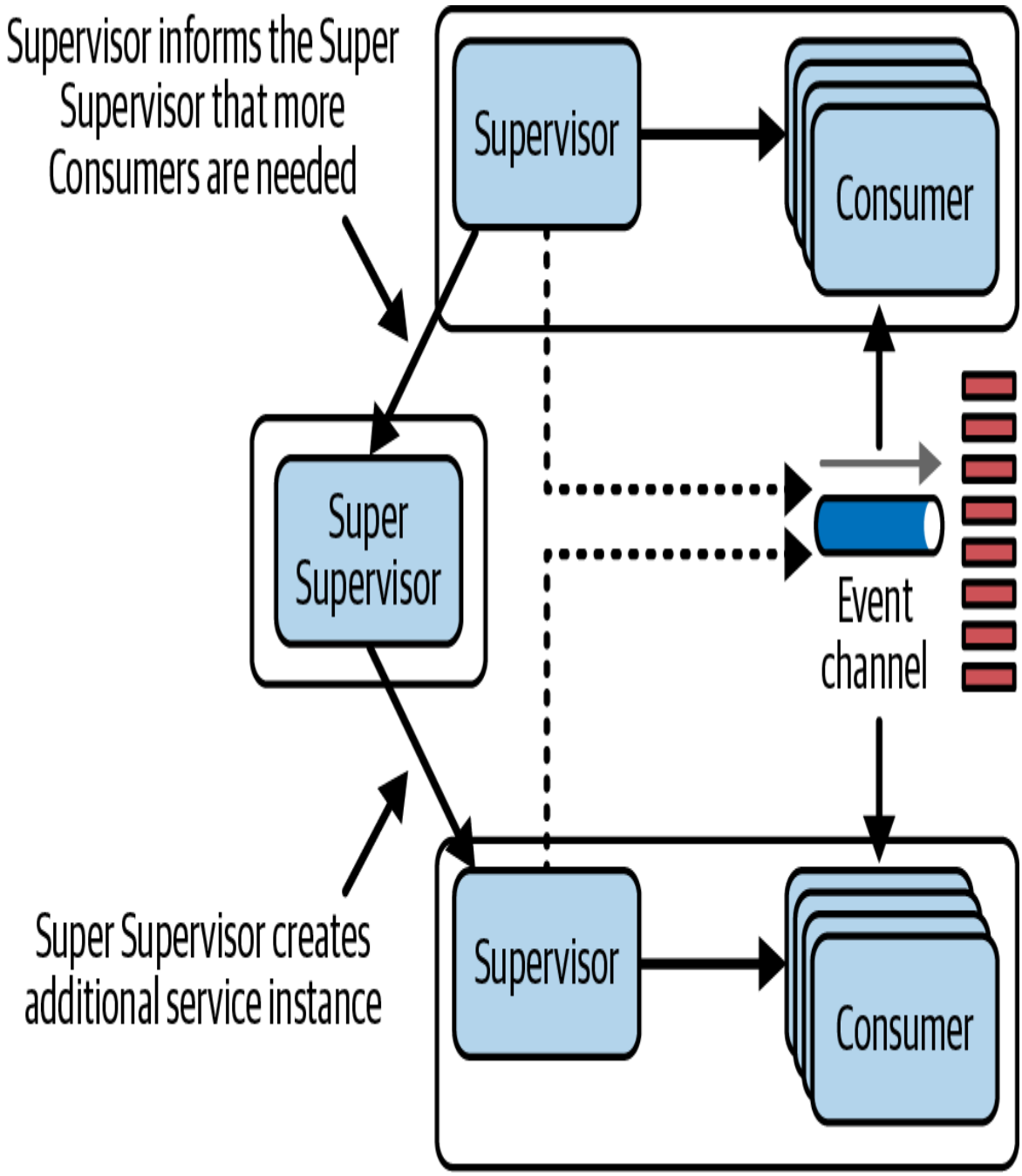


Figure 33-7. The Super Supervisor component is responsible for creating additional service instances

To illustrate how this variant works, consider a single deployed instance of a `Customer` service that can handle a maximum of 300 pending messages (that's its `max_threshold` setting). Now, suppose 500 pending messages are in the queue. The `Supervisor` component creates 300 `Consumer` component threads, then notifies the `Super Supervisor` component that 200 messages remain. The `Super Supervisor` component

instantiates a new `Customer` service instance with its own `Supervisor` component, which then creates 200 `Consumer` component threads. Once the new `Supervisor` component sees that it's no longer needed, it informs the `Super Supervisor` that it's done processing, and the `Super Supervisor` component shuts down the added service instance.

Another option within this variant is to have the `Super Supervisor` component monitor the request queue or topic and tell the `Supervisor` components how many `Consumer` component threads to create. This implementation requires the `Super Supervisor` component to know the service's maximum thread threshold, which it can find out through a registration process when the instance starts up. This variant is illustrated in Figure 33-8.

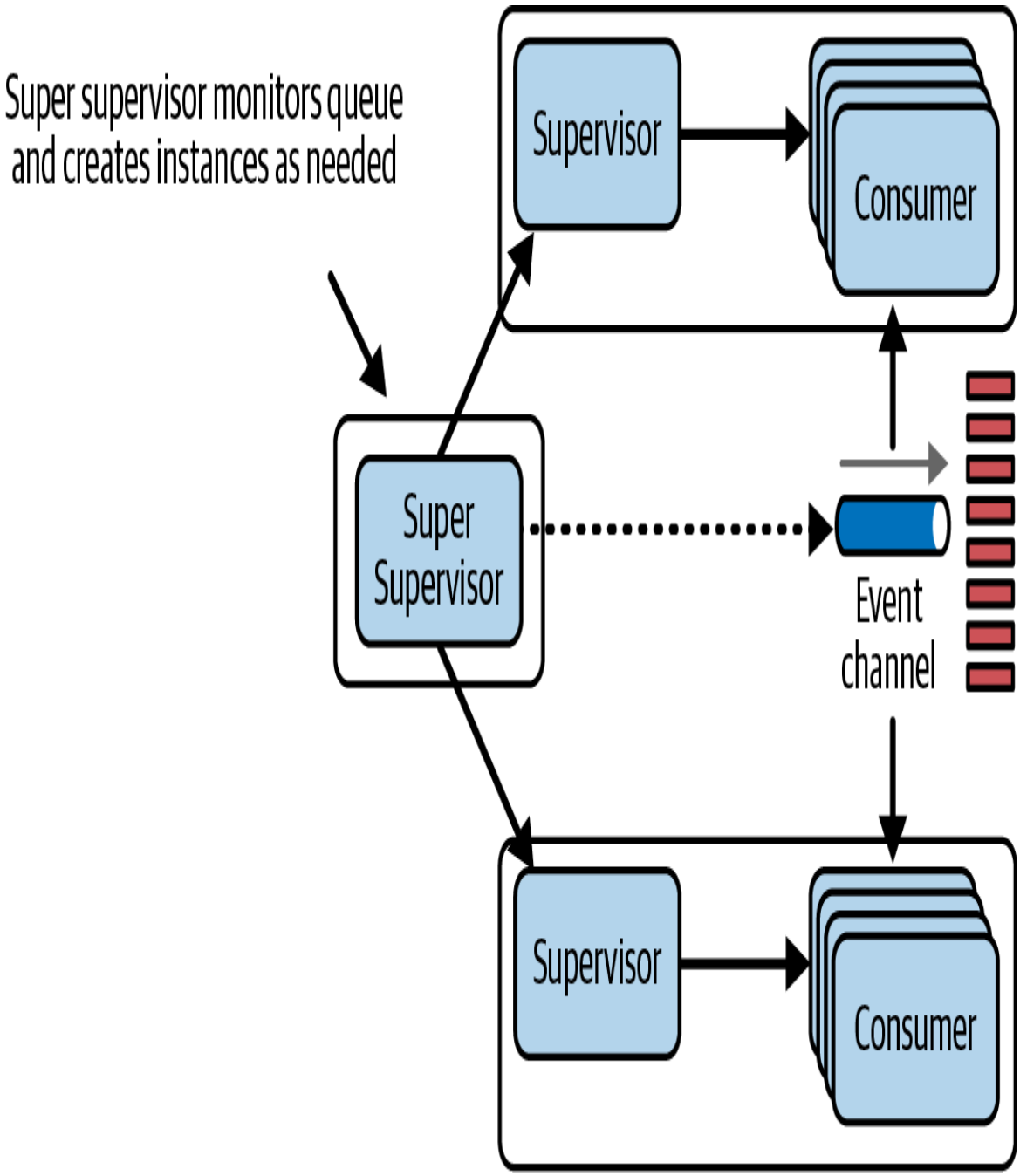


Figure 33-8. The Super Supervisor component can optionally be set to monitor the queue rather than the service instances

Trade-Offs

The Supervisor Consumer pattern can significantly improve a system's overall scalability, throughput, and responsiveness. However, architects

should be aware of the trade-offs, as outlined in the following sections.

Pros

Systems requiring high levels of operational capabilities like scalability, elasticity, message throughput, and responsiveness will see a significant benefit from applying this pattern as these are its main advantages. By being able to process message-based interservice requests concurrently in a single service instance, teams will be able to scale their systems more easily. Furthermore, by being able to dynamically adjust the number of message consumers in a single service, teams can optimize thread and memory usage within those services.

Cons

Although the Supervisor Consumer pattern's *topology* is not overly complex, dealing with errors associated with thread and memory saturation can be. Teams should make sure the benefits of applying this pattern outweigh the trade-offs associated with more complex source code associated with thread and memory management.

Teams should also be aware that although queue first in, first out (FIFO) order is preserved, message processing order is not. If message processing order is important, teams should look to the related patterns in "Related Patterns, Antipatterns, and Pitfalls" for possible solutions when applying this pattern.

Comparison Chart

Table 33-2 summarizes the star ratings associated with applying this pattern. The Without Pattern column refers to only having a single (or fixed number) of message consumers within a given service.

Table 33-2. Comparison chart for the Supervisor Consumer pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Elasticity	★	★★★★★
Responsiveness	★	★★★★★
Scalability	★	★★★★★
Throughput	★	★★★★★
DEGRADED		
Message Processing Order	★★★★★	★
Simplicity	★★★★★	★★

The following sections describe each of the capabilities found in Table 33-2.

Improved

The capabilities that are improved when applying this pattern include:

Elasticity

> This pattern works particularly well for elastic systems because the Mean Time To Start (MTTS) for creating additional `Consumer` component threads is magnitudes faster than creating a new service instance. When user load spikes, the `Supervisor` component can

respond immediately to create the `Consumer` components necessary to handle the increased load.

Responsiveness

This pattern significantly improves overall responsiveness, particularly for synchronous requests made through request/reply messaging, because it allows service instances to maintain the same response time regardless of the number of requests.

Scalability

The main advantage of this pattern is improved scalability. Enabling service instances to process multiple requests or events within the service instance is much more efficient and cost-effective than continually creating additional service instances to handle increased load.

Throughput

Because this pattern lets service instances process multiple requests and events concurrently, it improves overall throughput, thus reducing bottlenecks when the system is under increased load.

Degraded

The capabilities that are degraded when applying this pattern include:

Message Processing Order

This pattern does not maintain the message processing order. Even when using a FIFO queue, messages are processed concurrently, meaning some requests may finish before others even though they were dequeued after other requests.

For standard request/reply processing (such as our customer `get_name` request), this is seldom a concern. However, for message or event processing, it might present an issue. Refer to the related Thread

Delegate pattern in Chapter 34 for techniques to maintain message processing order when using the Supervisor Consumer pattern.

Simplicity

Although the algorithm for this pattern is fairly straightforward, managing multiple threads can get complex. Even when using thread pools, thread saturation and out-of-memory errors can occur within the service instance. Furthermore, if the `Supervisor` component consumes too many threads for the `Consumer` components, it could starve the service from performing other tasks (such as responding to a UI request through an API). Therefore, teams should be careful not to take up all of the available threads when using this pattern.

A common response to thread saturation is to allocate a certain percentage of threads for the `Consumer` components. Teams can manage this through the `max_threshold` attribute.

Influences

No other outside factors influence this pattern or change the star ratings found in the prior section.

How to Govern This Pattern

Architects can apply several measurements to determine whether to use this pattern and confirm that it's implemented correctly:

Locating bottlenecks through pending messages

To determine whether this pattern might be useful, architects can continuously measure and track queue depth to locate potential bottlenecks, particularly when deploying multiple instances of a service. Pending messages in a queue or topic suggest that the service instances can't keep up with the current request load; applying this pattern helps

reduce bottlenecks and hence improves overall throughput and responsiveness.

This measurement also tells the architect whether the Supervisor Consumer pattern is effective enough to handle the current load. Too many pending messages might be an indicator that the `Supervisor` component isn't using enough threads from its allocation or that additional service instances are needed. In these cases, architects should consider using the macro variant of this pattern.

Out-of-memory errors

This pattern allows architects to measure and track the frequency of out-of-memory errors within service instances, which indicate that the `Supervisor` component is consuming too many threads or too much memory. Architects can respond to these errors by reducing the `max_threshold` attribute value in each service instance, or by leveraging the macro variation of this pattern (see "Macro Variant") to increase the number of service instances.

Thread saturation

When using this pattern, architects can measure and track the occurrences and frequency of thread saturation. In response, they can reduce the `max_threshold` attribute value in each service instance, or leverage the macro variation of this pattern to increase the number of service instances.

Thread and memory usage

Gathering thread and memory usage metrics and tracking them is an excellent way to measure the macro variation of this pattern. The `Super Supervisor` component needs this information to determine if it should create or destroy additional service instances.

Related Patterns, Antipatterns, and Pitfalls

The following patterns are related to the Supervisor Consumer pattern:

Multi-Broker pattern (see Chapter 25)

> While the Supervisor Consumer pattern is about adding more consumers to improve scalability and throughput, the Multi-Broker pattern is about the opposite — adding more message brokers instead. That said, these two patterns can be used in conjunction to further improve scalability, responsiveness, and throughput.

Thread Delegate pattern (see Chapter 34)

> This complex pattern allows update-related messages to be processed in order. It can be used in conjunction with the Supervisor Consumer pattern to preserve the message-processing order when using multiple `Consumer` component threads for update-related requests and other processing.

Chapter 34. Thread Delegate Pattern

Imagine a customer placing an order for an item in an online store, hitting Enter, and suddenly realizing they've just ordered the wrong item. They *immediately* cancel the order, relieved that they caught it in time. Or did they?

One issue associated with message-based and event-driven architectures is managing the *processing order* of messages through the system. The Thread Delegate pattern not only improves scalability, responsiveness, and throughput, it guarantees that messages are processed in the order they were sent.

Table 34-1 summarizes the metadata for the Thread Delegate pattern.

Table 34-1. Metadata for the Thread Delegate pattern

CATEGORY	METADATA
CONTEXT:	Event-Driven Architecture, High-Volume Processing, Message-Based Architectures
IMPROVED:	Elasticity, Message Processing Order, Responsiveness, Scalability, Throughput
DEGRADED:	Cost, Error Handling, Simplicity

How Can I Guarantee Message Processing Order?

Many systems require messages or events to be processed in the order in which they're sent. Take the example of the customer who places an order and immediately cancels it: although orders might be *received* in order through a first-in, first-out (FIFO) queue, with multiple consumers, the `cancel_order` message *could* be processed before the `place_order` message, even though the `place_order` message was *received* first. Another example: suppose a customer places an order to sell their IBM stock in order to buy some Apple stock. The IBM sell order *must* complete before the buy order for Apple, or else the customer won't have enough money in their brokerage account to make the Apple purchase. These are just a few examples of systems where the processing order of messages matters.

Since this pattern applies to both events and messages, for brevity we'll refer to them interchangeably here and throughout the chapter.

Most architects use a single consumer, as shown in Figure 34-1, to preserve message-processing order — in other words, single-threading the processing of the messages in the queue. However, while this solution preserves message processing order, it's generally too slow for most business systems. Messages will stack up in the queue and likely time out.

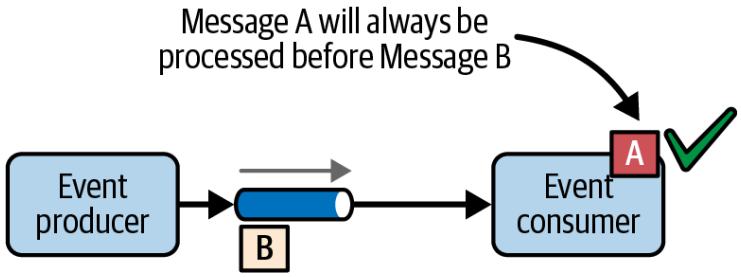


Figure 34-1. Using a single consumer with a FIFO queue guarantees messages will be processed in order, but is usually too slow

To improve scalability, responsiveness, and throughput, many teams use the Competing Consumers pattern, where multiple message consumers can process messages from the queue simultaneously. While this does dequeue messages in FIFO order, message processing order is no longer preserved because no mechanism exists to tie related messages together. As shown in Figure 34-2, `Message A` is picked up by the first message consumer, and

then `Message B` is immediately picked up by the second message consumer. However, the second message consumer ends up completing its processing of `Message B` before the first consumer completes `Message A`, resulting in these messages being processed out of order.

Many architects don't believe it's possible to process multiple messages simultaneously and still guarantee message processing order. Software architect and magician Michael Carducci gives a conference talk titled "The Art of the Impossible" where he turns an ordinary red handkerchief into a real egg.[1] Impossible? Of course. How does he do it right in front of the audience's eyes? As all great magicians do, he uses sleight of hand.

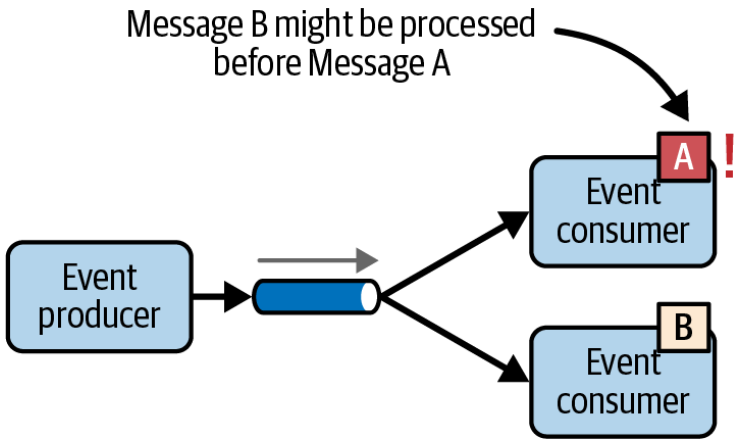


Figure 34-2. Using multiple consumers is faster but doesn't guarantee message processing order

The Thread Delegate pattern uses sleight of hand as well to guarantee message processing order. Here's the trick: not *every* message in the queue needs to be processed in order — only those within a specific context. Most business systems give messages some sort of context, such as a customer ID or account number. The Thread Delegate pattern leverages this premise to process *groups* of messages concurrently within a particular context.

To illustrate this sleight of hand, consider the following set of stock trades. A customer with brokerage account A-136 sells 2,000 shares of Google stock (GOOG) to buy 1,000 shares of Apple stock (AAPL), but then decides to cancel the Apple purchase:

```
Trade 1: PLACE SELL GOOG A-136 2,000.00
Trade 2: PLACE BUY AAPL A-136 1,000.00
```

```
Trade 3: CANCEL BUY AAPL A-136 1,000.00
```

Clearly, these stock trades must be placed in the same order in which they were sent to the `trade_order` queue (Trade 1, then Trade 2, then Trade 3). However, consider the current contents of the `trade_order` queue:

```
Trade 1: PLACE SELL AAPL A-136 2,000.00
Trade 2: PLACE SELL GOOG V-976 650.00
Trade 3: PLACE BUY ATT V-976 900.00
Trade 4: PLACE BUY IBM A-136 1,300.00
Trade 5: CANCEL BUY ATT V-976 900.00
```

Based on the brokerage account (which is the context of the stock trades), Trades 1 and 4 for brokerage account A-136 must be processed in order, and trades 2, 3, and 5 for brokerage account V-976 must be processed in order. However, *both* of these contexts can be processed concurrently while guaranteeing the processing order *within* each context group. This is essentially how the Thread Delegate pattern works.

Applying the Pattern

The Thread Delegate pattern is a relatively complex pattern that gets its name from the fact that it uses a single message consumer (called an *event dispatcher*) that delegates message processing to a thread it owns (called a *processor thread*). Figure 34-3 illustrates the components and basic topology of this pattern.

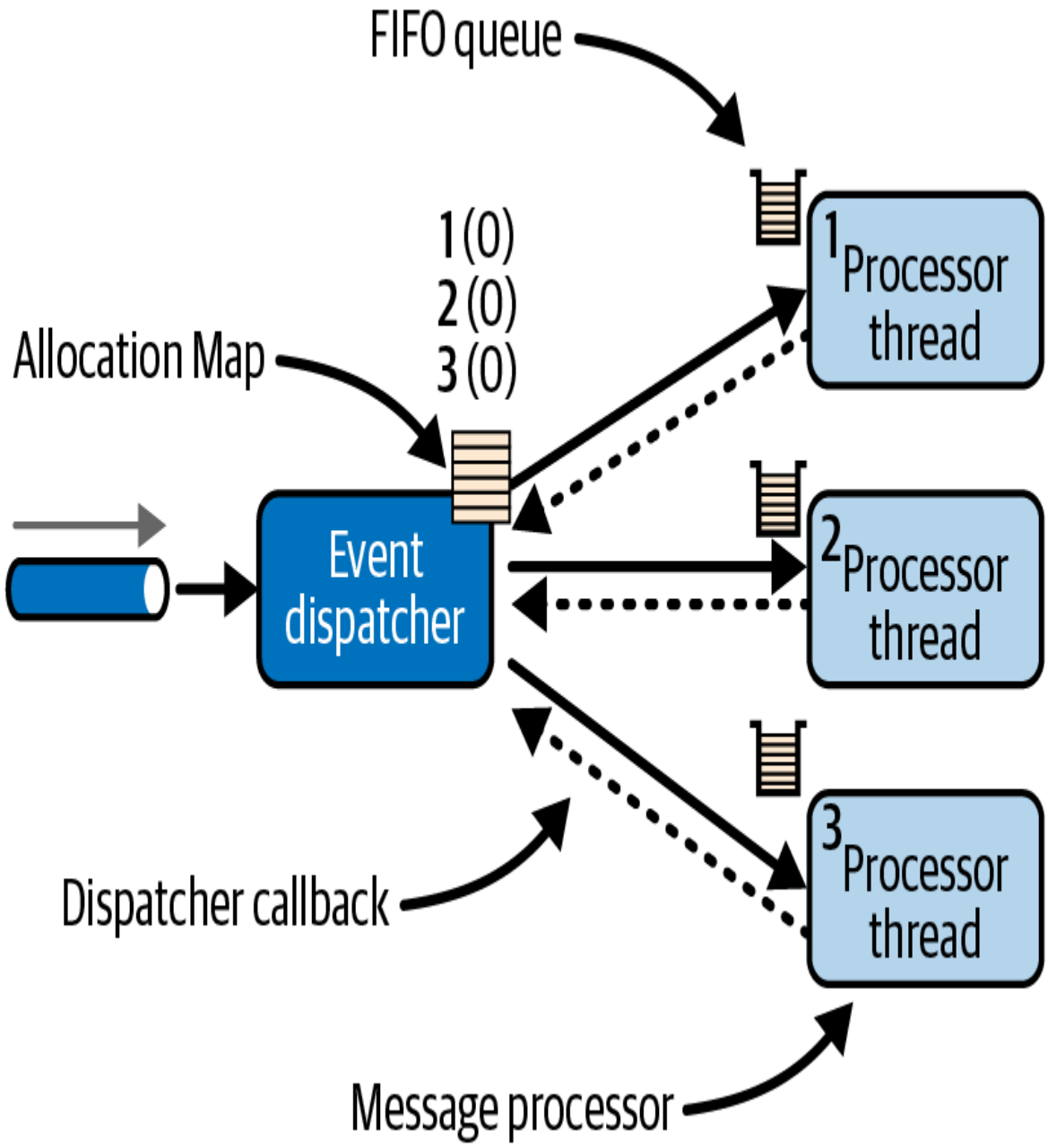


Figure 34-3. The basic topology of the Thread Delegate pattern

The event dispatcher is responsible for listening to the queue and determining which processor thread (message processor) should receive the message for processing. It uses an *Allocation Map* to keep track of which processor thread components are processing which contexts. Each processor thread component contains a FIFO queue, which can be implemented as an internal queue (such as `java.util.queue`) or an external queue (using a message broker).

The Allocation Map contains a list of the threads the event dispatcher owns, how many active messages each thread is assigned, and the context each thread is currently processing. Teams can implement the Allocation Map using three separate maps based on the `thread_id`, as shown in the following pseudocode:

```
var context_map = concurrent_map<context, thread_id> ❶
var message_count_map = concurrent_map<thread_id, message_count>
❷
var thread_object_map = concurrent_map<thread_id,
thread_delegate_object> ❸
```

❶ Map containing the context each thread is currently working on (if any).
❷ Map containing the number of active and pending messages sent to the thread.
❸ Map containing the actual thread object and its associated thread ID.

Alternatively, teams can create a single, more complicated Allocation Map containing all the information for each thread (`thread_id`, `thread_object`, `context`, and `message_count`). While either of these implementations works, we prefer using separate maps to represent the Allocation Map, making it easier for the event dispatcher to query and update the individual pieces of information. Using separate maps to represent the Allocation Map also makes it a little easier to have a single thread process multiple contexts if needed.

The Thread Delegate pattern consists of three main actions:

1. Assign the next message to a thread.
2. Process messages.
3. Update the Allocation Map when a thread has completed its processing.

The following sections detail the processing for each of these actions.

Assign a Message to a Thread

Upon receiving a message from the queue, the event dispatcher extracts the context from the message and checks the Allocation Map to see if any threads are currently assigned to that context. If no entry is found, the event dispatcher looks for the first available thread in the Allocation Map where the message count is zero and assigns that thread to the context, setting its message count to 1. Otherwise, if no threads are currently assigned to the context, the event dispatcher increments the message count for the thread currently assigned to that context, indicating that it's now processing multiple messages for the same context.

To illustrate how this algorithm works, assume the Allocation Map currently contains the following entries, where Thread 1 and Thread 2 are idle (message count 0), and Thread 3 is currently processing 1 message for brokerage account T-355:

```
Thread 1: [0]
Thread 2: [0]
Thread 3: [1, T-355]
```

The event dispatcher receives a message for brokerage account A-136. Since no other thread is working on that context, the event dispatcher finds the first available thread (in this case, Thread 1) and assigns it to the A-136 brokerage account (that's the context):

```
Thread 1: [1, A-136]  <--
Thread 2: [0]
Thread 3: [1, T-355]
```

Now suppose another message for the same brokerage account (A-136) comes into the event dispatcher. In this case it finds in the Allocation Map that Thread 1 is currently processing messages for A-136, so it increments the message count, effectively assigning the message to Thread 1 for processing:

```
Thread 1: [2, A-136]  <--
Thread 2: [0]
Thread 3: [1, T-355]
```

After assigning the message to a thread or incrementing the message count for that context, the event dispatcher then sends the message to the processor thread represented by that `thread_id`, along with a callback reference for later notification that processing is complete for that message. The event dispatcher can either send the message directly to the processor thread or to its queue. If sending directly to the processor thread itself, the processor thread is responsible for enqueueing the message. Alternatively, if it goes to the processor thread queue, the processor thread would dequeue the message when it's idle or when it has finished processing the current message.

When implementing this pattern, we generally prefer decoupling the processor thread queue implementation from the event dispatcher, instead placing responsibility for queueing on the processor thread component. This way, the event dispatcher doesn't need to be concerned about the queue's name or implementation (internal or external).

The pseudocode in Example 34-1 shows how the event dispatcher determines which processor thread an incoming message should be assigned to and sends the message to the processor thread for queueing or processing.

Example 34-1. Pseudocode for dispatching a message to a Thread Delegate component

```
var context_map = concurrent_map<context, thread_id> ❶
var message_count_map = concurrent_map<thread_id, message_count>
var thread_object_map = concurrent_map<thread_id,
thread_delegate_object>

var message = get_next_message(queue)
var thread_id = null

if context_map.contains(context) {
   thread_id = context_map.get(context)
} else {
   thread_id = get_next_available_thread() ❷
   context_map.put(context, threadId);
}
message_count_map.put(thread_id,
```

```
message_count_map.get(thread_id)+1);
threadpool.get(thread_id).assign_message(message); ❸
```

❶ These three maps represent the Allocation Map.

❷ "Running Out of Threads" describes ways of handling when no threads are available.

❸ This method, located in the processor thread component, receives the message and adds it to its FIFO queue.

Figure 34-4 shows the final outcome of the three messages assigned to each processor thread component for processing from the prior example. Notice that Thread 1 is currently processing a message and has another one queued up for processing *after* the one being processed completes. This is what maintains the processing order of all messages *within* a particular context.

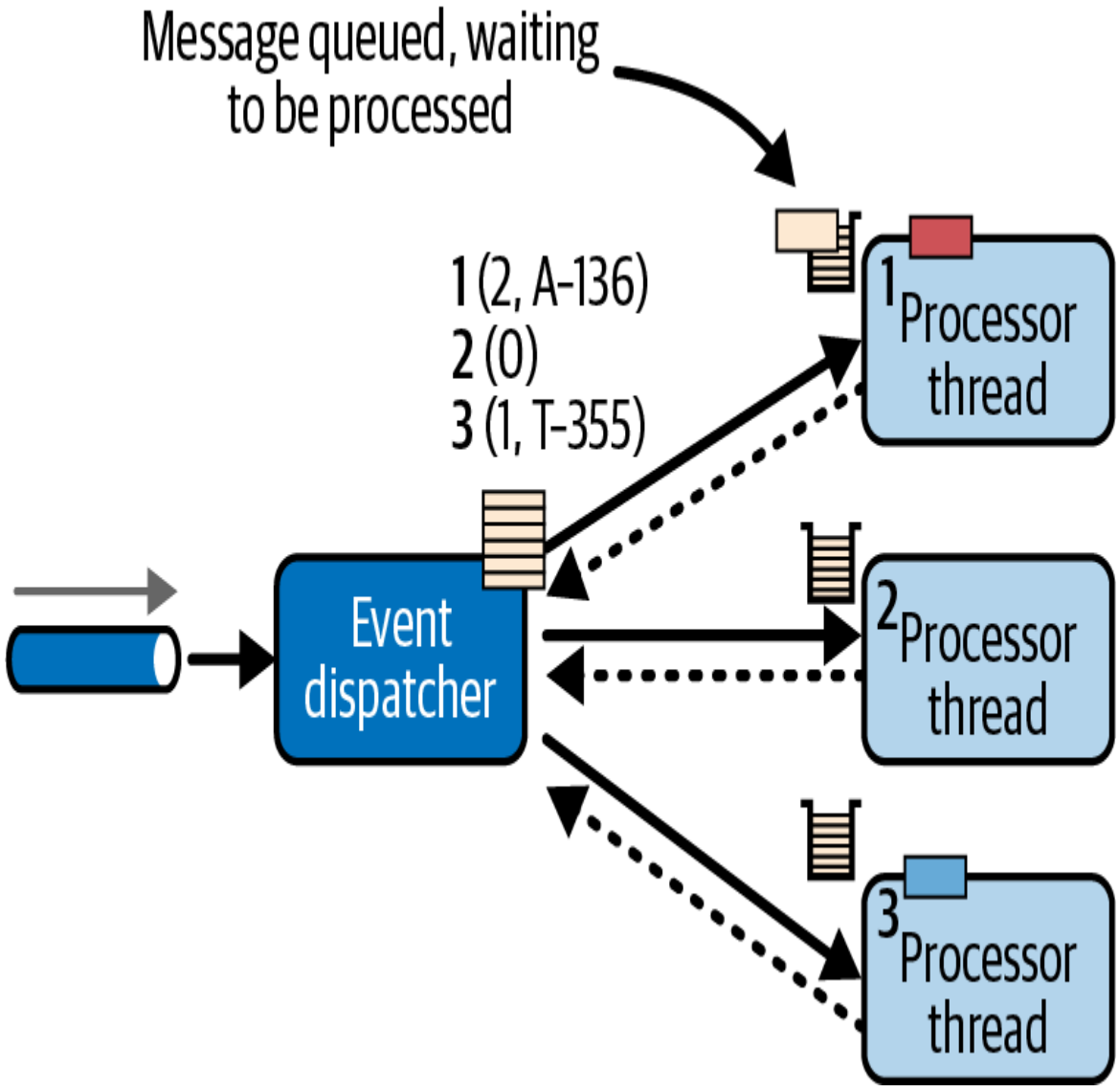


Figure 34-4. Three messages are assigned to threads, and two are currently being processed

Process Messages

Once the processor thread receives a message from the event dispatcher, it decides whether to queue it or process it. For simplicity, when implementing this pattern, we usually prefer to have the processor thread *always* enqueue a message it receives from the event dispatcher so that the normal processing simply dequeues and processes the next message. This implementation avoids the need for switching and state logic to let the processor thread know if it's currently processing a message or not.

Once the processor thread is done processing a message, it sends a notification back to inform the event dispatcher. This notification can be a simple callback by invoking a method in the event dispatcher using an object reference, or it can be a separate, external asynchronous notification through an external queue. We usually prefer the former, since the thread is owned by the event dispatcher. That way, the object reference to it can easily be passed into the processor thread when instantiating the `Processor Thread` object and creating the thread that contains it.

Example 34-2 shows the pseudocode within the processor thread component for receiving a message, processing it, and notifying the event dispatcher that it's done processing.

Example 34-2. Pseudocode for processing a message in the Thread Delegate component

```
var dispatcher = dispatcher_object_reference ❶
var fifo_queue = create_internal_linked_list_queue()

funct assign_message (var message) { ❷
   fifo_queue.add(message)
}

funct process_messages() {
   while (true) {
      if (fifo_queue.size() > 0) {
         var message = fifo_queue.get_next_message();
                process_message(message)
         dispatcher.processingComplete(thread_id, context); ❸
      }
   }
}
```

❶ The dispatcher object reference is usually passed in when instantiating a processor thread component.

❷ This function is invoked by the event dispatcher and adds the message to its internal or external queue.

❸ This is the callback to the event dispatcher so that it can maintain the Allocation Map (see "Update the Allocation Map").

Error handling is a fairly complex undertaking when applying this pattern (see "Trade-Offs"). Teams should consider using the Workflow Event pattern (see Chapter 35) to help manage errors when they occur. It's up to the processor thread, however, to hold off on processing any further messages in its queue (or specifically for that context) until it can successfully reprocess the faulty message.

Update the Allocation Map

When the processor thread finishes processing a message, it tells the event dispatcher through a callback so it can update the Allocation Map accordingly. When receiving a callback, the event dispatcher uses the `thread_id` to locate the thread that was processing the message, then decrements the message count for that thread. If the message count goes down to zero, the event dispatcher removes the associated context from that thread, freeing it up to process other messages in a different context.

Consider the Allocation Map from the prior example of stock trades:

```
Thread 1: [2, A-136]
Thread 2: [0]
Thread 3: [1, T-355]
```

Suppose the processor thread associated with Thread 1 notifies the event dispatcher via a callback mechanism that it has completed processing a message. The event dispatcher decrements the message count for Thread 1, changing the Allocation Map as follows. Thread 1 is now processing a single message — the one that was in its queue waiting to be processed:

```
Thread 1: [1, A-136]  <--
Thread 2: [0]
Thread 3: [1, T-355]
```

Immediately afterward, Thread 3 also completes its processing and notifies the event dispatcher, which decrements the count for Thread 3. Since the message count is now zero, it clears out the context, marking Thread 3 as available for another context to process:

```
Thread 1: [1, A-136]
Thread 2: [0]
Thread 3: [0]  <--
```

Running Out of Threads

Because threads are generally limited within a thread pool, it's possible to run out of threads. Perhaps all the threads are busy processing messages within particular contexts, and another message comes in with a context not currently being processed. The event dispatcher can either start up a new thread or overload another thread with an additional context.

In this situation, if the `Event Dispatcher` chooses to start up new threads, the team can overlay the Thread Delegate pattern with the Supervisor Consumer pattern (see Chapter 33), where the event dispatcher acts as a supervisor, spinning up and tearing down threads as needed. If the event dispatcher arrives at the maximum number of threads it can create, it can leverage the Super-Supervisor variant of the pattern to pass control to an external mediator to spin up additional service instances.

Another option is to allow a single processor thread to process multiple contexts within the same thread. For example, in this modified Allocation Map, Thread 2 is processing messages for *both* R-866 and W-234:

```
Thread 1: [1, A-136]
Thread 2: [2, R-866]  <--
Thread 2: [1, W-234]  <--
Thread 3: [0]
```

This option would require the event dispatcher to look for the `context` in addition to the `thread_id` when updating the Allocation Map. While this option still maintains message processing order, it degrades scalability, throughput, and responsiveness, because messages could queue up in the processor thread queue. This is one of those trade-offs that teams must analyze.

Trade-Offs

While the Thread Delegate pattern supports message processing order, it does come with some significant trade-offs that architects should be aware of and analyze prior to applying this pattern. The following sections describe the pros and cons of this pattern.

Pros

Preserving message processing order is this pattern's superpower. Aside from single-threading messages through a single consumer, not many techniques exist for preserving the processing order of messages while maintaining a high level of scalability, responsiveness, and throughput at the same time.

However, this is not the only reason to use the Thread Delegate pattern. Systems that don't need (or care about) message processing order can still take advantage of this pattern to achieve extremely high levels of scalability, elasticity, responsiveness, and throughput. Doing so makes error handling easier and reduces the overall complexity of this pattern.

Cons

One of the biggest drawbacks of this pattern is its inherent complexity. Not only does this tend to drive up costs, but teams may find it difficult to get this pattern right the first time.

Another big drawback is overall error handling. This pattern can spawn many error conditions, some of which might be behavioral in nature (such as a processing thread getting an exception during processing), or they might be infrastructure related (such as losing a thread context or running out of threads or memory). Regardless, teams should be aware of the complexities associated with error handling and not underestimate the effort involved in dealing with it.

Comparison Chart

Table 34-2 summarizes the trade-offs associated with applying this pattern.

Table 34-2. Comparison chart for the Thread Delegate pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Elasticity	★	★★★★
Message Processing Order	★	★★★★★
Responsiveness	★	★★★★★
Scalability	★	★★★★
Throughput	★	★★★★★
DEGRADED		
Cost	$	$$$$$
Error Handling	★★★★★	★
Simplicity	★★★★★	★

The following sections describe each of the capabilities found in Table 34-2.

Improved

The capabilities that are improved when applying this pattern include:

Elasticity and Scalability

Because this pattern uses multiple message consumers, it does a great job at supporting high levels of elasticity and scalability. However, because messages *within the same context* are processed in a single-threaded fashion, overall elasticity and scalability will not be quite as high as when processing messages out of order.

Message Processing Order

Maintaining message processing order when using multiple consumers is a very challenging undertaking. Although this pattern is complicated, it does preserve message processing order by grouping messages with the same context and processing those *groups* simultaneously. Messages within a queue that don't have distinctive contexts (or that all have the same context) don't match well with this pattern, so teams should analyze the nature of their messages to make sure they can leverage this technique.

Responsiveness

Because this pattern simultaneously processes multiple groups of messages, responsiveness is generally very good.

Throughput

Since the event dispatcher only needs to determine to which thread to forward the message when dequeueing messages, overall message throughput through the initial queue is very high.

Degraded

The capabilities that are degraded when applying this pattern include:

Cost

The Thread Delegate pattern is a relatively complex pattern to implement, so teams will spend more time and effort (and money) to implement systems using this pattern. Error handling (see below) also

tends to increase costs because the myriad errors that could occur require additional effort.

Error Handling

Perhaps the biggest negative trade-off is that teams must deal with a host of errors, including managing threads and memory, dealing with unresponsive threads, and handling faults during message processing when other same-context messages are in the processor thread queue.

Simplicity

Managing threads, thread pools, thread saturation, memory consumption, and error handling all increase the overall complexity of this pattern. Teams should make sure the benefits outweigh the negative aspects of this pattern before applying it.

Influences

No outside factors influence the star ratings for this pattern.

How to Govern This Pattern

Architects can apply the following measurements to determine whether they've implemented the pattern correctly and whether it's the right fit for their situation:

Thread saturation

When using this pattern, architects can measure and track the occurrences and frequency of thread saturation. In response, they can either reduce the maximum threads in the thread pool in each service instance or enable a single processor thread to process multiple contexts.

Error rates in Thread Delegate components

Error handling is particularly challenging in this pattern. By measuring and tracking error rates within the processor thread components, teams can determine the viability of using this pattern. Those systems that frequently have high error rates are not likely good candidates for this pattern.

Related Patterns, Antipatterns, and Pitfalls

The following patterns are related to the Thread Delegate pattern because they have to do with creating scalable systems, maintaining message processing order, and error handling:

Ambulance pattern (see Chapter 3)

The Thread Delegate pattern could be a pattern to use to maintain the processing order of priority messages when not using the Carpool pattern variant of the Ambulance pattern.

Multi-Broker pattern (see Chapter 25)

The Multi-Broker pattern does not preserve message processing order. Teams might be able to leverage the Thread Delegate pattern to create message affinity to a particular broker using the message context, ensuring that all messages of that same context go to the same broker.

Supervisor Consumer pattern (see Chapter 33)

Increasing the number of consumer threads in a service when using the Supervisor Consumer pattern does not preserve the message processing order. To preserve it, teams can combine the Thread Delegate pattern with the Supervisor Consumer pattern.

Workflow Event pattern (see Chapter 35)

Since error handling is typically challenging when using the Thread Delegate pattern, teams can leverage the Workflow Event pattern to

attempt to fix the problem and republish the message so that it can be processed successfully.

---

[1] Fast-forward about 15 minutes into the video to see the actual magic trick. In the video, surprisingly, Carducci shows the audience how he performs this trick.

Chapter 35. Workflow Event Pattern

Airport security lines are bad enough, but imagine how much worse they would be if carry-on bags tagged for further inspection were manually inspected while everyone else in line had to wait. Fortunately, those tagged carry-on bags are delegated to a separate conveyor belt and manually inspected asynchronously, while other bags without issues go straight through the regular conveyor belt without delay. That's how the Workflow Event pattern works.

Teams can apply the Workflow Event pattern to address two common problems associated with error handling: errors that arise during asynchronous processing — when no one (human or otherwise) is actively listening for them — and dealing with errors without holding up other requests.

Table 35-1 summarizes the metadata for the Workflow Event pattern.

Table 35-1. Metadata for the Workflow Event pattern

CATEGORY	METADATA
CONTEXT:	Event-Driven Architecture, Message-Based Architectures, Microservices, Workflow
IMPROVED:	Elasticity, Error Handling, Message Throughput, Responsiveness, Scalability
DEGRADED:	Message Processing Order, Simplicity

How Do I Handle Asynchronous Errors?

Asynchronous communication, primarily found in message-based and event-driven architectures, is a powerful way of improving scalability, elasticity, and responsiveness. However, these advantages come at a price — they make error handling more complex.

FastOrders is a (fictitious) high-volume worldwide ecommerce retailer. Because shipping times vary by location, the company guarantees that each order placed is at least *ready* for shipping within six hours. To accomplish this amazing feat, the FastOrders product-ordering system directs each customer order to one of hundreds of warehouses around the world. The packers in the warehouses must work extremely fast to pull items from shelves and get the order ready for shipping. They do so using the company's advanced fulfillment system, FastFulfillment.

Now suppose a packer is ready to fulfill the next customer order. The FastFulfillment system retrieves it, but the packer runs into an issue. Although the system shows plenty of inventory for this item, it doesn't have a warehouse location code, so FastFulfillment can't direct the packer to its exact location in the warehouse.

The company has a strict policy that *every* customer order must be fulfilled no matter what, and throwing an error would risk the customer order never being fulfilled. Instead, FastFulfillment instructs the packer to start searching the warehouse for the item. Two main issues should be clear. First, this workflow is forcing the packer to spend way too much time searching the warehouse for a single problem order, instead of fulfilling the next order on the screen. Second, if the system *did* simply throw an exception, no one (human or otherwise) would be paying attention.

The Workflow Event pattern is a way of handling errors during asynchronous processing that allows a problem request to "step aside" momentarily so that the system can process other pending requests while it fixes the problem asynchronously.

Applying the Pattern

Although it's considered a general architectural pattern, the Workflow Event pattern typically falls under the umbrella of *Reactive Architecture*:

> *Systems built as Reactive Systems are more flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving users effective interactive feedback.*
>
> — *The Reactive Manifesto*, September 16, 2014

Since the Workflow Event pattern applies to both event-driven and message-based architectures, we'll use the terms *event* and *message* interchangeably in this chapter; when we use one, assume we mean both an event and a message.

The basic algorithm of the Workflow Event pattern borrows concepts from continuous availability and business continuity. It focuses on four key principles when faults occur:

- *Delegate* the problem to someone else and move on.
- *Isolate* the problem so that it doesn't impact any further processing.
- *Contain* the problem so that it doesn't spread to other parts of the system.
- *Fix* the problem and resubmit it for processing.

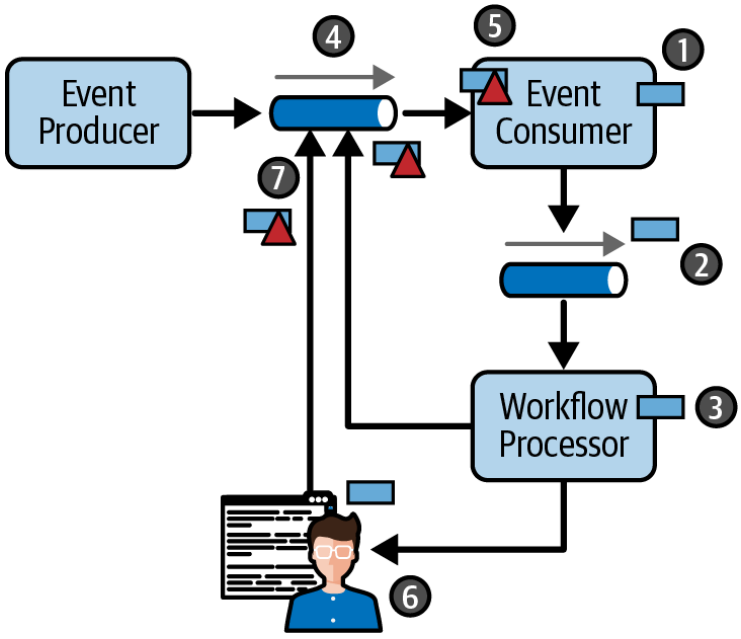


Figure 35-1. The basic flow of the Workflow Event pattern

Figure 35-1 illustrates the basic flow of this pattern:

1. The `Event Consumer` receives the next event and gets an error during processing.
2. The `Event Consumer` delegates the error by sending the event to a queue, along with helpful error message information.
3. The `Workflow Processor` component receives the event in error and tries to figure out how to fix it programmatically (see "Workflow Event Processor").
4. The `Workflow Processor` applies a fix to the problem and sends the updated event to the queue or topic to be reprocessed by the `Event Consumer`.
5. The `Event Consumer` receives the updated event and tries to reprocess it. If it succeeds, all is good and the workflow ends. However, if it still has issues, it tries again to send the event to the error queue and continues to step 6.
6. The `Workflow Processor` component continues trying to repair the error and the cycle continues until it finally gives up. At this point it sends the error to a dashboard (usually via another queue) for manual repair by a human (see "Manual Intervention (Dashboard)").
7.

The human resubmits the fixed event to the topic or queue so that the `Event Consumer` can reprocess it.

The following sections continue this example as we dive into the details of the components involved in these steps and how they work.

Workflow Event Processor

Steps 3 and 6 in the flow in Figure 35-1 involve the `Workflow Processor` component, which tries to fix the problem programmatically and, if that doesn't work, sends it off to a human for manual intervention. Typically, the error queue and `Workflow Processor` are tied to a specific context or service (or set of services), so typically a system has many `Workflow Processor` components.

In our example, the warehouse location for the item the customer purchased was not included in the order details. This produced an error in the `Event Consumer` service (which is responsible for displaying the order and the corresponding item locations to the packer), so the packer had to search for the item. If the team instead applied this pattern, the customer order with the missing-item-location error (we'll call it Order 1) would simply be sent to an error queue and the packer's screen would display the next order (Order 2) to be fulfilled.

While Order 2 is being fulfilled, Order 1 (the problem order) is being addressed by the `Workflow Processor` asynchronously. It's a lot like the carry-on bag that was tagged for manual inspection at the beginning of this chapter.

While the packer is fulfilling Order 2, the `Workflow Processor` sees that the warehouse location is missing from Order 1. It then searches the history of prior customer orders and notices that the item was previously ordered and picked from bin 108 on shelf 34 in the warehouse. It adds this information to the customer order and republishes the event to the order topic so it can be reprocessed with the new information. It also raises the order's priority to high (see the related Ambulance pattern in Chapter 3).

The `Event Consumer`, having finished Order 2, now receives the updated Order 1 as the next event in the topic. It's time to display the item location information. Here, two outcomes are possible:

- The item is there and the packer picks it from the shelf.
- The item location is wrong. The packer indicates on the UI that the item isn't there, and the UI again forwards that error to the error queue for the `Workflow Processor` to repair.

In this second round, the `Workflow Processor` might dig a little deeper and search for other potential locations, checking older orders or even the restocking system to find the correct location. If it finds a different location, it updates the customer order with the new location information to try again.

But what happens if the `Workflow Processor` can't find any new information? In this case, it would give up (as indicated in step 6) and send the customer order to a dashboard queue for manual processing (see "Manual Intervention (Dashboard)"). At this point, the `Workflow Processor` has completed its responsibility, so it moves on to the next issue in the error queue.

Using agentic AI in the Workflow Processor

The `Workflow Processor` uses error information passed from the `Event Consumer` to determine what changes (if any) it can apply to fix the problem: for instance, it may use historical information, preset rules, trend analysis, or AI agents.

The `Workflow Processor` can ask AI agents to suggest ways to fix the problem. Alternatively, the `Workflow Processor` itself could be implemented as an AI agent. Because AI is largely nondeterministic, we recommend having the `Workflow Processor` first try to resolve the issue deterministically. If nothing else works, then call upon agents to assist.

Dead letter queue

Most message-based systems use a dead letter queue (DLQ) when messages can't be delivered, time out, or are rejected by another queue or process. The problem with a DLQ is that typically no further downstream services exist to address the rejected message: it just sits in limbo, unprocessed, like children's letters addressed to "Santa, North Pole."

The Workflow Event pattern is a sophisticated version of a DLQ. Rather than sending problem messages to a generic DLQ to be ignored, teams can designate which error queue and corresponding `Workflow Processor` to use for a specific class of errors (or a specific service or workflow).

Whenever we see teams using a DLQ, we encourage them to consider using the Workflow Event pattern instead.

Manual Intervention (Dashboard)

After repeatedly trying to fix the problem, the `Workflow Processor` will eventually give up and forward the event to a queue for manual processing. A FastOrders order support employee receives it through a dashboard UI that looks similar to an email client, where errors appear as messages with an alert sound. The employee clicks to view the error information and the event payload.

The dashboard interface allows the person fixing the problem to change the event payload manually and resubmit it to a queue or topic for reprocessing. In our FastFulfillment example, they might search the warehouse themselves or research the problem manually. Once they find the correct item location information, they add it to the event payload and resubmit it for processing.

Trade-Offs

The following section outlines the general pros and cons of this pattern.

Pros

Because the Workflow Event pattern delegates error handling to another process, errors don't delay normal event processing. This generally improves operational capabilities like throughput, scalability, elasticity, and responsiveness.

Another advantage of this pattern is that it separates abstraction from error-handling processes: the `Event Consumer` processing the event doesn't need to know anything about the dashboard, retry logic, AI agent interaction, or error processing — all of this is delegated to the `Workflow Processor` so that the `Event Consumer` can focus on processing events.

Cons

Perhaps the biggest disadvantage of this pattern is that it doesn't natively preserve message processing order. While events in error are delegated and fixed asynchronously, `Event Consumers` can process the next event out of order. If it's important to preserve message processing order, teams can combine the Thread Delegate pattern (see Chapter 34) with the Workflow Event pattern, or record the context of the error (such as by customer ID or brokerage account), so that the `Event Consumer` processes no other events within that same context until the event in error is processed successfully.

Cost isn't usually a factor when applying this pattern because errors must be handled one way or another — whether through the `Event Consumer` or a `Workflow Processor`. However, because this pattern tends to add complexity, we urge architects to analyze whether their system would benefit from the advantages of scalability, elasticity, and responsiveness. If not, have the `Event Consumer` handle the error or send it off to a human for manual processing.

Comparison Chart

Table 35-2 summarizes the trade-offs associated with applying this pattern. The "Without Pattern" column shows the star ratings when resolving errors synchronously in real time using the `Event Consumer` service.

Table 35-2. Comparison chart for the Workflow Event pattern

Capability	Without Pattern	With Pattern
IMPROVED		
Elasticity	★★★	★★★★★
Error Handling	★	★★★★★
Message Throughput	★★★	★★★★★
Responsiveness	★★★	★★★★★
Scalability	★★★	★★★★★
DEGRADED		
Message Processing Order	★★★★★	★
Simplicity	★★★★★	★★★

The following sections describe each of the capabilities found in Table 35-2.

Improved

The following capabilities are improved when adopting this pattern:

Elasticity

Unlike scalability, elastic systems must be able to respond quickly to sudden bursts of heavy traffic. This pattern can improve throughput when errors occur — thus improving the system's elasticity — because the `Event Consumer` can guarantee consistent response time and throughput through error delegation. We only raised this rating by two stars, because this pattern only improves elasticity when errors occur. The more errors the system encounters, the more elastic it becomes when using this pattern.

Error Handling

Rather than responding to errors by simply "throwing an exception" and logging the issue, `Event Consumer` components can delegate the task of figuring out what went wrong and either fixing it programmatically or further delegating the error to a human for manual problem resolution. We increased this rating by four stars when using this pattern because most services don't have the sophistication to handle complex problem resolution, particularly when involving AI agents.

Message Throughput

Like in the airport security example, delegating errors maximizes the throughput of normal event processing. However, we only raised this rating by two stars because throughput is only gained when errors occur.

Responsiveness

Overall responsiveness in the system is improved because the additional time spent resolving error conditions is delegated to the `Workflow Processor`, creating consistent response times in the `Event Consumer` for events that don't have errors. We only raised this rating by two stars because responsiveness is only gained when errors occur.

Scalability

As user load increases, the probability of errors also increases. Systems that use this pattern can process more users because they spend less time trying to resolve errors, instead focusing on processing as many events as possible. Like the other operational capabilities, we only raised this rating by two stars because scalability is only gained when errors occur.

Degraded

The following capabilities are degraded when adopting this pattern:

Message Processing Order

Because errors are delegated to another process, message processing order is not preserved. "Trade-Offs" describes a few techniques for addressing this disadvantage.

Simplicity

Applying this pattern requires additional queues, components, and user interfaces, so it adds complexity to the system. However, we only degraded simplicity by two stars because these additional components are relatively straightforward to implement.

Influences

No other outside influences impact the star ratings found in Table 35-2.

How to Govern This Pattern

Teams can track several governance metrics to determine whether this pattern is a fit and assess its overall effectiveness in resolving problems:

Pattern use

The Workflow Event pattern is all about error handling, so teams might not find it useful if errors are rare in their system. Teams can measure

and record the number and frequency of errors the system encounters to determine whether or not it's worthwhile to adopt this pattern.

Teams can also track and record the nature of the errors they encounter. If the errors are minor and only flagged as warnings, this pattern would not be overly useful. However, if the errors encountered are critical and the corresponding problem resolution is complex, then teams should consider adopting this pattern.

It's also worth tracking the number of *repeated* errors encountered and fixed, which could indicate that something upstream can be changed to address the repeated errors.

Problem resolution

Teams can measure the effectiveness of this pattern by measuring and tracking how many errors the `Workflow Processor` resolves. If a majority of errors are simply forwarded to the dashboard for manual processing, it's a good indicator that the `Workflow Processor` is not as effective as it should be.

Related Patterns, Antipatterns, and Pitfalls

The following pattern is related to the Workflow Event pattern:

Thread Delegate pattern (see Chapter 34)

As discussed in "Trade-Offs", message processing order is not preserved when applying this pattern. If message processing order is important, teams may apply the Thread Delegate pattern on top of the Workflow Event pattern to preserve it.

Chapter 36. Build Your Own Pattern Analysis

Throughout this book, you've seen a wide variety of patterns, their capabilities, and what forces influences the trade-offs for each. Most pattern books treat the patterns (and antipatterns) as purely standalone forces, but, as we emphasize via our First Law of Software Architecture, everything in software architecture is a trade-off. Thus, in this book, we've focused on a more real-world glimpse of what evaluating patterns for actual use is like — how to choose a pattern (or avoid an antipattern or a pitfall) not in isolation, but within a larger context.

Inevitably, you'll encounter situations where you need to decide between two or more patterns that aren't covered in this book. So, in this chapter, we're going to go meta and show you how we derived our capabilities and comparisons, so that you can use the same techniques in your own work.

Evaluating Trade-Offs for Patterns

We pioneered a software architecture trade-off analysis technique we call *qualitative* analysis, represented in this book by all the trade-off charts as well as in our previous books *Fundamentals of Software Architecture*, *Software Architecture: The Hard Parts*, and *Head First Software Architecture*. Why is this a useful technique? It allows architects to make decisions without the luxury of building and measuring multiple solutions.

Imagine for a moment that the Big Boss has tasked you with implementing a new accounting workflow. It must keep all the monetary transactions accurate, of course, but must also operate at scale. As an architect, you have a couple of options to provide: "Boss, I have good news and bad news about the new accounting workflow. The good news? I'm going to deliver the *objectively best possible* accounting workflow, and I can prove it. The

bad news? To figure out which one that is, I have to implement it a half-dozen different ways and measure each result, so the budget is many multiples of the estimate, and the time frame is…extended."

Architects must constantly make decisions without full knowledge of all the exact engineering details, using only estimates. Building something and measuring the result is an example of quantitative analysis, which we distinguish from qualitative analysis. *Quantitative analysis*, or analyzing from hard data, appears throughout the book in the form of governance sections for each pattern — we are quantifiably measuring data from running software. This type of data is always preferred.

In a perfect world, when architects can't decide between one pattern versus another, they could implement both solutions and measure the results, guaranteeing the best outcome. Unfortunately, that perfect world is far from common: architects constantly must make and implement decisions based on incomplete knowledge. That's where qualitative analysis comes in.

Qualitative analysis relies on architects' basic understanding of what improves or degrades particular capabilities. For example, let's say that you're trying to compare the scalability of two patterns. One of the patterns uses synchronous communication; the other uses asynchronous. Even if you haven't implemented the patterns yet, your general experience indicates that the synchronous version won't scale as well because it involves more interaction between services that blocks while waiting for a response, harming the overall scalability.

Over the next few sections, we describe how you can build your own trade-off analysis.

Choosing Capabilities

The first thing you need are bases for comparison, and most of ours focus on capabilities.

We base most of our comparisons on capabilities, because that's the primary way the patterns differ from one another. We focus on general-purpose

architecture patterns, and *general-purpose* refers to the behavior of the pattern. For example, when choosing between Orchestration and Choreography, the desired *behavior* is workflow communication, which both patterns support more or less equally. The two solutions differ in what kinds of capabilities they improve or degrade. This is often the case, especially with general-purpose patterns.

However, when you're analyzing patterns within an organization, all bets are off: you might end up mixing and matching behaviors and capabilities when identifying the trade-offs for particular patterns or approaches. Your goal is to identify all the factors that will change some dimension of the trade-offs. This is the part that requires the most institutional and domain knowledge.

We didn't start this book with a predetermined list of capabilities. Instead, we focused on each pattern (or antipattern or pitfall — for this chapter, we'll simply say "pattern") individually, using a tool that allowed us to effortlessly link similar things together. (For more information about how we tracked this information, check out "Using a Zettelkasten (Slip Box)".) Once we completed all the patterns, we generated a cross-section of all affected capabilities so that we could build a "reverse index" in Chapter 2.

Identifying Influences

It would be nice if patterns in the real world acted like those in most books, which seem to live in blissful isolation. However, in the real world, where the patterns' rubber meets the road, architects must take many other factors into account. For each pattern, we list influences that may affect its capabilities and trade-offs one way or the other.

For each pattern, we choose factors that we knew would modulate the trade-off charts one way or another. For example, Orchestration has a particular level of scalability compared to Choreography, but whether you choose synchronous or asynchronous communication for Orchestration greatly affects your system's scalability. In other words, we identified other

common decisions architects might make that will affect the pattern's applicability.

When you build your own pattern analysis, you should also identify and account for any forces that change your trade-offs.

Stars and Other Icons

Many years ago, Mark and Neal started teaching workshops about the fundamentals of software architecture in an attempt to distill the minimum subset of information a developer would need to start their journey toward a software architect role. Any such curriculum should cover a variety of architecture styles (to distinguish styles from patterns, check out "Styles Versus Patterns"). We had to figure out a reasonable basis for comparison so that we'd be comparing "apples to apples" (like things).

We realized that the unifying common feature of the styles was that they're *generic* — the solution can be implemented for just about any behavior. The thing that makes them generic is the flexibility to solve any reasonable problem.

The primary difference was in how each one handled particular architectural characteristics or capabilities. Some styles lend themselves to certain capabilities and make others more difficult.

Our first pass was a simple binary comparison — does this style support this capability or not? We used a Roman-emperor-like thumbs-up or thumbs-down, as shown in Figure 36-1.

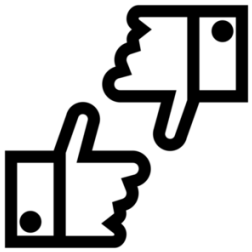


Figure 36-1. Our original trade-off metric, a binary thumbs-up or thumbs-down

In Figure 36-1, a thumbs-up indicated that the style supported that capability. However, we quickly realized that a binary scale would be too

coarse-grained for more than two comparison candidates. For example, different styles have different degrees of capabilities: it's not a binary "yes" or "no." Initially, we added a third state, a "sideways" thumb represented by a dash ("--"). However, we still felt this lacked nuance.

Five stars

We eventually settled on five stars as our preferred scale and have been using it for years. We like five stars because it allows nuance between competing patterns without being artificially precise. For example, if we chose a 100-point scale, how could we really distinguish between 67 and 73 stars? Using 5 stars means we have a 20% difference between each increment, allowing for differences without too much speculation.

These stars are always estimates based on architects' experience, not precise measurements (like quantitative analysis), so you should avoid trying for unjustified precision.

When evaluating a pattern trade-off, we look at the two options (generally, what the capabilities are with and without the pattern), and score maximized capabilities at five stars. If the other option is the same (in other words, there's no trade-off for that capability), we also assign it five stars. However, generally we score less supported capabilities with fewer stars, depending on the degree to which that support is lacking. This tells you that a negative trade-off exists and its magnitude.

When not to use stars

While we like stars as our iconic marker, don't be afraid to use other icons when they make more sense.

In the first edition of *Fundamentals of Software Architecture*, we chose stars for all our trade-off charts, including for cost. We wanted to highlight that a desirable cost is low and vice versa. However, when using stars for cost, five stars indicates the best value…which means that the lower-cost option gets five stars. Visually, this is counterintuitive, as we learned when we taught this material. Our mistake was choosing an icon (stars) that reflected the opposite of what we really wanted to measure, covered in the next

section. In this book (and in the second edition of *Fundamentals of Software Architecture*), we use the obvious iconic representation of money as "$".

Reflect what you want to measure

When you create trade-off charts for your candidate patterns, make sure your grading reflects what you want to measure.

For example, for our stars-versus-dollar signs example above, while stars represent the beneficial marker, people have a strong visual association with a larger number of items equaling a larger number. Translating that into the meaning we wanted required the reader to do some mental gymnastics.

The other important aspect of reflecting what you want to measure has to do with how you couch each pattern's benefits. For example, should you create a capability category called "coupled" or "decoupled"? Both measure the same aspect of the system, but which do architects value more? If you choose "coupled" as the capability, a high number of stars will indicate a negative aspect (a high degree of coupling). Conversely, measuring the capability as "decoupled" allows for a larger number of stars to indicate a more desirable criterion (a loosely coupled system). You'll notice this inversion in many of our patterns, where a higher number of stars indicates a more beneficial aspect. For example, when choosing between complexity versus simplicity as an effect of a pattern, most architects would rather see five stars for simplicity than one star for complexity.

Using a Zettelkasten (Slip Box)

While entirely optional, if you end up building your own pattern analysis, you might be interested in using the same types of tools we used to facilitate the comparisons, many of which are open source. Ultimately, we wanted to investigate not only the patterns but also the cross-pattern capabilities listed in Chapter 2. Toward that end, we used the open source Obsidian, which implements a Zettelkasten.

A *Zettelkasten* (German for "slip box," indicating a box with slips of paper, like index cards) is a linked note-taking technique developed centuries before software. It has long been implemented with an intensely manual process, but software makes it much easier to manage.

In Obsidian, all text is in Markdown. You can link items in two ways:

Double brackets

> Any item encapsulated within double brackets (`[[ ]]`) automatically creates a link to the item with the same name. We used this to name each pattern.

Hashtags

> Obsidian also allows users to create linkages using the familiar hashtag (`#`). However, we wanted to track several linkages via hashtags, so we added our own namespaces to them. For example, to every capability we referenced, we added the `#capability/` namespace; a capability like "responsiveness" appears as `#capability/responsiveness`. The tool autofills as the user types, so that you can't fail to create a link because you accidentally misspell a hashtag.

We focused on each pattern's context (`#context`), improved or degraded capabilities (`#capabilities`), and influences by other patterns (`#influence`). Using this tool allowed us to easily cross-reference every pattern that affects a particular capability by looking at all the patterns that link to it.

At the end of the process, we were able to create an interactive graph view in Obsidian, shown as a screenshot in Figure 36-2.

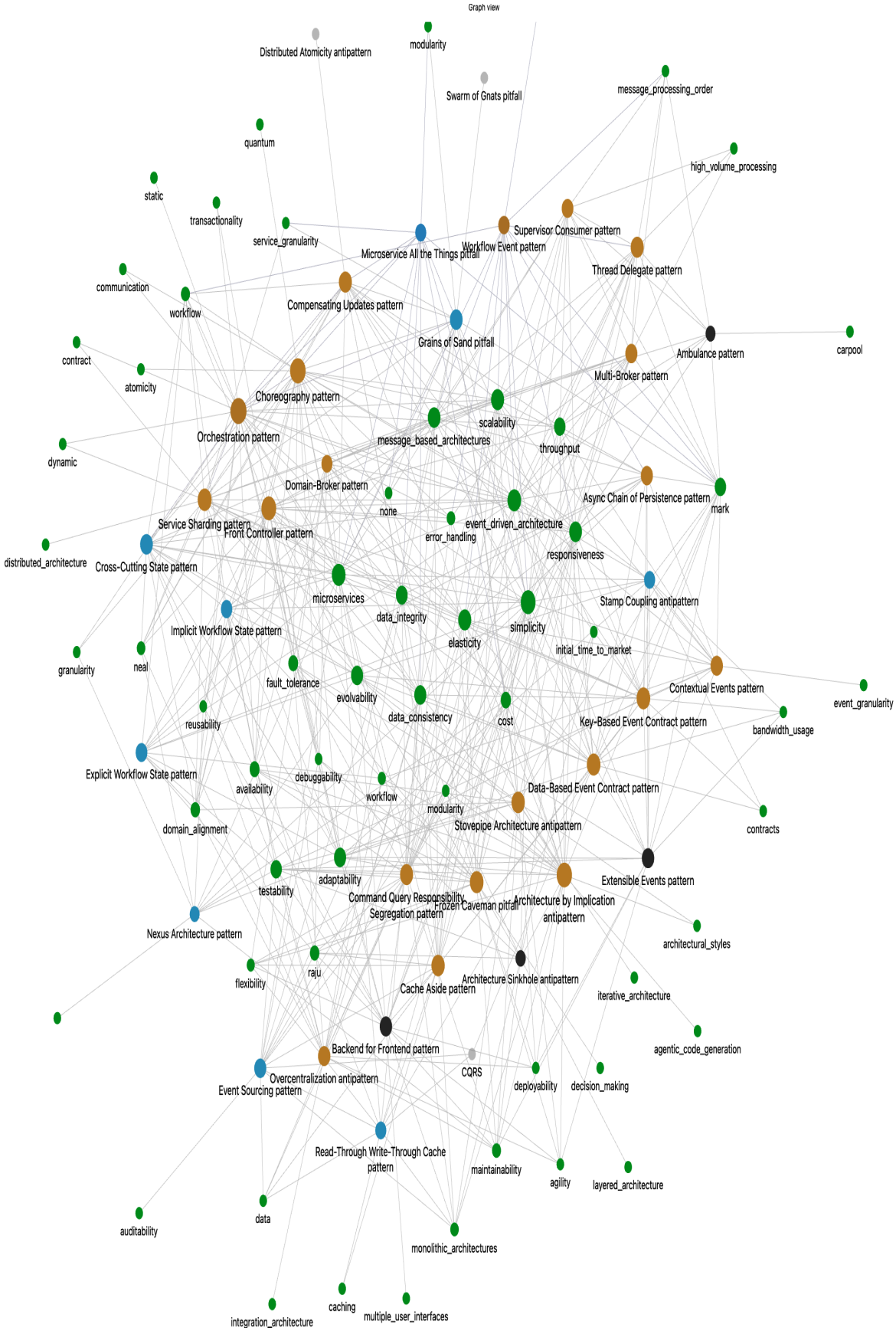

Graph view
Distributed Atomicity antipattern
modularity
Swarm of Gnats pitfall
message_processing_order
quantum
high_volume_processing
static
transactionality
service_granularity
Supervisor Consumer pattern
Microservice All the Things pitfall
Workflow Event pattern
Thread Delegate pattern
communication
workflow
Compensating Updates pattern
Grains of Sand pitfall
Ambulance pattern
carpool
contract
atomicity
Multi-Broker pattern
Choreography pattern
scalability
Orchestration pattern
message_based_architectures
throughput
dynamic
Domain-Broker pattern
Async Chain of Persistence pattern
mark
none
Service Sharding pattern
Front Controller pattern
event_driven_architecture
error_handling
responsiveness
distributed_architecture
Cross-Cutting State pattern
microservices
Stamp Coupling antipattern
data_integrity
simplicity
Implicit Workflow State pattern
elasticity
initial_time_to_market
granularity
neal
fault_tolerance
Contextual Events pattern
event_granularity
evolvability
data_consistency
cost
Key-Based Event Contract pattern
reusability
bandwidth_usage
Explicit Workflow State pattern
debuggability
availability
Data-Based Event Contract pattern
workflow
modularity
domain_alignment
Stovepipe Architecture antipattern
contracts
adaptability
Extensible Events pattern
testability
Command Query Responsibility Segregation pattern
Frozen Caveman pitfall
Architecture by Implication antipattern
Nexus Architecture pattern
architectural_styles
raju
flexibility
Architecture Sinkhole antipattern
Cache Aside pattern
iterative_architecture
agentic_code_generation
Backend for Frontend pattern
CQRS
Overcentralization antipattern
deployability
decision_making
Event Sourcing pattern
Read-Through Write-Through Cache pattern
maintainability
agility
layered_architecture
auditability
data
monolithic_architectures
caching
integration_architecture
multiple_user_interfaces

Figure 36-2. The Obsidian view of all the patterns, capabilities, and contexts

In Figure 36-2, each pattern appears in color while the capabilities and contexts appear in gray. You can interact with a live view of this graph at *http://softwarearchitecturepatterns.com*.

Evaluating Your Own Patterns

If you spend much time as a software architect, you'll find chances to analyze the trade-offs of patterns and other decisions. Feel free to use the techniques we cover here to make your comparisons useful.

Identify union of capabilities

The most difficult and contextualized part of this exercise is to build the list of capabilities you think this pattern will affect, whether negatively or positively. This list is highly domain- and organization-specific, based on experience and previous decisions. Often, an architect who wasn't present for some of the previous decisions can find architecture decision records (ADRs) or similar documentation for previous decisions. When looking at this information, pay close attention to the rationales for previous decisions, which may provide domain, technical, or organizational insight.

We recommend first looking at each pattern in isolation without a specific list of capabilities and trying to find the most impactful differences. Once you've done this for each pattern, unite the set of capabilities and see if some can be merged or eliminated.

Improved/degraded

When looking for affected capabilities, you can initially not pay attention to the impact of the capabilities but rather just have a contextualized list of them. We did that for each pattern in the book: we built the metadata charts that lead the chapters first, then only after careful analysis (including the possible influences) did we try to assign qualitative values and decide on improved/degraded.

It's useful to put off the determination of better/worse and the star ratings until you've seen the remainder of the capabilities — that provides you enough context to make better decisions about your eventual ratings. We discovered this over and over while writing the book — often we initially assumed a narrower context for a capability but realized that when we put it in context, it changed our evaluation.

From Qualitative to Quantitative Analysis

Using qualitative analysis allows architects to make trade-off decisions even when we don't have the time and resources to directly measure the effects of those decisions. However, qualitative analysis is a shortcut, not a substitute. Always consider governance for the patterns you choose, in addition to the original implementation. One of the reasons we include governance strategies for each pattern is to model how architects should think about building similar mechanisms.

Building governance via fitness functions allows you to complete the feedback loop you began with qualitative analysis. When architects verify their qualitative analysis via concrete fitness functions, they're testing their judgment within their ecosystem, and it's always a good idea to make sure that judgment reflects reality. The best way to verify that your evaluations are useful and hone their estimation ability is to measure the results of your decisions after implementation.

The Value of Patterns

No other industry has embraced patterns so enthusiastically, and even created the concept of antipatterns,[1] as software development has. Surely part of the reason is that we have so many similar problems that recur in surprising frequency and commonality.

It also reflects how detailed software design is, including software architecture. That's why we found it useful to catalog these architectural patterns that appear just about everywhere.

Our book is about general patterns. However, you can apply the same principles. During the design process, you often figure out relationships and common solutions within your ecosystem. When you find a good solution, you should catalog it like our architectural pattern and publish it internally.

One of the astute observations from Domain-Driven Design (DDD) is the concept of a *ubiquitous language*, making sure that all members of a software development team use precise language to prevent misunderstandings. In DDD, that concept resides at the domain level. But you can do the same thing for all the architects within your organization, creating your own ubiquitous language to describe architecture: common definitions of capabilities (you can use our "Capability and Context Cross-Reference" in Chapter 2 as a starter list), and the common architecture patterns in use within the organization.

Just like design patterns, architecture patterns encapsulate a huge amount of context (structural, trade-offs, influences, and so on) into a single name, creating a shorthand language that architects can use to both shorten their communication and make it more precise. We encourage architecture teams to create and curate their own list of patterns.

Since we're done here, now would be a good time to start your own pattern catalog.

---

[1] Coined in 1995 by Andrew Koenig.

Chapter 37. Fundamental Definitions

When writing a book about architecture patterns (and antipatterns and pitfalls), the authors have to either assume some existing knowledge on the reader's part, or redundantly put it into every chapter where it applies. For example, architects have many nuanced measures for coupling, like the architecture quantum concept covered in "Architecture Quantum". Many of our patterns affect the architecture quantum, but we can't assume that all readers are familiar with the concept. If we could assume that readers are familiar with our other books on the subject, like *Fundamentals of Software Architecture* (now in its second edition) and *Software Architecture: The Hard Parts*, we could skip the exposition — but however much we would like that to be true, we can't rely on it.

This being a patterns book, we also can't assume that readers will consume it in linear order. While we try to accommodate that type of reading, this is a patterns book, so it functions more like a catalog for an architect with an applicable problem. That means we can't rely on placing definitions in the first chapter where the reader will encounter them, as we would in a linear book — we don't know what chapter anyone will read first.

This chapter serves as a common place to define concepts readers might already know that are relevant to many different patterns. Whether or not these ideas are familiar, all readers will benefit from having a common place to reference these recurring concepts. As we mentioned in Chapter 2, these terms aren't locked down like mathematical notation — different institutions may have different definitions for many of these things that make perfect sense within their context. However, we provide some general definitions that fit within the context of our patterns.

ACID Versus BASE

Transactionality is a common concern in software architecture, representing an intersection between data concerns and architecture. It's defined by the acronym ACID, which stands for Atomic, Consistent, Isolated, and Durable, and originated in 1983.[1] We define each of these terms and their impact on distributed architecture below.

For many decades, most concerns about data consistency and transactions were relegated to singular relational databases, but as distributed architectures became more popular, architects were forced to distribute data as well. However, one of the core features of relational databases — ACID transactions — doesn't exist in distributed architectures. This section defines ACID transactions, explains why they're impossible in distributed architectures, and provides some alternative ways to handle this problem.

The acronym ACID is a way of describing a set of desirable properties for transactional systems. To understand the need for compensating updates, architects need to understand what capabilities they're partially replacing.

ACID is a convenient acronym because it's a real word. In chemistry, the pH scale ranges from acids to bases. Thus, to mimic the metaphor in software architecture, the opposite of ACID is *BASE*: Basic Availability, Soft State, and Eventual Consistency:

Basic Availability

> *Basic Availability* means that users can access data storage concurrently without waiting on transactions.

Soft State

> *Soft State* implies that some updates may be temporary or "soft." For example, in Figure 37-7, the first two tables (`order` and `payment`) are in a changed state before the workflow is complete. The architect may implement a pattern to restore the state of the previous updates, as covered in the Compensating Updates pattern in Chapter 11.

Eventual Consistency

> One of the most common characteristics of microservices is *Eventual Consistency* rather than atomicity. Eventual Consistency doesn't insist that all changes happen in lockstep, as in an atomic transaction, but rather that they will eventually reach a consistent state. For example, in Figure 37-7, the tables in the three services don't form a proper order until the system has updated all three services, eventually becoming consistent. Of the three features of BASE, Eventual Consistency is the most common and appears as the target of many of our patterns.

It's important to understand the implications of ACIDity for software architects because one of them, *atomicity*, is often handcrafted using the aforementioned Compensating Updates pattern or similar.

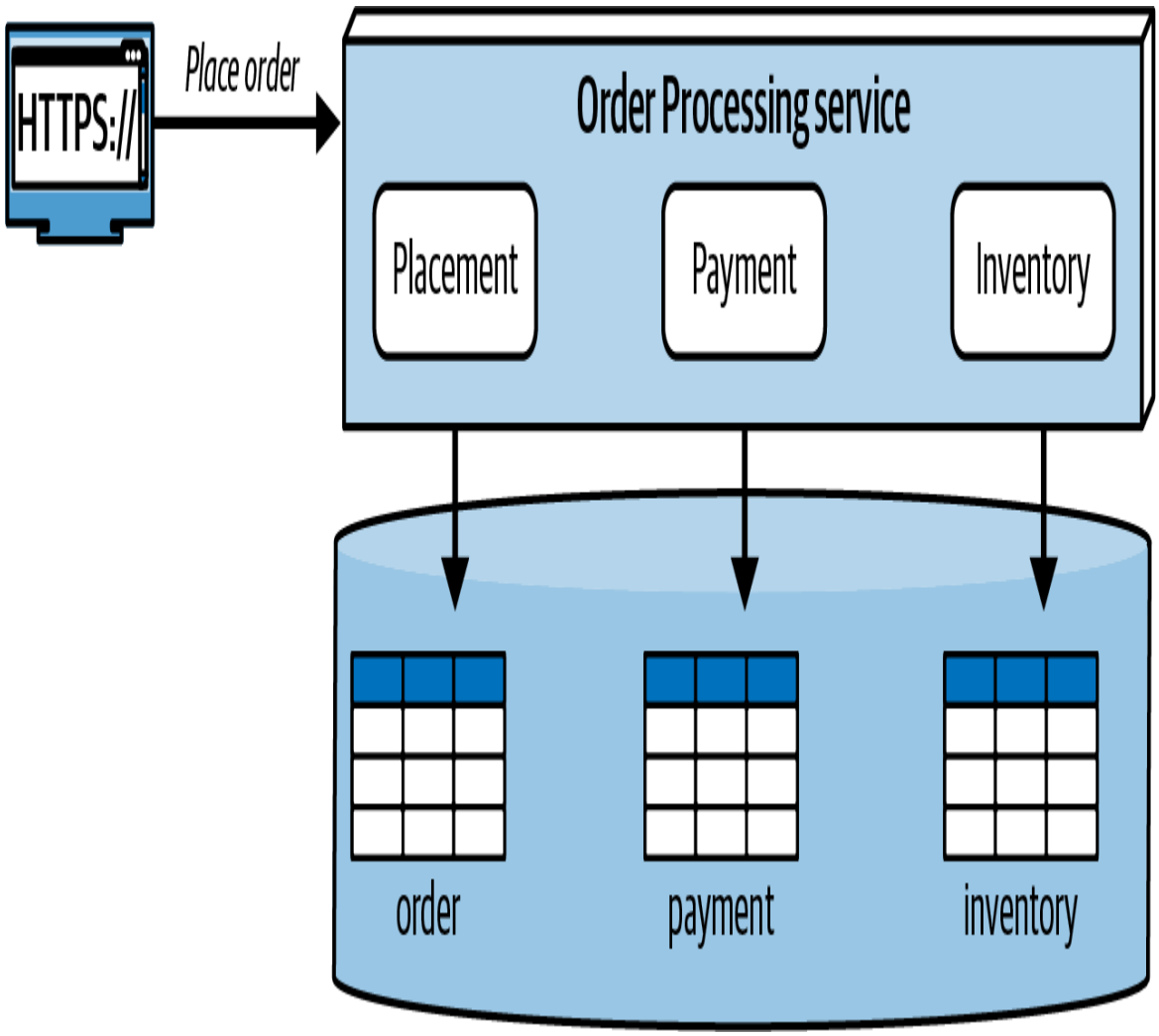


Figure 37-1. The topology for a simple monolithic order processing system

In Figure 37-1, we assume that inventory has an existing value.

Alternatively, this system could be modeled as a distributed architecture such as microservices, as illustrated in Figure 37-2.

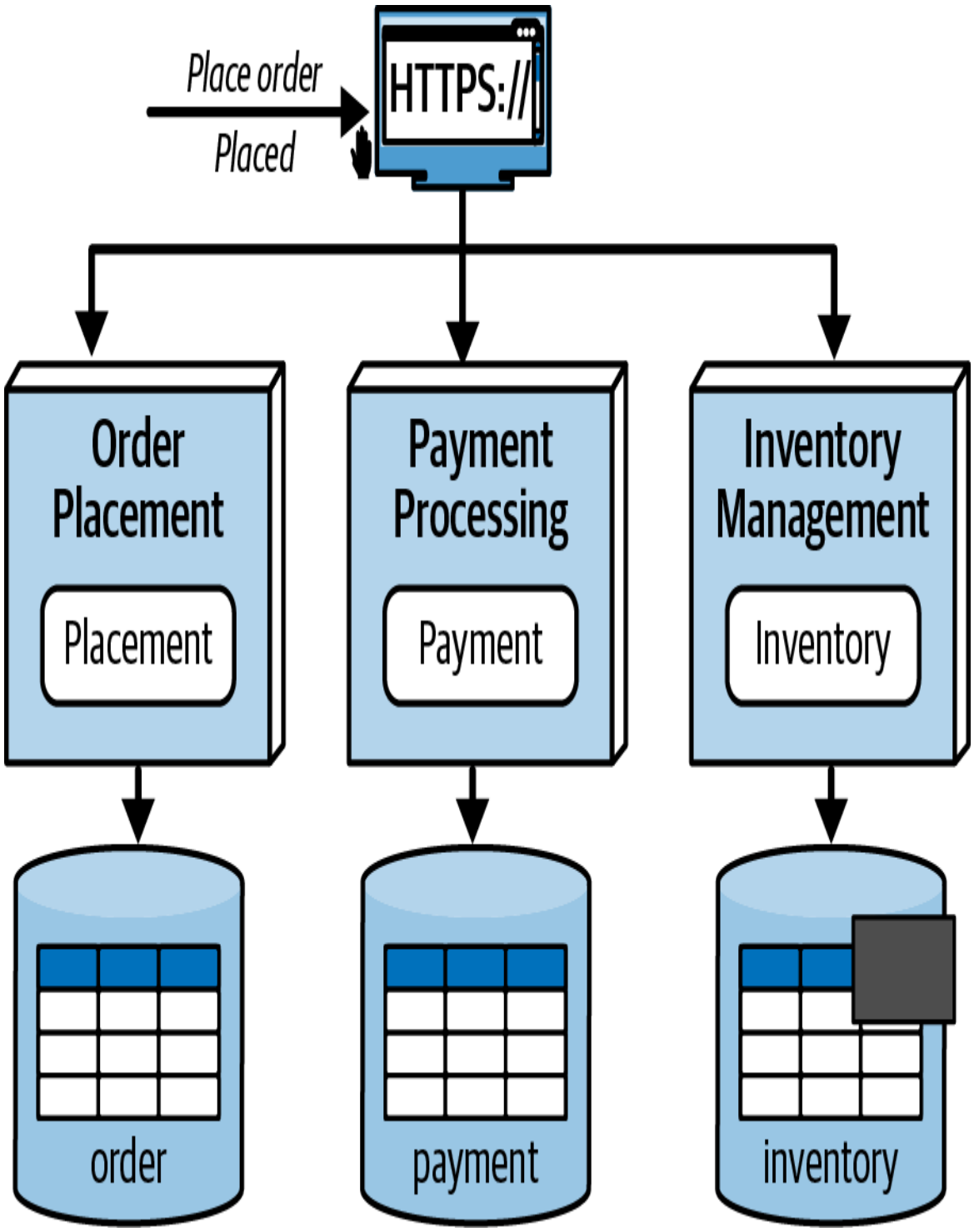


Figure 37-2. The order processing system could also be implemented in a microservices architecture

In Figure 37-2, each of the system's capabilities is modeled as a microservice with its own data source.

While Figure 37-1 has a single database, it can rely on its transactional capabilities. However, in Figure 37-2, each service has its own database instance, preventing ACIDity across the databases.

Architects, developers, and business users have become accustomed to the transactional capabilities relational databases offer (and often assume their presence). However, ACID isn't possible in distributed architectures, for reasons we'll illustrate property by property.

Atomicity

Atomicity refers to the ability to treat a multistep process as if it were a single, unified process; we use it as a composite term for *data consistency* (making sure data is synchronized) and *data integrity* (making sure data is correct). It guarantees that the system completes an entire unit of work. If it can't for any reason, atomicity ensures that the system restores to its previous state.

To illustrate, consider the successful workflow illustrated in Figure 37-3.

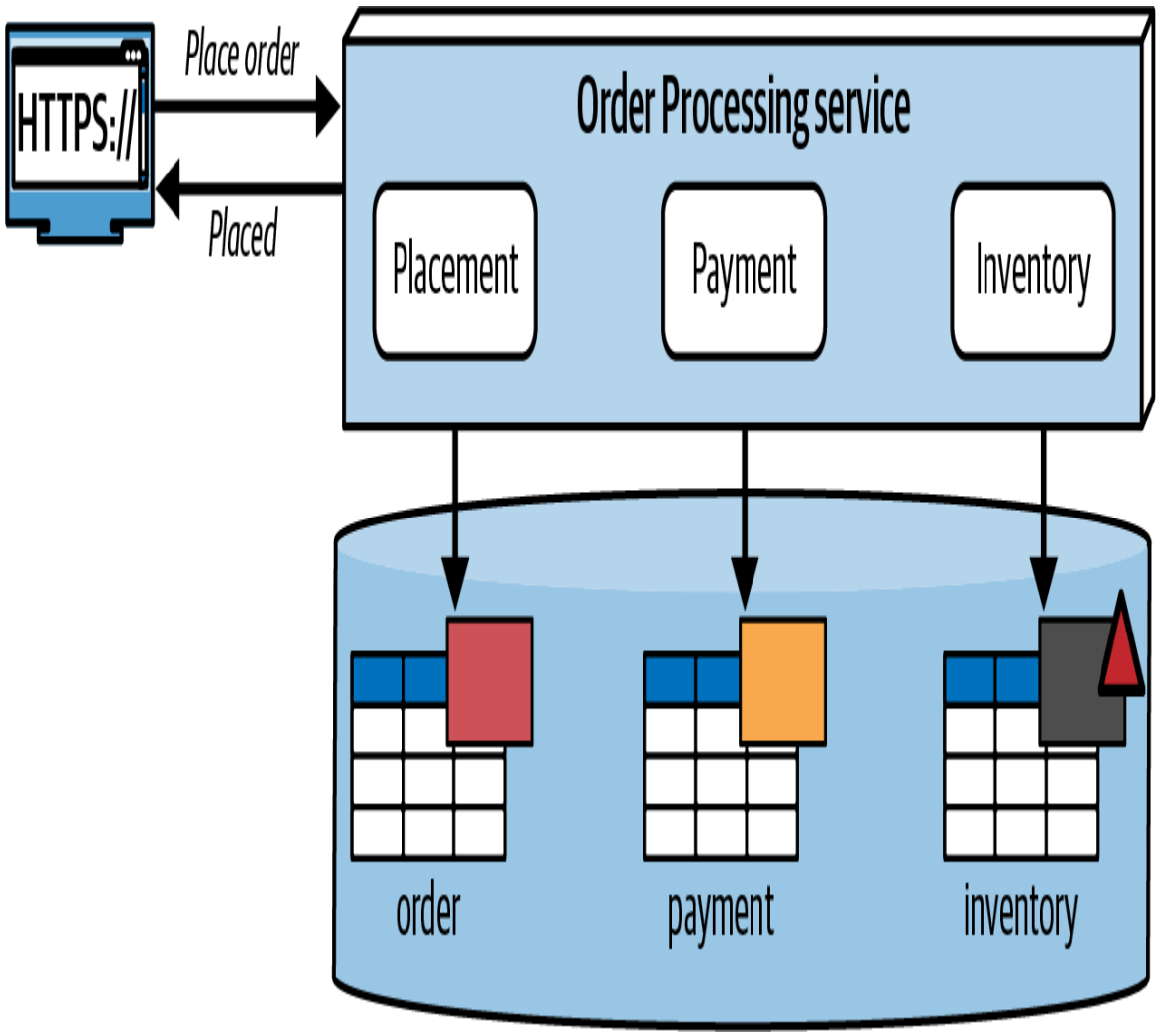


Figure 37-3. When a workflow is successful, atomicity ensures that all the parts of the workflow are updated correctly

In Figure 37-3, the user places an order, which updates the `order`, `payment`, and `inventory` tables and returns success. Thus, when everything works correctly, atomicity is invisible.

However, consider the case illustrated in Figure 37-4.

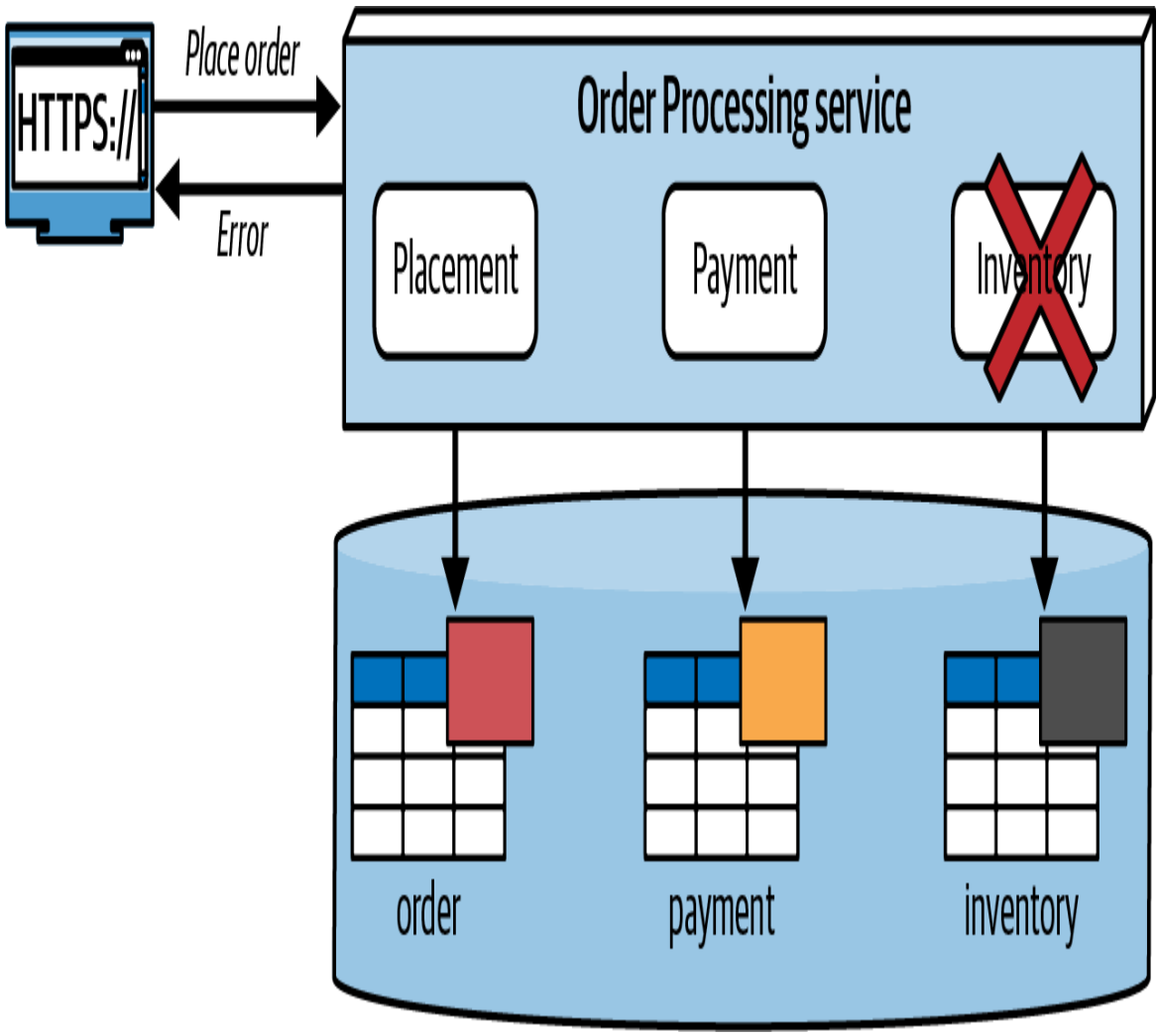


Figure 37-4. When an error occurs, atomicity ensures that everything returns to the previous state

In Figure 37-4, the workflow successfully added to `order` and `payment` but could not update `inventory`, for whatever reason. The relational database thus automatically restored the previous values for both `order` and `payment` before returning an error condition to the caller. Most critically, when the workflow completed, none of the updates from the attempted erroneous workflow survived. It's as if the attempted transaction never occurred.

When many architects refer to *transactions*, they often mean it as shorthand for this property of ACID. Atomicity is the most important of the four properties for nontechnical end users. However, atomicity isn't possible in microservices, as shown in Figure 37-5.

Each service can update its data independently, which means that the workflow can't be atomic. However, architects can *simulate* atomicity using the Compensating Updates pattern (see Chapter 11).

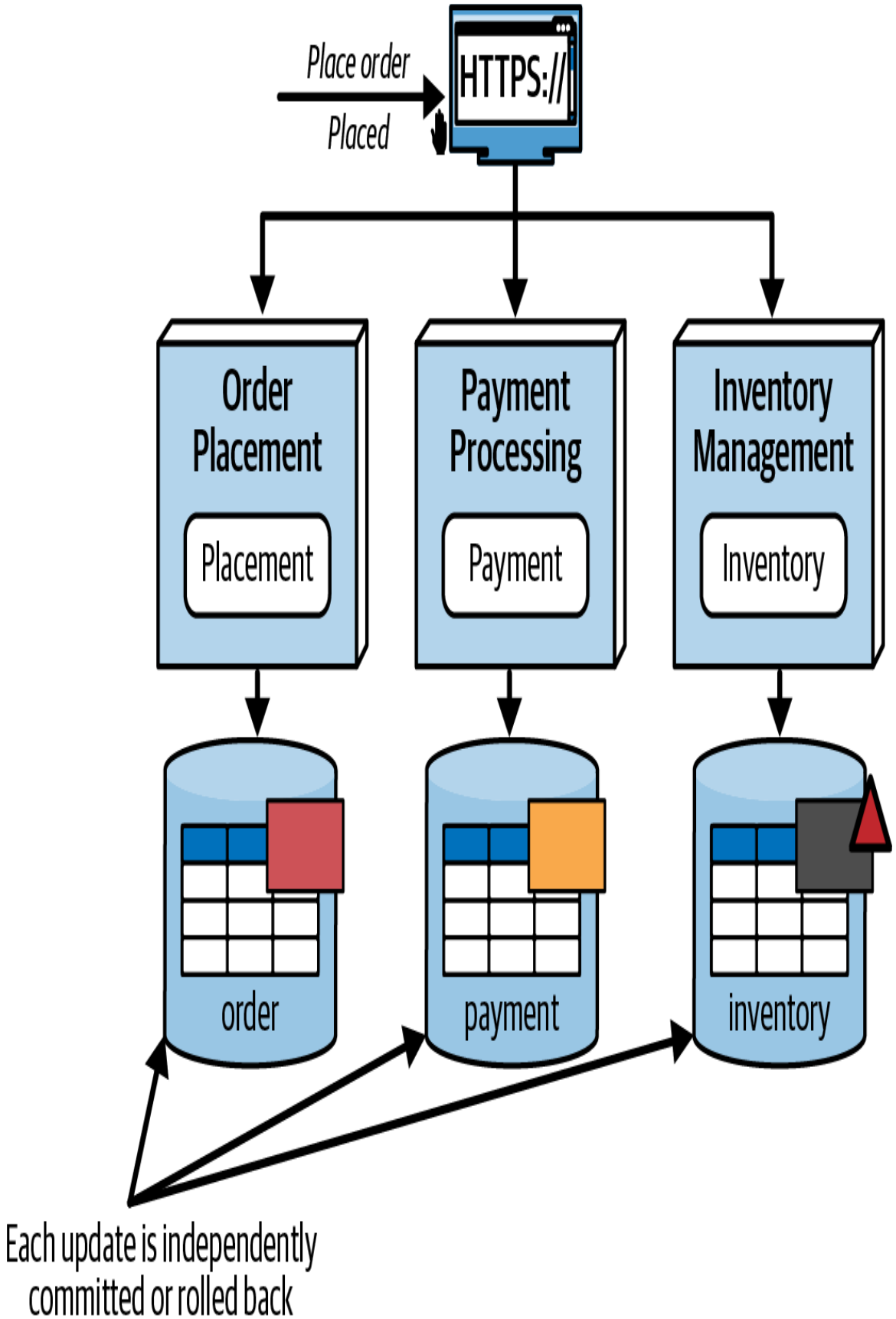

Place order
Placed
HTTPS://
Order Placement
Placement
Payment Processing
Payment
Inventory Management
Inventory
order
payment
inventory
Each update is independently committed or rolled back

Figure 37-5. Atomicity isn't possible across database instances in microservices

Consistency

Consistency is the ACID property that ensures that the relationships established by the database are honored, guaranteeing that data relationships are consistent between tables. For example, a proper `order_key` must exist in the `payment` table before a user can insert a `payment` record, as shown in Figure 37-6.

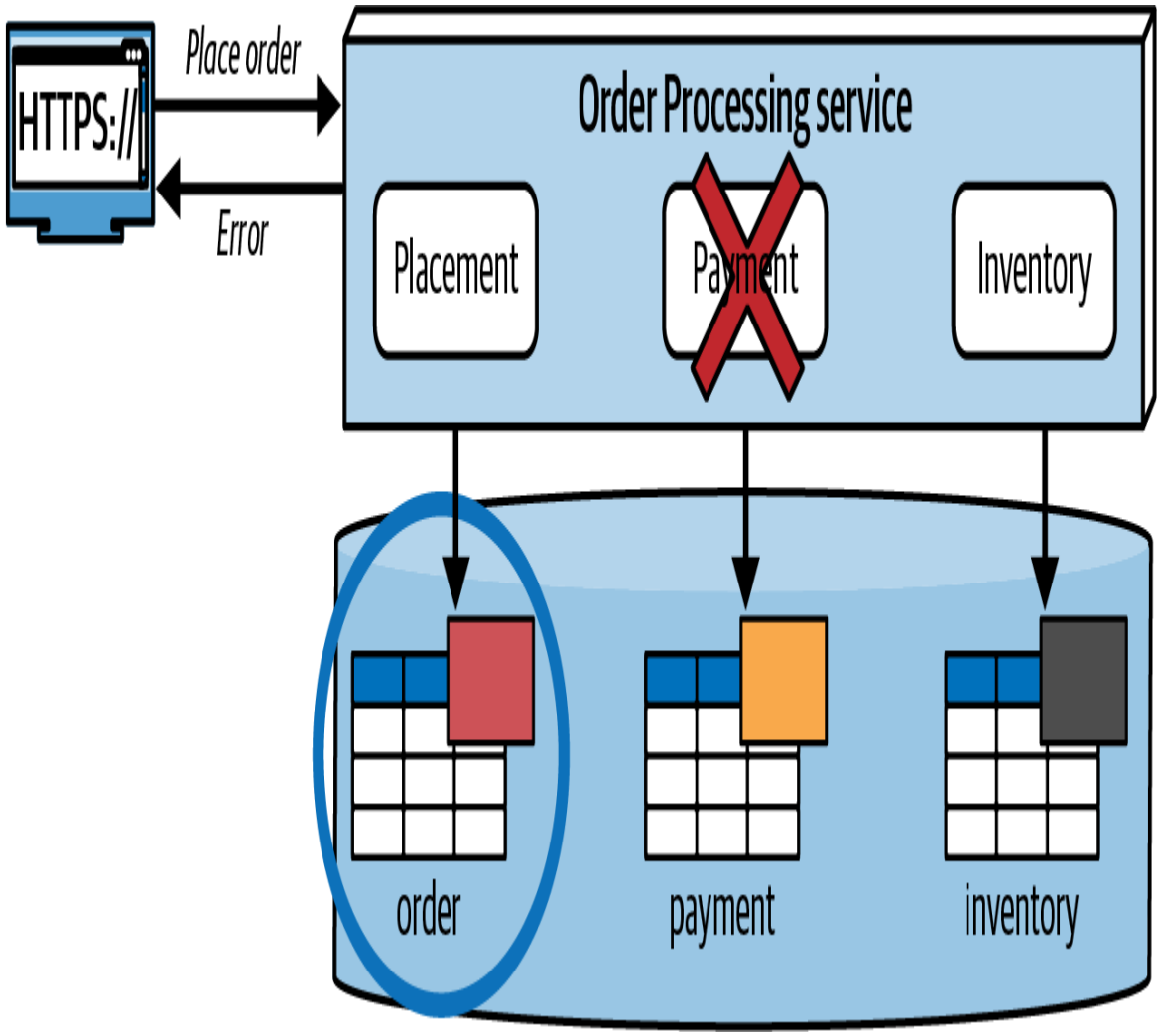


Figure 37-6. Consistency ensures that the required relationships exist

In Figure 37-6, when a user tries to insert a payment for an order that doesn't exist in the `order` table, the database rejects the payment (and inventory) updates. Consistency ensures that developers honor the

referential integrity rules defined at the data level and prevents updates that would violate those rules.

However, when data is split across multiple databases, there's no automatic integrity check, as shown in Figure 37-7.

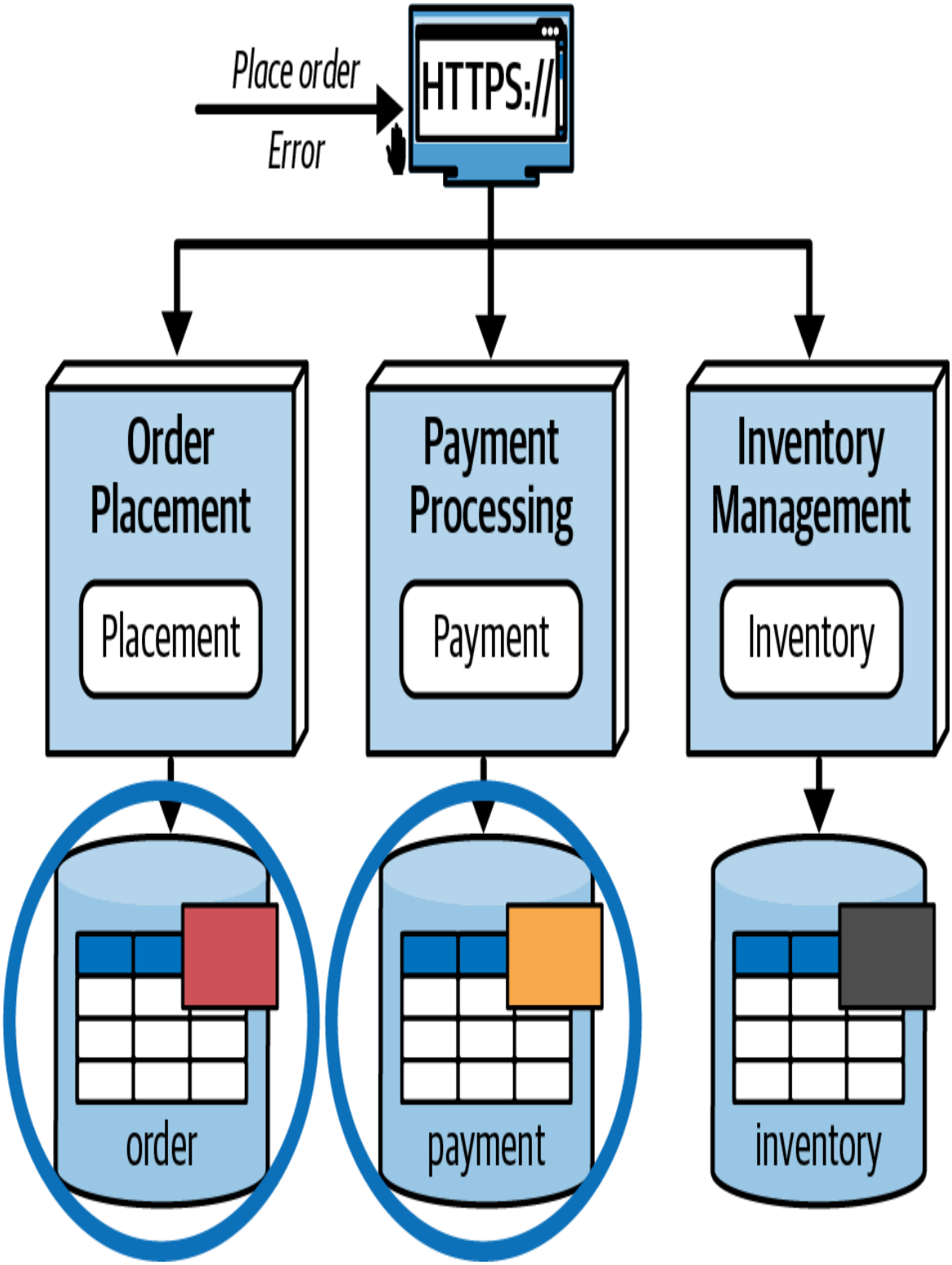


Figure 37-7. Microservices can't ensure consistency because each service updates its own data

In Figure 37-7, each service updates independently, preventing *automatic* referential integrity. Of course, the architect can code that behavior

themselves — nothing prevents an architect from designing the system shown in Figure 37-7, in which `PaymentProcessing` queries the `OrderPlacement` service to ensure that an order exists before issuing payment. The issue is that architects have become accustomed to receiving that behavior for free, and in microservices they have to implement it themselves.

> **EVENTUAL-ALL-THE-THINGS**
>
> While many architects have heard of eventual consistency, not all take the "eventual" philosophy into the other parts of ACID. For example, an architect can implement their own consistency using interservice calls. However, it doesn't happen within a single process like a database transaction does, making it eventual rather than transactional.

Of the four characteristics of ACID, *consistency* is the one that architects tend to fret over the most because it affects the integrity of domain data. In fact, the common alternative to *atomic* is *eventual consistency*, which is common in distributed architectures and covered by BASE.

Isolation

Isolation is the database property that mimics isolation from other interactions. Databases are multiuser, but have built-in protections for in-flight transactions. Consider the example shown in Figure 37-8.

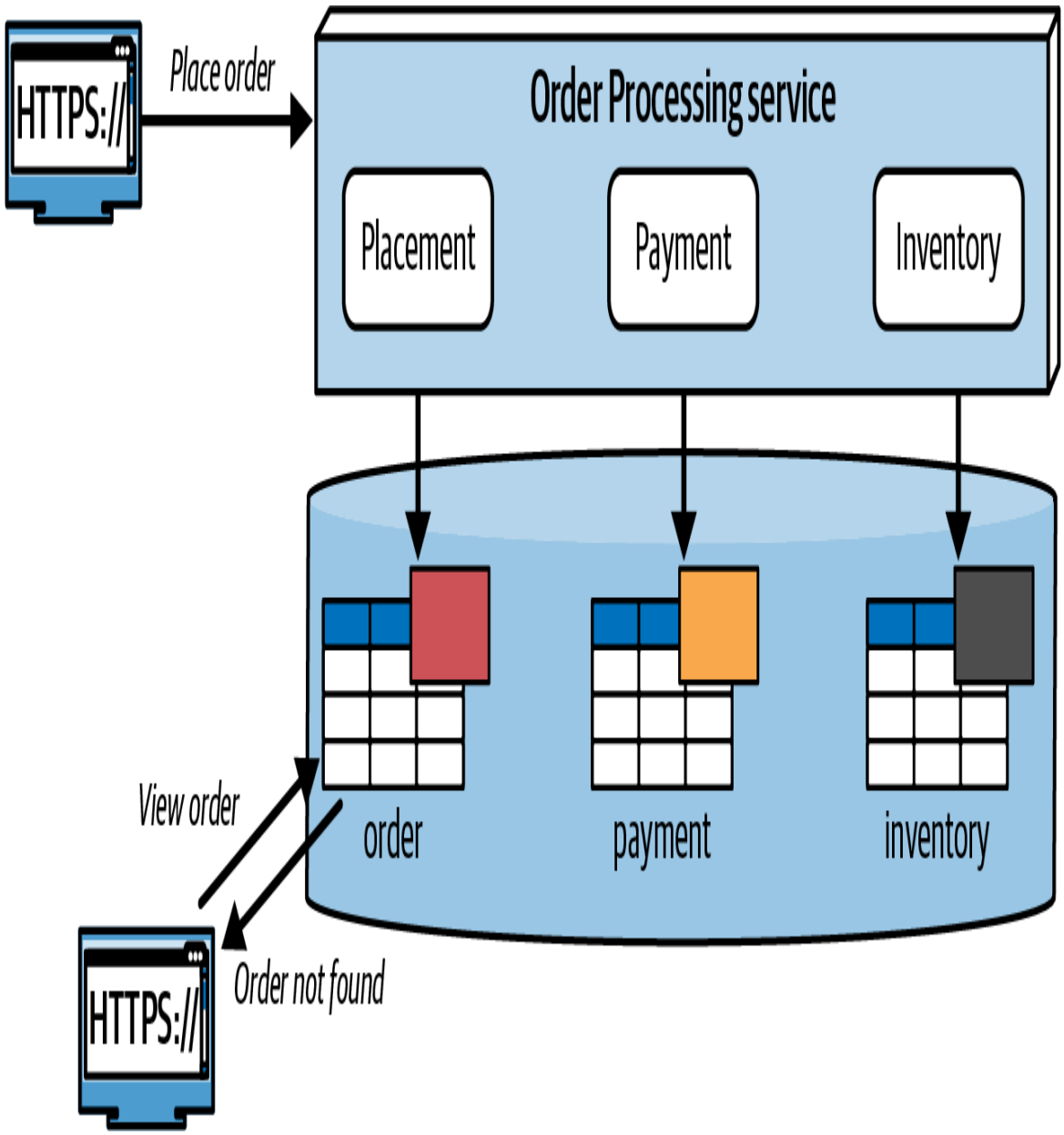


Figure 37-8. Isolation ensures that uncommitted changes are not visible outside the workflow

If another system queries the database for one of the values affected by an order placement update during the order-placement workflow, it will not see those changes until the workflow has completed and the transactions have been committed. Thus, in Figure 37-8, querying for the new order from the other system will fail. In a microservices architecture, architects can't rely on isolation. In Figure 37-9, as soon as the `OrderPlacement` service adds the order, it will be visible to any other services that query for it, even if the workflow is still executing.

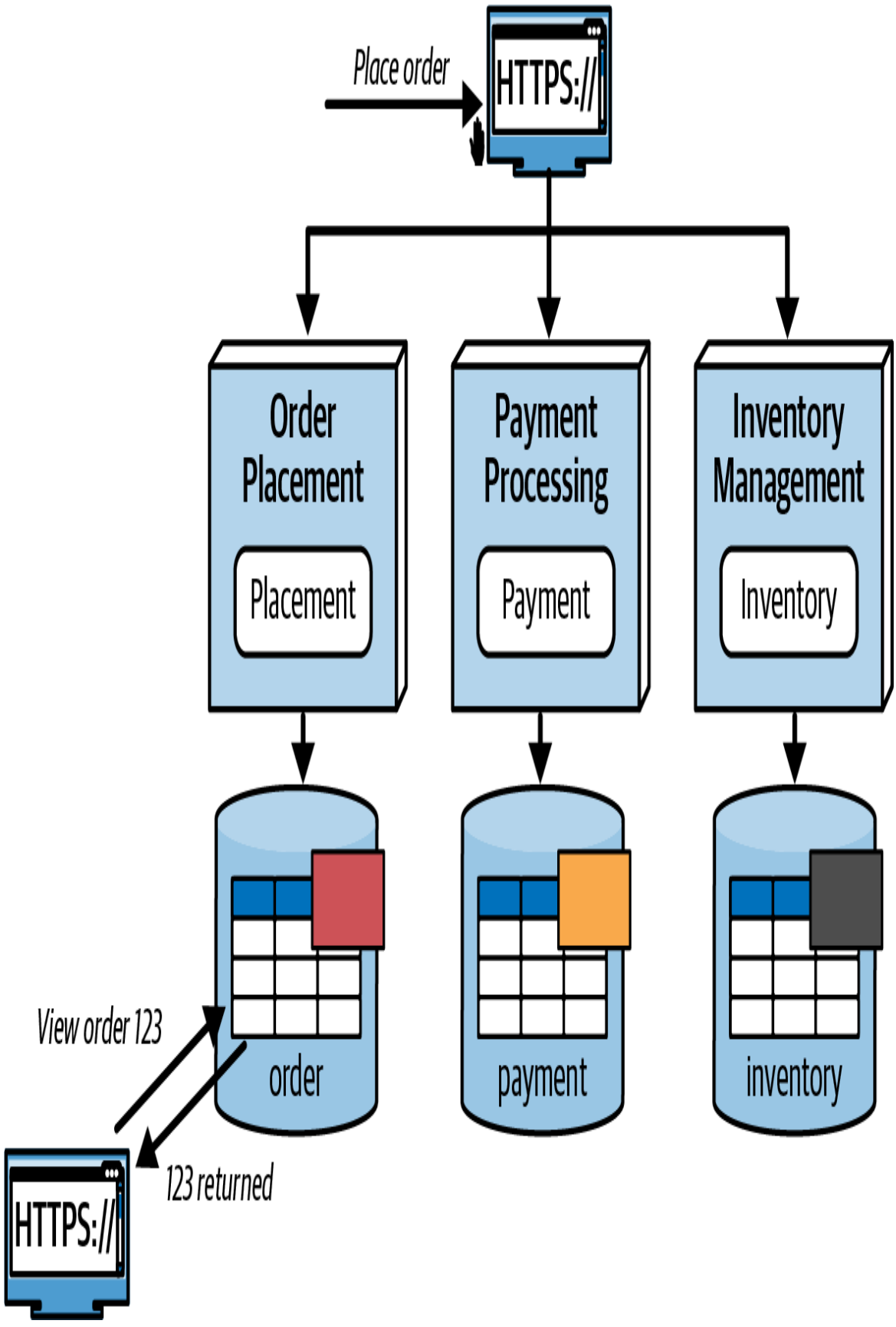

Place order
HTTPS://
Order Placement
Placement
Payment Processing
Payment
Inventory Management
Inventory
order
payment
inventory
View order 123
123 returned
HTTPS://

Figure 37-9. Microservices can't guarantee isolation beyond a single service

Isolation is a fundamental difference between transactional and distributed architectures, and it's so hard to replicate that most architects just find ways to deal with it.

Durability

Durability ensures that workflow changes are permanent once a transaction has been committed, as shown in Figure 37-10. Once the transaction completes, no data will be lost in the event of a system outage.

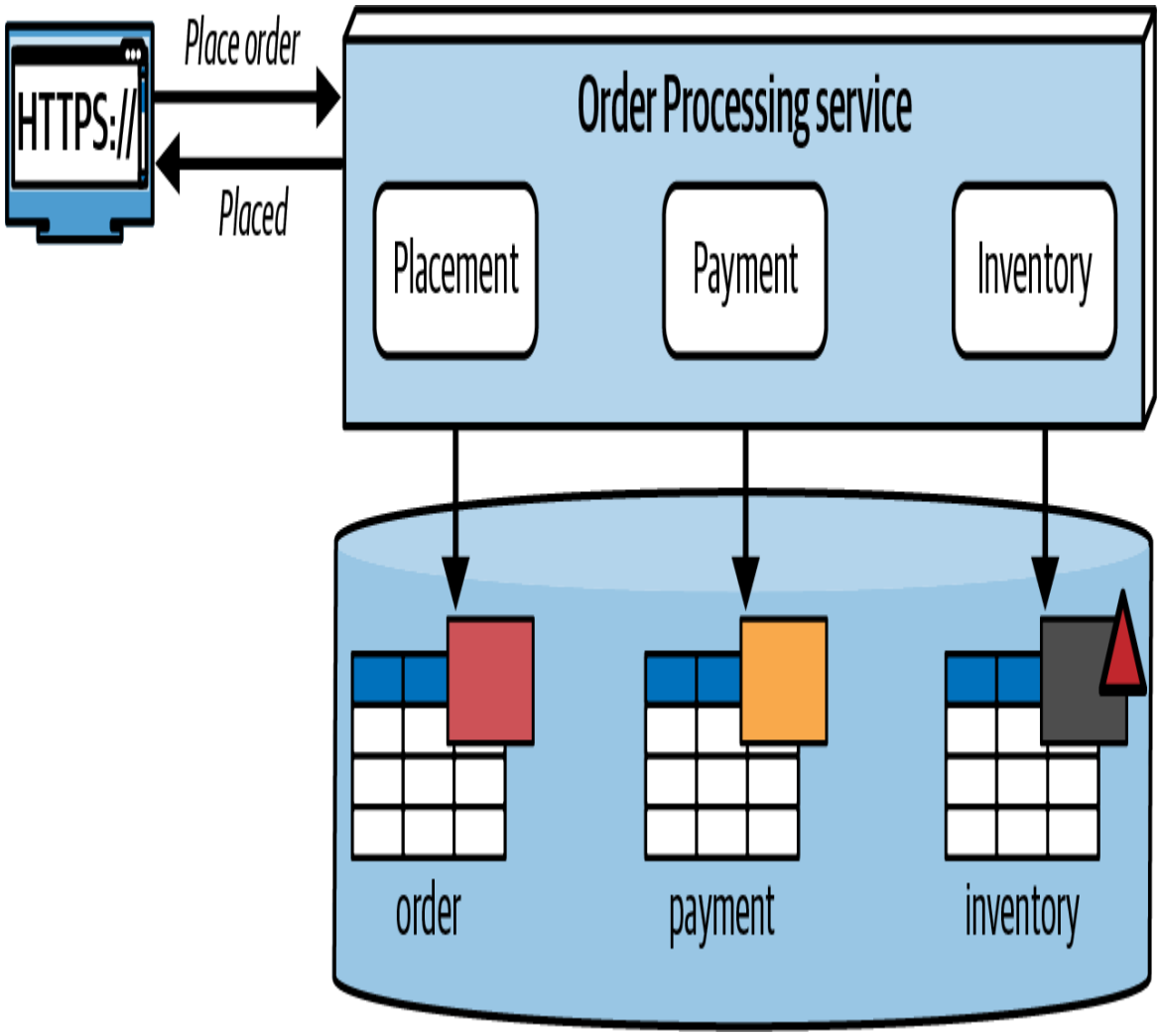


Figure 37-10. Durability ensures that changes are permanent when a transaction concludes

In microservices and other distributed architectures, durability is determined at the service level rather than the database or system level. Thus, the only guarantee in those architectures is durability at the level of a single service, as shown in Figure 37-11.

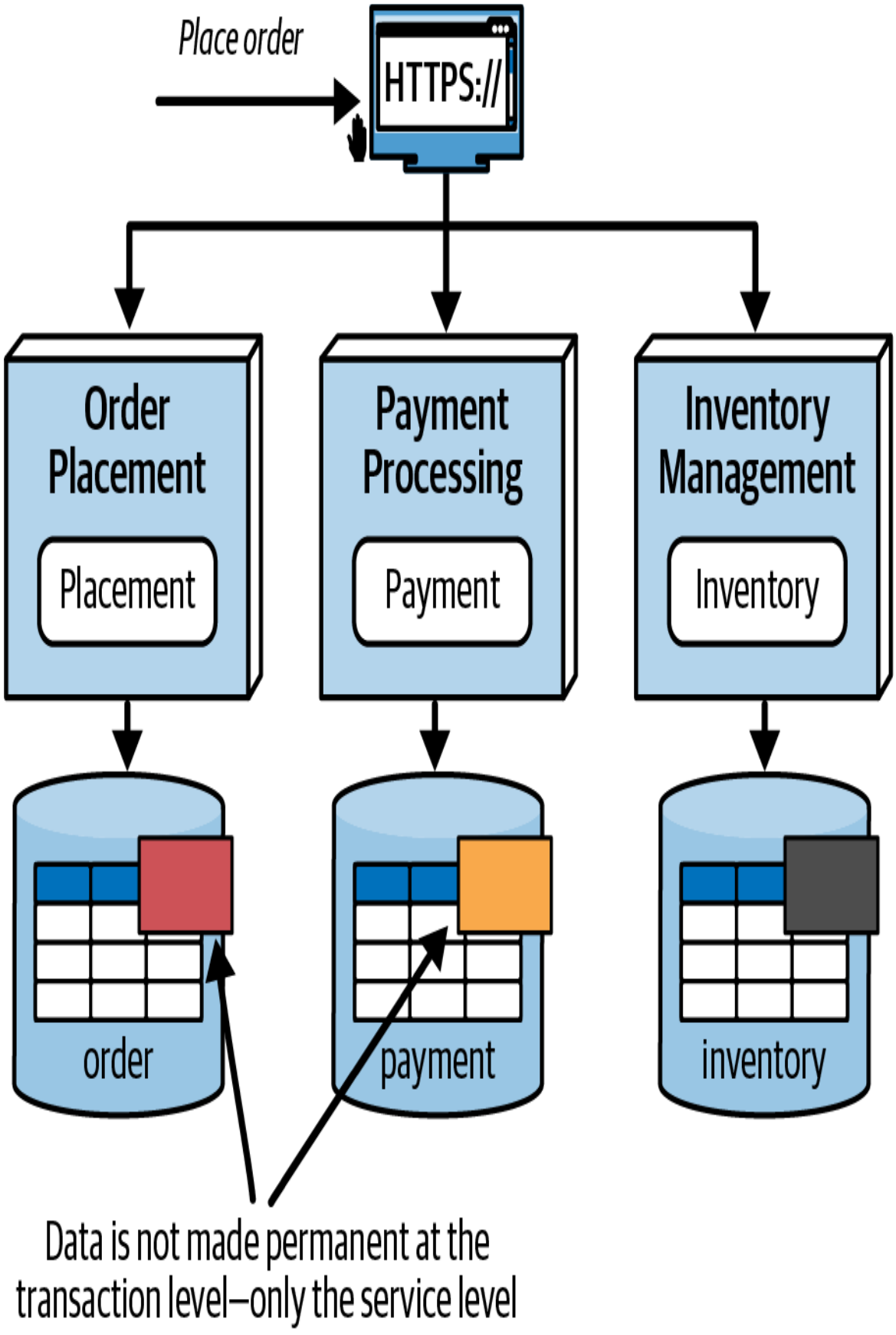

Place order
HTTPS://
Order Placement
Placement
Payment Processing
Payment
Inventory Management
Inventory
order
payment
inventory
Data is not made permanent at the transaction level–only the service level

Figure 37-11. Microservices are only durable at the service level

Any one of the services shown in Figure 37-11 might be unavailable during the workflow, potentially leaving the data in an inconsistent state. If architects can't rely on ACID for distributed architectures, how can they manage transactional workflows? That's where the Compensating Updates pattern comes in: it gives architects a way to simulate the behavior of a single relational database.

Faking Parts of ACID

ACID transactions aren't possible in distributed architectures for the reasons outlined above. However, users are accustomed to systems that are (or seem to be) transactional and are loath to surrender that feature. Thus, many architects rely on patterns to implement their own pieces of ACID:

Atomic

> Architects can't rely on the atomic behavior of a single database, so they must write code within a distributed workflow to mimic the effect. Often, they relax strict atomic behavior and rely on eventual consistency.

Consistent

> Data consistency is important to many problem domains, encouraging architects to implement that behavior through patterns like the Compensating Updates pattern.

Isolated

> *Isolation* is hard to replicate because, by definition, transactions are scoped at the service level in microservices, making workflow-level isolation difficult.

Durable

> If durability is a big concern, architects can use durable message queues, redundancy, and other mechanisms to improve it.

Two of the capabilities we track for virtually every pattern in this book are *data consistency* and *data integrity*, often consolidated into the composite *atomicity*. We discuss these definitions and their nuances in "Atomicity". Most architects in microservices try to avoid atomicity entirely and model eventual consistency instead.

Architecture Quantum

The concept of a *software architecture quantum*, originally defined in the book *Building Evolutionary Architectures*, is a way to analyze the overall evolvability of different architecture styles. Many science-literate architects know the concept of a quantum from physics, where it refers to the smallest possible amount of something, usually energy. The word *quantum* derives from Latin, meaning "how great" or "how much." In general usage, it tends to mean "small, unbreakable thing," which is how we define an *architecture quantum*. Another common informal definition of *architecture quantum* is "the smallest part of the system that runs independently." For example, microservices often form multiple architecture quanta, exemplifying this definition: a service can run independently within the architecture, including its own data and other dependencies.

Defined

An *architecture quantum* establishes the scope for a set of architectural capabilities. It features:

- Independent deployment from other parts of the architecture
- High functional cohesion
- Three types of coupling:
  - Static
  - Dynamic
  - Atomic

We dive into the details of each definition element as follows:

Establishes the scope for a set of architectural capabilities

> Architects use architecture quantum as a boundary delineating a set of architectural capabilities, particularly operational ones. Because it's independently deployable and has high functional cohesion (discussed next), the architecture quantum provides a useful measure of modularity because it includes both logical architecture (components and dependencies) and physical architecture (services and databases). This use of the architecture quantum is critically important, because it clearly defines a scope for a set of capabilities, which many modern practices (such as agentic code generation) need.

Independently deployable

> An architecture quantum includes all the necessary components to function independently from other parts of the architecture. For example, if an application uses a database, that database is part of the quantum, because the system won't function without it. This requirement means that virtually all legacy systems that are deployed using a single database, by definition, form a quantum of one. However, in the microservices architecture style, each service includes its own database (part of the *bounded context* driving philosophy), which creates multiple quanta within that architecture because each service has its own architectural capabilities scope.

High functional cohesion

> *Cohesion*, in component design, refers to how unified the contained code is in its purpose. For example, a `Customer` component, with properties and methods all pertaining to a *Customer* entity, exhibits high cohesion. A `Utility` component with a random collection of miscellaneous methods would not. High functional cohesion implies that an architecture quantum does something purposeful. This distinction matters little in a traditional monolithic application with a single database, where the cohesion is essentially the entire system.

However, in distributed architectures, like event-driven or microservices, architects are more likely to design each service to match a single workflow (a *bounded context*, as described in Domain-Driven Design), so that service would exhibit high functional cohesion.

Architecture Quantum Coupling

Beyond delineating a scope for architectural capabilities, the quantum concept also provides a useful framework for defining the various types of coupling within distributed architectures.

Three types of quantum coupling

Most architects talk about *coupling* a lot — it's one of the dominating features of architectural design. Decomposing the term *coupling* to its constituent parts allows architects to fine-tune just *how* decoupled their architectures really are. In addition to scoping capabilities, the architecture quantum concept also defines three different types of coupling: static, dynamic, and atomic.

Static coupling

In a distributed architecture, *static coupling* refers to dependencies between components that are fixed at compile-time or design-time, while *dynamic coupling* refers to dependencies that are established and change at runtime. The key difference lies in when and how the relationship between components is formed.

Static coupling represents a more rigid, predefined connection between services. It's determined before the system is even running and is typically based on source code or configuration files. This type of coupling can be beneficial for its predictability but can also create a brittle system that's hard to change. We call an architecture *brittle* when a single implementation change can cause unexpected rippling side effects that break many other (ostensibly unrelated) things.

For example, say a microservice called `OrderService` needs to check inventory before placing an order. If `OrderService` has a hardcoded URL or a client library that directly points to the `InventoryService`, it forms static coupling between the two. If the `InventoryService`'s API changes, the `OrderService`'s code must be updated, recompiled, and redeployed to reflect the changes. This creates a cascade of changes that makes the system less agile.

In general, tight coupling is desirable when high cohesion is required, such as within a service or subsystem. The broader the scope of the system's coupling, the more loose coupling will help to make the architecture less brittle. For example, an architect might rename a field in a service call from `State` to `StateCode`, thinking it will only affect one caller, only to discover that many other dependencies have just broken unexpectedly.

SCOPE OF COUPLING

Architects can allow tighter coupling for narrower scopes; the broader the scope, the looser the coupling should be.

Static coupling refers to the "wiring" of an architecture — how its services depend upon one another. Two services are part of the same architecture quantum if they both depend on the same coupling point. For example, let's say that two microservices, `Catalog` and `Shipping`, need to share address information, so they both create a dependency to a shared component. Because both services are coupled to that dependency, they're part of the same architecture quantum. In Figure 37-12, the first two services are statically coupled together because they share a component. If the component changes, both services must change to accommodate that change.

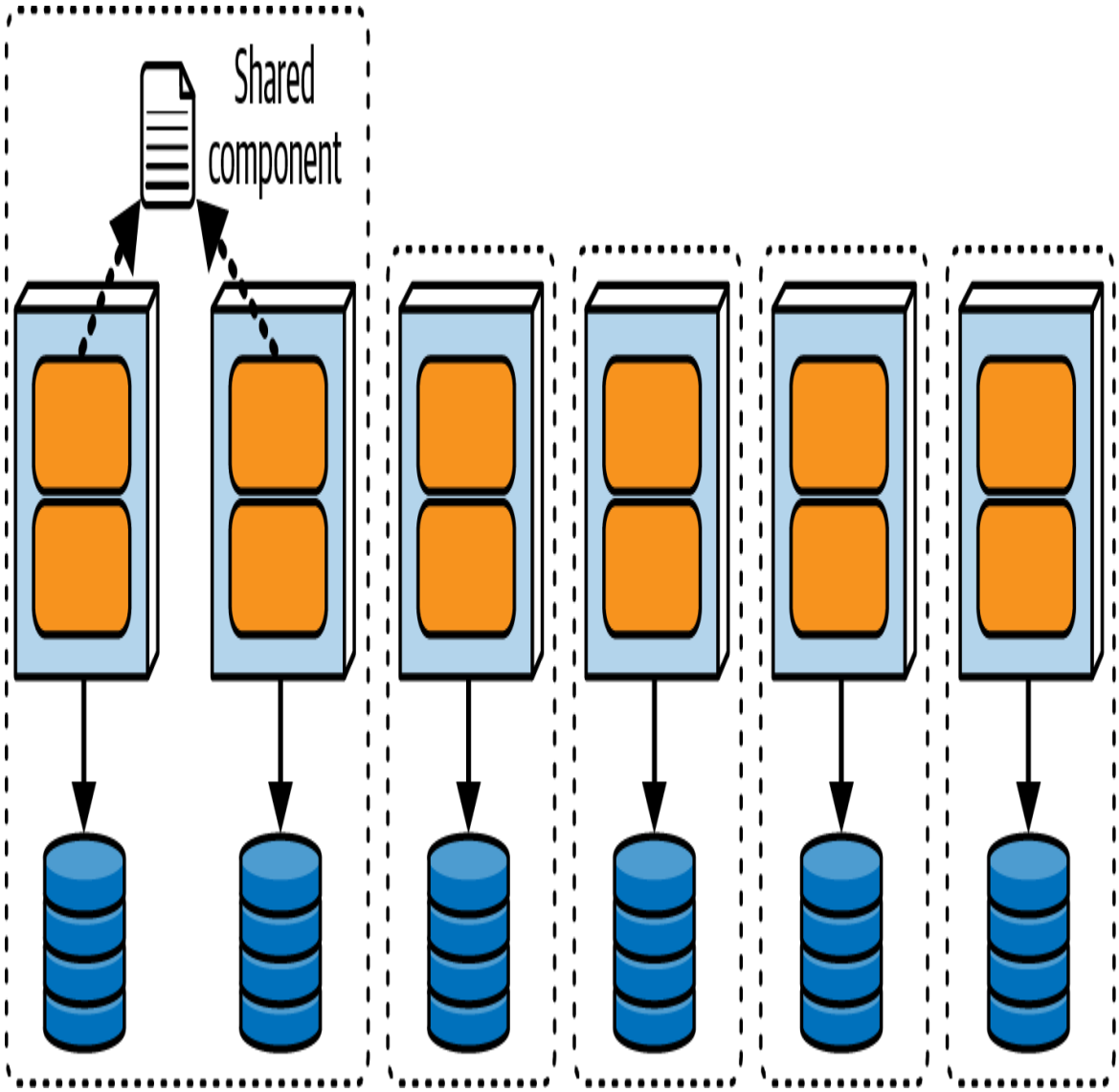


Figure 37-12. Static coupling between services creates quantum coupling

Here's an easy way to think about coupling in software architecture: two things are coupled if changing one might break the other. Static coupling defines the scope dependencies in an architecture. For example, if several services use the same relational database, they're part of the same quantum.

Dynamic coupling

Dynamic coupling describes the forces involved when architecture quanta must communicate with each other. For example, when two services are running, they must communicate to form workflows and perform tasks across the system. Architects must consider the trade-offs when

communicating between services in a distributed architecture, as highlighted in many of our patterns. In particular, they must decide between synchronous or asynchronous communication.

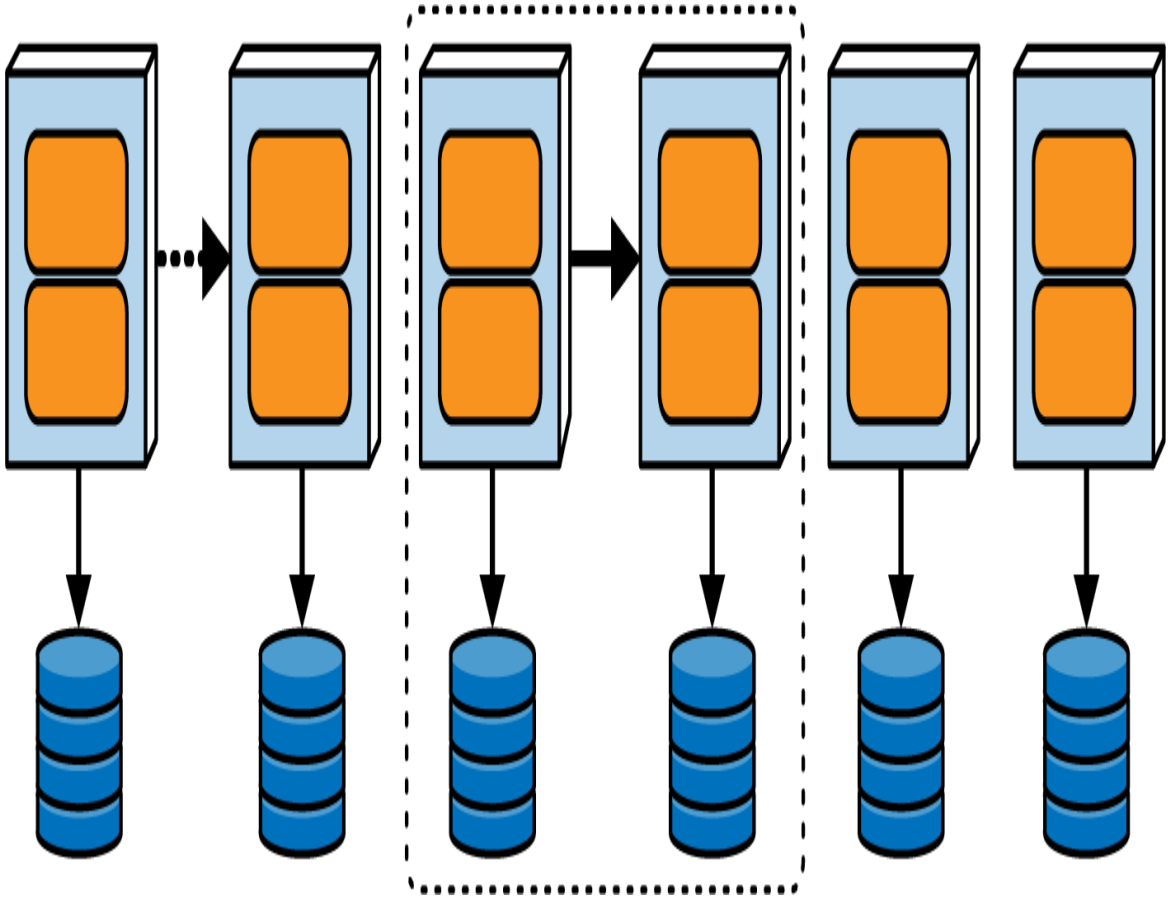


Figure 37-13. Synchronous communication creates operational coupling between services

In Figure 37-13, two services that communicate synchronously (typically represented by solid arrows) temporarily bind their operational architectural capabilities together. For example, consider two services, `Catalog` and `Payment`. `Catalog` scales to 1,000 users/sec but `Payment` only scales to 200 users/sec. When `Catalog` makes a synchronous call to `Payment`, blocking to wait for the return, suddenly the scalability of `Catalog` is the same as that of `Payment`, at least for the duration of that call.

DYNAMIC COUPLING'S IMPACT ON OPERATIONAL CAPABILITIES

In general, when two or more otherwise decoupled services call each other synchronously, the operational capabilities of both sink to match the lowest value of the set.

When architects say they want a "decoupled architecture," they're mostly talking about static coupling. However, to really achieve the benefits of decoupling, communication should be as decoupled as well.

DYNAMIC QUANTUM ENTANGLEMENT

We chose to use the term *quantum* because it means "small, unbreakable thing." However, we didn't anticipate further analysis revealing that, just as in subatomic physics, architecture quanta can become entangled along the dimension of their operational capabilities when a more capable service calls a less capable one. We didn't plan this happy coincidence, but it makes the metaphor even deeper.

Atomic coupling

Atomic coupling is by far the most difficult kind of couple to handle in distributed architectures. It occurs when a workflow needs transactional-style coordination across multiple service calls, as illustrated in Figure 37-14.

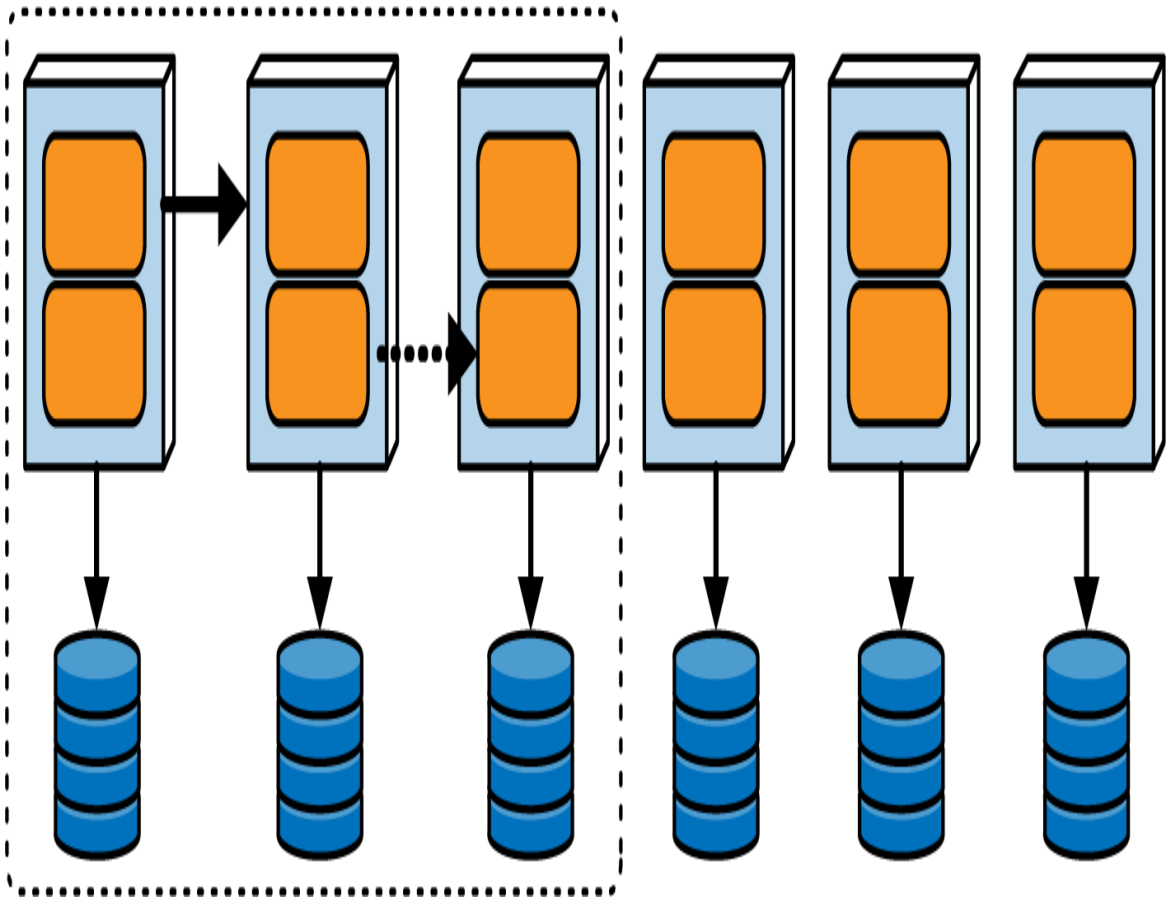


Figure 37-14. Transactional (or atomic) coupling occurs when services must collaborate in all-or-nothing workflows

In Figure 37-14, the three leftmost services collaborate in a workflow where all the data items are in sync (consistency) and correct (integrity). Nondistributed architectures rely on databases to handle these capabilities; architects must handle them when the database can't. Throughout the book, we use the capability *atomicity* (see "Atomicity") to lump together data consistency and data integrity. When they differ, we call it out, but they overlap so commonly that we use *atomicity* as shorthand for both.

Composite types of coupling

Two additional types of coupling manifest with the architecture quantum context, each combining some features of the three core elements of the definition.

Contract coupling

Any time two software artifacts communicate, a contract is involved. We use the term *contract* to identify the information and semantics developers pass from one endpoint to another, separate from the protocol they use. The term *protocol* describes an implementation for both the contract information and the technical details of transport. Examples include an alphabet soup of acronyms, such as SOAP, REST, gRPC, XMLRPC, and many more. Generally, we distinguish the underlying architectural concern from the tools that implement patterns. This distinction is useful because most tools and frameworks implement a hodgepodge of different patterns, making isolation more difficult.

Contracts combine static and dynamic coupling: they're defined as static coupling dependencies, but only manifest dynamically at runtime.

Contracts are important for architecture patterns because they create coupling points between two artifacts. That is, two software artifacts exhibit contract coupling and form an architecture quantum that combines static and dynamic coupling anytime they must agree on passed data elements for success, as illustrated in Figure 37-15.

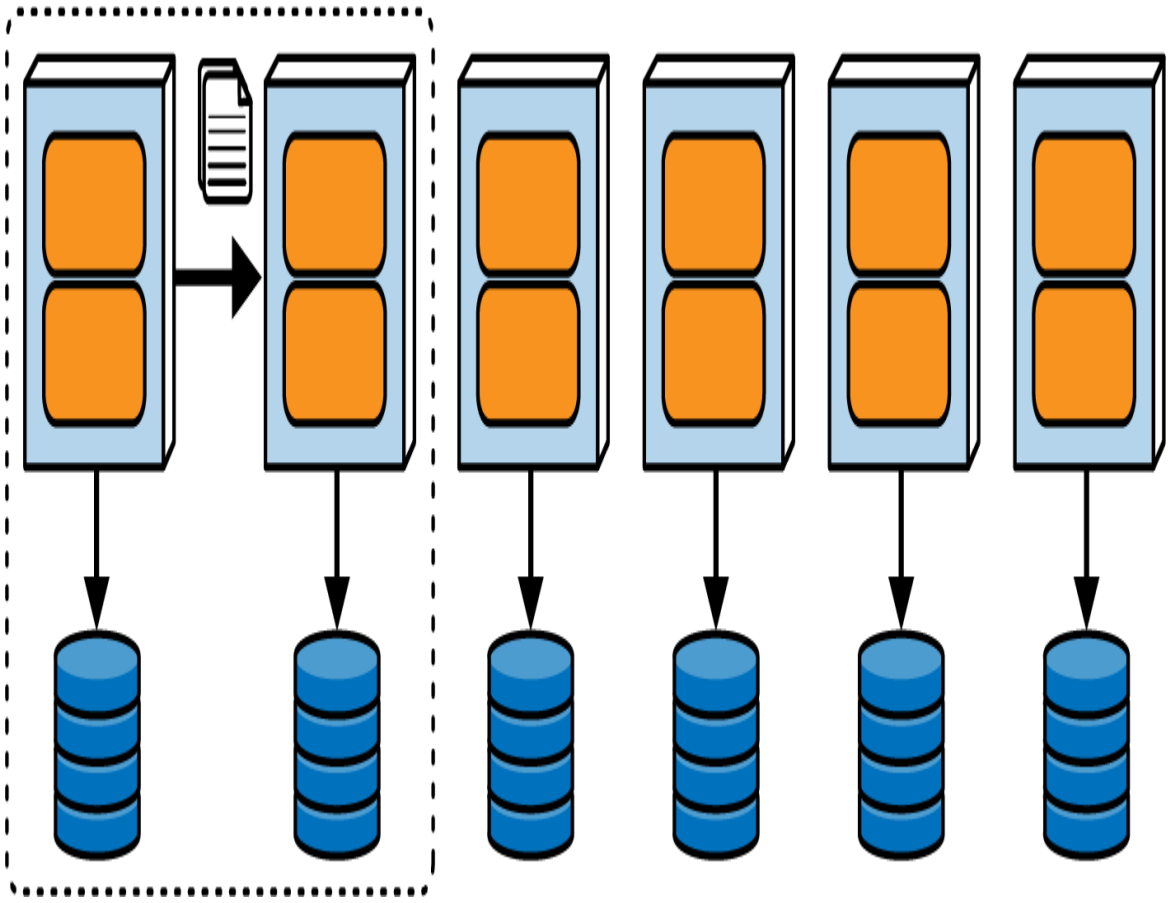


Figure 37-15. Contract coupling between two services: if the contract changes, both services might have to change based on how strict the contract is

In Figure 37-15, two services that share a common contract are coupled together. The degree of contract coupling depends on how strict or permissive the contract is. For example, a strict contract includes tightly bound metadata about the parameters, types, and sometimes other metadata such as minimums or maximums. Permissive contracts tend toward simple name/value pairs in JSON.

Contract coupling refers to the degree of coupling imposed by the type of contract. For example, many RPC protocols (such as SOAP, gRPC, and so on) mimic strongly typed method calls, including many (or all) required fields. Overusing strict contracts often inadvertently leads to the Stamp Coupling antipattern, discussed in Chapter 31.

Temporal coupling

Temporal coupling is a special case that combines static, dynamic, and contract coupling with a time element. Temporal coupling most often arises when architects use mechanisms like caches as an efficient way to access data they don't own.

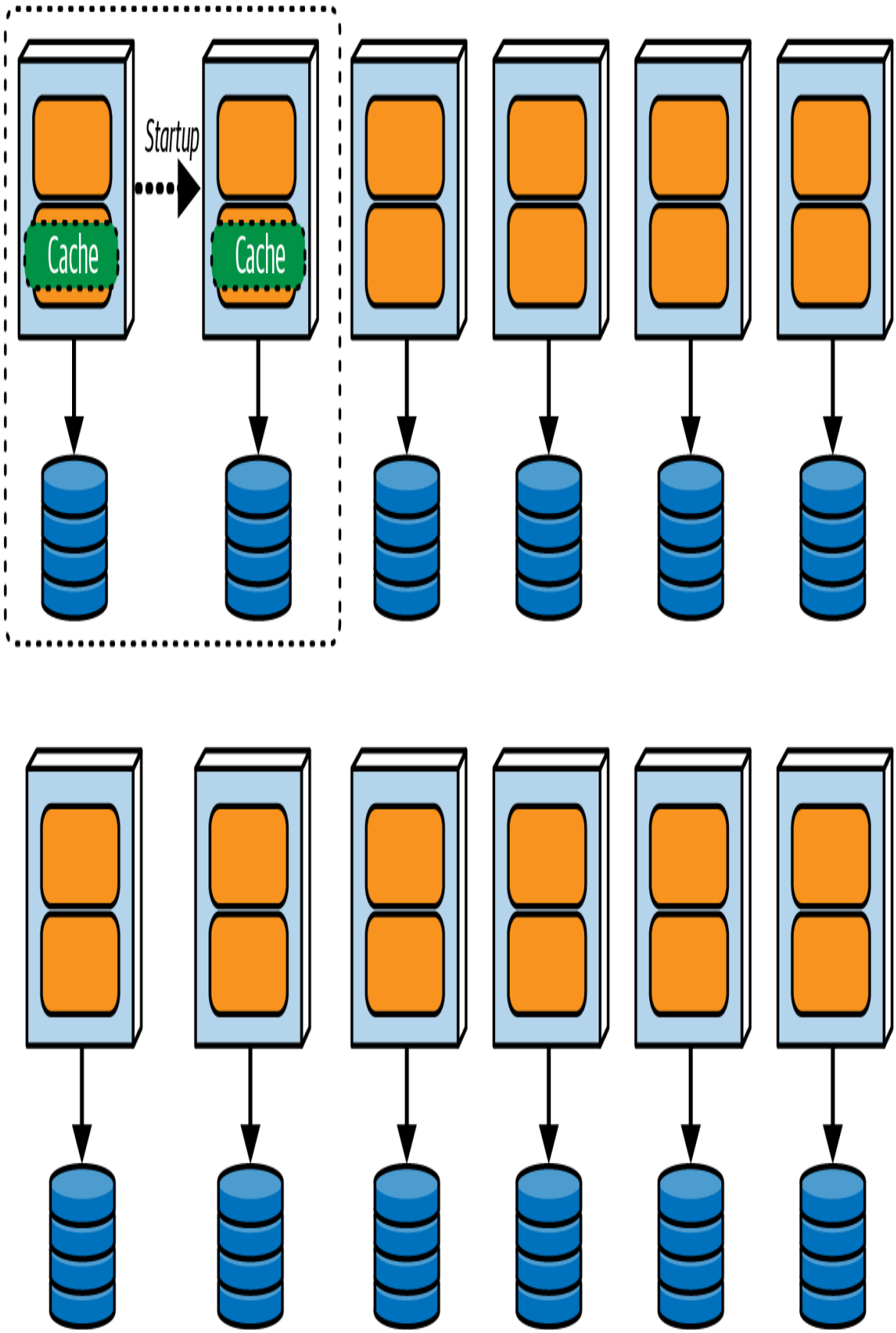

Startup
Cache
Cache

Figure 37-16. Temporal coupling occurs when loading caches at startup

In Figure 37-16, at startup, the leftmost service blocks and waits for the coupled service to provide (usually) read-only data to fill its cache. After that, it operates in a decoupled manner, except for sporadic, generally asynchronous updates to the cache.

A number of patterns in this book address this type of coupling, including Compensating Updates (see Chapter 11) and Service Sharding (found in Chapter 30). In patterns influenced by architecture quantum, we explain the pattern's impact on each type of coupling.

When architects casually say they want a "decoupled architecture," they're generally talking about static coupling. However, to build truly decoupled systems, architects need to pay attention to all five types of coupling.

Events Versus Messages

Event-driven architecture leverages events to pass and process information. But is an event really that different from a message? Turns out, yes, it really is. Here are the ways they differ:

Broadcasting versus commanding

> An *event* broadcasts to other event processors that something has already happened: "I placed an order." A *message*, however, is more of a command or query, such as "Apply the payment for this order" or "Give the shipping options for this order." This is not a subtle difference. *Event processing* is about reacting to something that has already happened, whereas a message describes something that needs to be done. In our example, "I placed an order" is clearly an event, because it does not describe what kind of processing needs to occur as a message would.

Response expected?

The second main difference between an event and a message is that an event typically does not require a response from the receiver, whereas a message usually does. This reduces back-and-forth communication between event processors, further decoupling them from one another.

Broadcast versus point-to-point

An event is typically broadcast to multiple event processors, whereas a message is almost always directed to just one. In a simple order processing example, several event processors (such as `Payment`, `Fulfillment`, and `Inventory`) are interested in and respond to the `order created` event. However, only one event processor (the `Payment` processor) responds to the message "Apply payment." Events typically use a *publish-and-subscribe* form of communication (one-to-many, sometimes also called *pub-sub*), whereas messages typically use a *point-to-point* (one-to-one) form.

Communication channel

A final difference between events and messages is the physical artifact that represents the communication channel. Events use a *topic*, *stream*, or *notification service* so that multiple event processors can subscribe to the channel and listen for the event. Messaging typically uses a *queue* or *messaging service* to guarantee that only one type of event processor will receive that message.

Quantum scope

An important aspect of the architecture quantum definition is the concept of static coupling to dependencies that allow the quantum to act as a standalone part of the architecture. That includes artifacts, such as the service boundary and database, but also required infrastructure, like the message broker.

Generally, in architectures that use *events*, the broker is part of the *sender*'s architecture quantum, as illustrated in Figure 37-17.

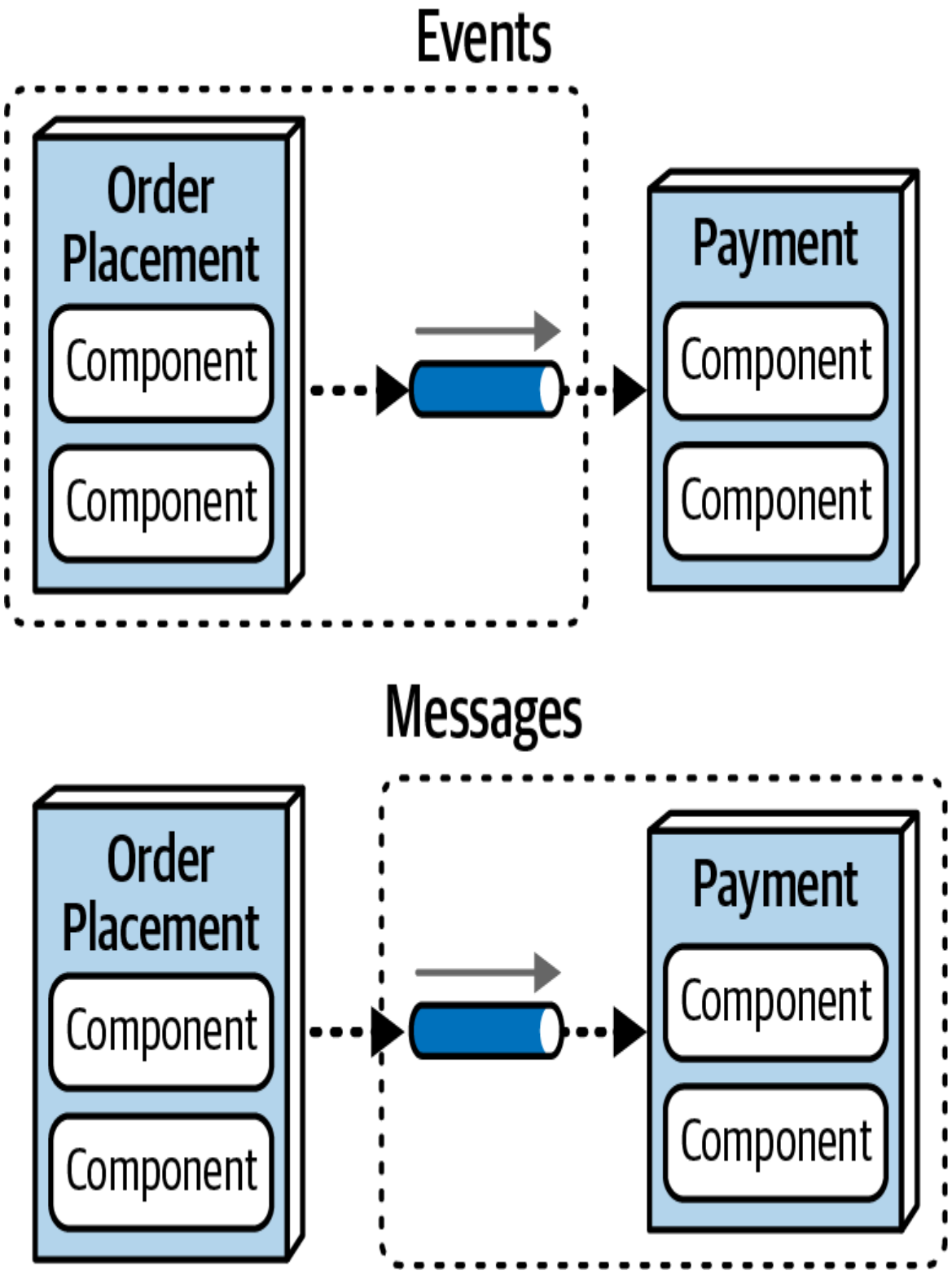


Figure 37-17. The static-coupling part of the architecture-quantum definition suggests infrastructure ownership

In Figure 37-17, using events (on the left), the event queue belongs to the `OrderPlacement` ecosystem. However, in the message-based version of this workflow, the message queue is typically part of the architecture quantum of the *receiver*, as shown in the righthand workflow in Figure 37-17. This is a subtle but important distinction that leads to a host of considerations around reliability, availability, coupling, and other factors. For an example of the impact of this distinction, see the Domain-Broker pattern in Chapter 15.

Event-driven architecture mostly uses events (hence its name), but it can use messages on occasion, such as requesting data from another event processor; an example appears in the Choreography pattern in Chapter 9. Messages are typically used in mediated or orchestrated architectures; for an example, see the Orchestration pattern in Chapter 27.

Which of the following choices are events and which are messages?

- "Adventurous Air Flight 6557, turn left, heading 230 degrees."
- "In other news, a cold front has moved into the area."
- "OK, class, turn to page 145 in your workbooks."
- "Hi, everyone! Sorry I'm late for the meeting."

Let's look at these one by one:

"Adventurous Air Flight 6557, turn left, heading 230 degrees."

> This is a *message* because it's a command (something that needs to be done) and because it's directed toward one target, the pilot — even though several other pilots may hear the message.

"In other news, a cold front has moved into the area."

> This is an *event*: it's being broadcast to multiple people, it describes something that has already happened, and the news broadcaster is not expecting a reply. (However, not all messages require a reply.)

"OK, class, turn to page 145 in your workbooks."

This one is a little tricky. Turns out this is a *message*, even though it's being broadcast to multiple students. It's a command to do something, not an announcement of something that has already happened (which would make it an event). This illustrates an important point: broadcasting a command (such as turning to page 145 in the workbooks) through a publish-and-subscribe channel doesn't turn it into an event.

"Hi, everyone! Sorry I'm late for the meeting."

This is an *event*, because this person being late for the meeting has already happened. It's also being broadcast to multiple people, and no response is expected.

MICROSERVICES VERSUS EVENT-DRIVEN ARCHITECTURE

Architects are often confused by the dual nature of events and messages in distributed architecture styles like microservices and event-driven architectures (EDAs). (EDAs predate microservices by a couple of decades — the modern concept of microservices had to wait for Eric Evans to envision Domain-Driven Design [DDD], which inspired the coupling and data characteristics of microservices.) Thus, when we talk about *event-driven architectures* as an architectural style, we're describing a system that uses event or message queues as its primary structural elements, typically with a single relational database for storage. The microservices style, on the other hand, adheres closely to DDD's bounded-context philosophy, which requires high degrees of decoupling and data fidelity at the service level — each service owns its own data.

The confusion arises because of the distinction between architecture styles and patterns, which we made in Chapter 1 in "Styles Versus Patterns". EDA may act as either an architecture style *or* a communication pattern. When an architect builds an entire system consisting only of event processors — without the strict data and coupling requirements imposed by microservices — they have built an architecture in the event-driven architecture style. However, if an architect designs an architecture that closely adheres to the microservices philosophy *and* uses events and/or messages as a way to improve responsiveness (a common optimization in these architectures), that's a communication pattern, not an architecture style.

Each system really only has one dominant architecture *style*, but systems can use many patterns. The overlap between EDA and microservices nicely illustrates the difference between styles and patterns.

Generally, events are asynchronous and messages are synchronous, but that's not universally true and is based on the requirements of the situation. Throughout the book, we illustrate synchronous communication with solid arrows and asynchronous communication with dotted arrows, as illustrated in Figure 37-18.

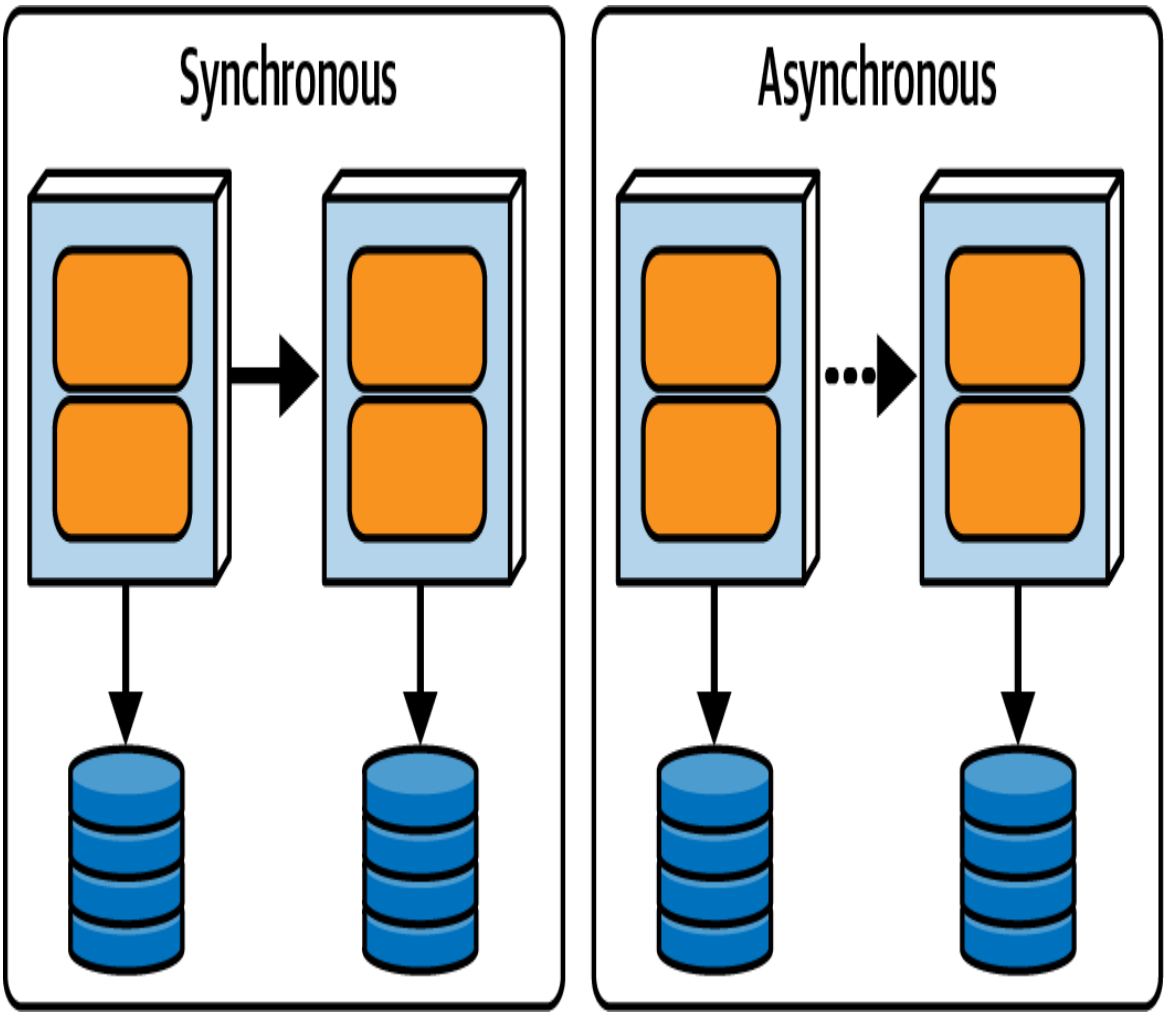


Figure 37-18. Solid lines for synchronous communication, dotted for asynchronous

Architects should prefer neither messages or events by default — each has its own strengths and trade-offs. Some of our patterns diverge when using one or the other (because the distinction changes the values of one or more trade-offs), while others are unaffected.

Automating Architectural Governance: Fitness Functions

One key job of a software architect is *governing* the decisions they make and aspects of the codebase as it grows. If an architect designs a brilliant system but ignores its implementation, their vision won't come true. Nor is the code likely to exhibit high quality, such as well-defined coupling, high cohesion, useful modularity, security, and a host of other features that must reflect both design and implementation. One way architects can build that governance is with *architectural fitness functions*.

Originally defined in the book *Building Evolutionary Architectures*, an *architectural fitness function* acts as verification mechanism. It's sort of like a unit test, but for architectural capabilities rather than the problem domain. In other words, fitness functions test *capabilities*, while traditional testing (like unit, functional, and user-acceptance testing) tests *behavior*. Thus, some fitness functions are implemented like unit tests and run as part of automated builds, but others must be implemented with monitors, forensic logging, and other verification mechanisms. Architects can't check all the things they might want to govern at build time — some require DevOps support.

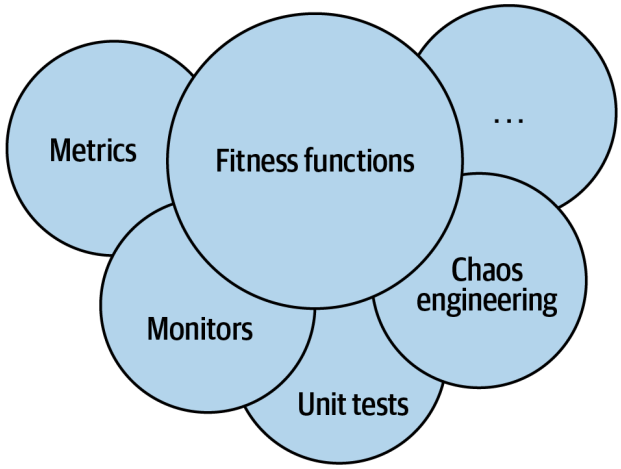


Figure 37-19. Fitness functions encompass a number of verification mechanisms

In Figure 37-19, architects implement some fitness functions using testing libraries (such as ArchUnit), but runtime operational measures require

monitors. When an architect wants to assess the internal quality of code, they can use a variety of metrics. In fact, many projects wire tools like SonarQube into their build; those teams are using fitness functions, whether they realize it or not. Innovations such as Netflix's implementation of chaos engineering also qualify as architectural fitness functions.

From *Building Evolutionary Architectures*, the formal definition of an architectural fitness function is "any mechanism that provides an objective integrity check on some architectural characteristic(s)."

Several important parts of this short definition stand out:

Any mechanism

> Fitness functions are not restricted to any particular testing framework or library. While some structural fitness functions utilize testing frameworks, other use monitors or forensic debugging to learn about (and govern) architectural capabilities.

Objective integrity check

> A fitness function must have an unambiguous outcome. For example, an architect can set up a monitor but only use it as a dashboard. The key differentiation between that and implementing a fitness function with a monitor is *objective measurement*. Similarly, merely having a monitor in place does not equal a fitness function — having a monitor that *raises an alarm* when a measurement exceeds a threshold equals a fitness function.
>
> *Objective* doesn't imply *fixed*. While some fitness functions return discrete values (*true*, *42*), others may make real-time calculations.

Architectural capabilities

> Architectural capabilities describe the capabilities of the system. In Chapter 2, we list the various capabilities that were either improved or degraded for each of our patterns.

Part of choosing a pattern is providing some way to govern that pattern. Patterns always have negative side effects and things to watch out for — that's why they're *patterns*, not just *solutions*. Using architectural fitness functions allows architects to both define and govern patterns. To that end, we show how to govern a number of our patterns.

---

[1] Haerder, Theo, and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery." *ACM Computing Surveys* (1983).

Index

A

ACID (Atomic, Consistent, Isolated, Durable) transactions

atomicity, Atomicity-Atomicity

consistency, Consistency-Consistency

durability, Durability-Durability

faking parts of, Faking Parts of ACID

isolation, Isolation-Isolation

versus BASE, ACID Versus BASE-ACID Versus BASE

adaptability, Adaptability, Evolvability

ADRs (architecture decision records), How to Govern This Pattern, Identify union of capabilities

Advanced Message Queueing Protocol (AMQP), How Do I Prioritize My Messages?

agentic AI, Using agentic AI in the Workflow Processor

agentic mesh, Agentic Mesh, LLMs, and MCP-Agentic Mesh, LLMs, and MCP, Applying the Pattern (see also data mesh; service mesh)

aggregates, Applying the Pattern-Identifying Events and Aggregates, Schema Evolution, Consistency Model and Atomicity

agility, Agility

AI agents, Using agentic AI in the Workflow Processor

Allocation Map, Applying the Pattern-Assign a Message to a Thread

AMQP (Advanced Message Queueing Protocol), How Do I Prioritize My Messages?

analytical data, Data Mesh and Cooperative Quantum-Data Mesh and Cooperative Quantum (see also operational data)

anemic events, Anemic Events-Anemic Events

antipatterns, Antipatterns, The Value of Patterns

AOP (aspect-oriented programming), Aspect-Oriented Architecture?

APIs as products, How Can I Support Multiple User-Interface Types?

architects, Should I Only Base My Decisions on Past Experiences?, Events Versus Messages

architectural capabilities

- in Architecture by Implication antipattern, Why Is This an Antipattern?-Why Is This an Antipattern?
- architecture quantum as scope for, Defined
- composite, Agility
- governing with fitness functions, Automating Architectural Governance: Fitness Functions
- impact of patterns on, Architecture Patterns Versus Design Patterns
- patterns listed by, Capability and Context Cross-Reference
- rated with qualitative analysis, Qualitative Analysis-Qualitative Analysis

architectural components, Greenfield Systems

architectural governance

of closed layers, Am I in This Antipattern?

completing qualitative analysis, From Qualitative to Quantitative Analysis

with fitness functions, Automating Architectural Governance: Fitness Functions-Automating Architectural Governance: Fitness Functions

of patterns, How Is This Book Different from Other Patterns Books?

architecture

Aspect-Oriented, Aspect-Oriented Architecture?-Aspect-Oriented Architecture?

assessing need for, When Do I Need an Architecture?

Big Ball of Mud, When Do I Need an Architecture?

distributed, brokers in, How Many Brokers Should I Have in My Distributed Architecture?

First Law of Software, Trade-Offs, Why Is This an Antipattern?, Build Your Own Pattern Analysis

iterative, Why Is This an Antipattern?, Iterative Architecture and Residuality Theory-Iterative Architecture and Residuality Theory

layered, Architecture Sinkhole Antipattern, Opening Layers-Consider a Different Architecture Style, How Can I Support Multiple User-Interface Types?, Applying the Pattern

layered monolithic, Domain Alignment, Layered Architecture

logical, Why Is This an Antipattern?, Greenfield Systems

microkernel, Adaptability

monolithic stovepipe, Repairing a monolithic stovepipe architecture-Repairing a monolithic stovepipe architecture

physical, Styles Versus Patterns, Greenfield Systems, Brownfield Systems

Reactive, Applying the Pattern

service-based, How to Avoid This Pitfall-How to Avoid This Pitfall

service-oriented, Degraded

Third Law of Software, The Pattern, Antipattern, Pitfall Spectrum

architecture decision records (ADRs), How to Govern This Pattern, Identify union of capabilities

architecture fitness functions

in Async Chain of Persistence pattern, How to Govern This Pattern

in Backend for Frontend pattern, How to Govern This Pattern

in Cache Aside pattern, How to Govern This Pattern

in Compensating Updates pattern, Failure to Achieve the Compensating Update-What Level of Consistency Is Proper?

container-level, How to Govern This Pattern

in Contextual Events pattern, How to Govern This Pattern

in CQRS pattern, How to Govern This Pattern

in Data-Based Event Contract pattern, How to Govern This Pattern

defined, Building Quantitative Analysis, Automating Architectural Governance: Fitness Functions-Automating Architectural Governance: Fitness Functions

in Domain-Broker pattern, How to Govern This Pattern

in Event Sourcing pattern, How to Govern This Pattern

in Extensible Events pattern, How to Govern This Pattern

in Front Controller pattern, How to Govern This Pattern

in Key-Based Event Contract pattern, How to Govern This Pattern

in Nexus Architecture pattern, Improved

in Orchestration pattern, Intraservice Communication

reactive, Failure to Achieve the Compensating Update

verifying qualitative analysis, From Qualitative to Quantitative Analysis

architecture patterns, Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls, Architecture Patterns Versus Design Patterns, The Value of Patterns, Automating Architectural Governance: Fitness Functions

architecture quantum

atomic coupling, Atomic coupling

in Choreography pattern, Architecture Quantum-Architecture Quantum

contract coupling, Contract coupling-Contract coupling

defined, Defined-Defined

in Domain-Broker pattern, Trade-Offs, Architecture Quantum-Architecture Quantum

dynamic coupling, Dynamic coupling-Dynamic coupling

in Event Sourcing pattern, Architecture Quantum-Architecture Quantum

in Explicit Workflow State pattern, Influences

in Implicit Workflow State pattern, Influences

infrastructure ownership, Events Versus Messages

as inspiration for data mesh, Applying the Pattern

in Nexus Architecture pattern, Influences

in Orchestration pattern, Architecture Quantum-Architecture Quantum

overview of, Architecture Quantum

in Service Sharding pattern, Architecture Quantum

as single point of failure, How Many Brokers Should I Have in My Distributed Architecture?, Improved

static coupling, Static coupling-Static coupling

temporal coupling, Temporal coupling-Temporal coupling

architecture sleight of hand, Performance

Aspect-Oriented Architecture, Aspect-Oriented Architecture?-Aspect-Oriented Architecture?

aspect-oriented programming (AOP), Aspect-Oriented Architecture?

AspectJ, Aspect-Oriented Architecture?

asynchronous communication

in Compensating Updates pattern, Influences

data pumps, Applying the Pattern

error handling, How Do I Handle Asynchronous Errors?

joining architecture quanta, Data Mesh and Cooperative Quantum

in orchestrated workflows, The Case Where Everything Works

send mode, Event chaining

versus synchronous

in Choreography pattern, Communication (Synchronous Versus Asynchronous)-Degraded

in Orchestration pattern, Communication (Synchronous Versus Asynchronous)-Degraded

atomic consistency, Building Qualitative Analysis, Writes/Updates

atomic context, Applying the Pattern, Trade-Offs-Improved, How to Govern This Pattern

atomic coupling

in Choreography pattern, Architecture Quantum

defined, Atomic coupling

in Domain-Broker pattern, Architecture Quantum

in Event Sourcing pattern, Architecture Quantum

in Implicit Workflow State pattern, Influences

in Nexus Architecture pattern, Influences

in Orchestration pattern, Architecture Quantum

in Service Sharding pattern, How Should I Implement Atomicity in My Distributed Architectures?, Architecture Quantum

Atomic, Consistent, Isolated, Durable transactions (see ACID transactions)

atomicity, Atomicity, Service Granularity, Atomicity-Atomicity

attractors, Iterative Architecture and Residuality Theory

audit logs, Should I Capture My Data as an Event Stream?

auto-acknowledgement mode, Receiving a message

availability, Availability, ACID Versus BASE

B

bandwidth usage, Bandwidth Usage

BASE (Basic Availability, Soft State, Eventual Consistency), ACID Versus BASE-ACID Versus BASE

BDD (behavior-driven development), Improved

Beck, Kent, Repairing a distributed Stovepipe Architecture

Big Ball of Mud architecture, When Do I Need an Architecture?

bottlenecks, How Can I Improve Throughput and Scalability?, How to Govern This Pattern

boundary conditions, Pros

bounded contexts

- in Grains of Sand pitfall, Why Is This a Pitfall?-Why Is This a Pitfall?, Why Is This a Pitfall?
- in Key-Based Event Contract pattern, Data Access Considerations
- in microservices style, Events Versus Messages
- in Service Sharding pattern, Cons
- in Stovepipe Architecture antipattern, When changes involve new data requirements, When adding behavior to existing services

brittle architectures, Static coupling

broadcasting versus commanding, Events Versus Messages

broadly scoped orchestration, When Should I Use Global Coupling? (see also global orchestrator)

broker affinity, Related Patterns, Antipatterns, and Pitfalls

broker coupling, Cons

brownfield systems

adding behavior to existing services, When adding behavior to existing services-When adding behavior to existing services

changes involving new data requirements, When changes involve new data requirements-When changes involve new data requirements

changes requiring service orchestration, When changes require service orchestration-When changes require service orchestration

overview of, Brownfield Systems

repairing stovepipe architecture, Repairing a monolithic stovepipe architecture-Repairing a distributed Stovepipe Architecture

versus greenfield systems, How to Avoid This Antipattern

Build your own Radar, Developing a Personal Radar

Building Evolutionary Architectures (Ford et al.), Building Quantitative Analysis, Evolvability, Architecture Quantum, Automating Architectural Governance: Fitness Functions

business impact, Business impact

business intelligence tools, Degraded

business invariants, Identifying Events and Aggregates, Validation

business logic, Applying the Pattern

C

caching

in critical path, Cache in the critical path-Cache in the critical path

entry invalidation, Writes/Updates

failures, Handling Cache Failures

hits, Reads, Reads

misses, Reads, Reads

optimizing data consistency in, When Using Caching, How Do I Optimize Data Consistency?

stampedes, Cons

caching topologies

capabilities compared across, Caching Topology Comparison-Degraded

client-server, Caching Topologies, Influences

replicated in-memory, Caching Topologies, Influences

single in-memory, Influences, Influences

callbacks, Process Messages-Process Messages

capabilities (see also architectural capabilities; operational capabilities; process-related capabilities)

choosing, Choosing Capabilities

identifying union of, Identify union of capabilities

primary, Patterns by Capability

capabilities versus behavior

in Cross-Cutting State pattern, How to Govern This Pattern

defined, Architecture Patterns Versus Design Patterns

fitness functions versus testing, Automating Architectural Governance: Fitness Functions

in pattern comparisons, Choosing Capabilities

in Stovepipe Architecture antipattern, Why Is This an Antipattern?-Why Is This an Antipattern?

Carducci, Michael, How Can I Guarantee Message Processing Order?

Carpool variant, Applying the Pattern, Degraded

CDC (change data capture), A note on eventual consistency

change management, How to Govern This Pattern

change rate (see volatility)

chaos engineering, How to Govern This Pattern, How to Govern This Pattern, How to Govern This Pattern, Cache in the critical path

choreography, Workflow and choreography

churn (see volatility)

client acknowledgement mode, Client Acknowledgement Mode, Cons

client ID, Client Acknowledgement Mode

client-server cache, Caching Topologies, Influences

closed layers, Why Is This an Antipattern?

coarse-grained services, How to Avoid This Pitfall

code

duplication, Degraded, How Do I Manage Cross-Cutting Concerns?

shared, Shared code, Repairing a monolithic stovepipe architecture

volatility, Code volatility, Shared code

cohesion

in Cross-Cutting State pattern, Cons

defined, Defined

in Grains of Sand pitfall, Service scope and function-Service scope and function

in Implicit Workflow State pattern, Cons

cold cache, Caching Topologies, Influences

command and query services, Single Service for Both Command and Query-The read-your-own-writes challenge

command handler, Applying the Pattern, Command handler

command side of data model, Applying the Pattern

commands, Applying the Pattern

common definitions, Patterns by Capability, Fundamental Definitions

communication (see also asynchronous communication; synchronous communication)

channels, Events Versus Messages

in Choreography pattern, What Happens When Everything Works-What Happens When Everything Works, Communication (Synchronous Versus Asynchronous)-Degraded

intraservice, Intraservice Communication

in Orchestration pattern

example workflow, The Case Where Everything Works

trade-offs, Communication (Synchronous Versus Asynchronous)-Degraded

overhead, Granularity

patterns, Events Versus Messages

point-to-point versus publish-and-subscribe, Events Versus Messages

style, Styles Versus Patterns

compensating updates, Atomicity (Data Consistency and Integrity), Implementing the Pattern, Failure to Achieve the Compensating Update-Failure to Achieve the Compensating Update, Why Is This a Pitfall?

Competing Consumers pattern, How Can I Guarantee Message Processing Order?

complex workflows, Pros

complexity, Simplicity, Reflect what you want to measure

components

identifying, Greenfield Systems

refactoring patterns for, Repairing a monolithic stovepipe architecture

topology and dependencies, Styles Versus Patterns

concurrency, Handling Concurrency-Handling Concurrency

conflicting patterns, Conflicting Patterns, Applying the Pattern

connection waits, How to Govern This Pattern

consistency (see also eventual consistency)

ACID versus BASE, Consistency-Consistency

atomic, Building Qualitative Analysis, Writes/Updates

data, Data Consistency, When Using Caching, How Do I Optimize Data Consistency?, Faking Parts of ACID

in Event Sourcing pattern, Consistency Model and Atomicity

in Orchestration pattern, Influences

strong, Caching Topologies, Consistency Model and Atomicity, Write-Behind Cache Pattern Variant, Influences

consistent response time, How Can I Improve Throughput and Scalability?, Consistent Response Time

consolidated state machine, Applying the Pattern (see also distributed state machine)

consolidating services, When changes require service orchestration-When changes involve new data requirements

consumer groups, Applying the Pattern

consumer-driven contracts, Consumer-Driven Contracts-Consumer-Driven Contracts

container-level fitness function, How to Govern This Pattern (see also architecture fitness functions; reactive fitness function)

context

for antipatterns, Antipatterns

atomic, Applying the Pattern, Trade-Offs-Improved, How to Govern This Pattern

message, How Can I Guarantee Message Processing Order?

in pattern metadata, How Is This Book Different from Other Patterns Books?

contextual derived events, Applying the Pattern

contract coupling

in Choreography pattern, Architecture Quantum

defined, Contract coupling

in Domain-Broker pattern, Architecture Quantum

in Event Sourcing pattern, Architecture Quantum

in Explicit Workflow State pattern, Influences

in Implicit Workflow State pattern, Influences

in Nexus Architecture pattern, Influences

in Orchestration pattern, Architecture Quantum

in Service Sharding pattern, How Should I Implement Atomicity in My Distributed Architectures?, Architecture Quantum

contracts

consumer-driven, Consumer-Driven Contracts-Consumer-Driven Contracts

defined, Data-Based Event Contract Pattern, Architecture Quantum

event, How Much Data Should I Include in the Event Payload?, Cons, How to Govern This Pattern, Pros

loose, Applying the Pattern, Applying the Pattern, How Specific Should My Contract Be?, Contract Management, Loose Contracts, Contract coupling

management of, Cons, Contract Management-Contract Management, Consumer-Driven Contracts, Cons

quantum coupling, Contract coupling-Contract coupling

rate of change, How to Govern This Pattern, Rate of Contract Change

specificity of, How Specific Should My Contract Be?-How Specific Should My Contract Be?

strict, Applying the Pattern, How Specific Should My Contract Be?, Contract Management, Contract coupling

versioning of, Contract Management

cooperative quantum, Data Mesh and Cooperative Quantum-Data Mesh and Cooperative Quantum, Applying the Pattern

coordination

in Architecture Sinkhole antipattern, Consider a Different Architecture Style

in Backend for Frontend pattern, Degraded

in Choreography pattern, Influences

in Contextual Events pattern, Influences

in Orchestration pattern, Influences

in Stamp Coupling antipattern, Pros

cost, Cost, When not to use stars

coupling (see also atomic coupling; contract coupling; dynamic coupling; static coupling; temporal coupling)

broker, Cons

coupled versus decoupled ratings, Reflect what you want to measure

from data dependencies, When changes involve new data requirements

global, When Should I Use Global Coupling?-When Should I Use Global Coupling?

increased by external state, Cons

increased by Front Controller, Cons

increased by orchestration, Cons

increased by small services, Why Is This a Pitfall?

increased by strict contracts, Cons

in inferred workflow state, Cons

loose, Static coupling

misaligned, Cons

operational, How Do I Manage Cross-Cutting Concerns?-Service Mesh and Sidecar

reduced by single derived event, Influences

semantic and implementation, Why Is This a Pitfall?

tight, Static coupling

types of, How Should I Implement Atomicity in My Distributed Architectures?

coupling distance, Influences-Influences, Influences

coupling points (see also join points)

architecture quanta defined by, Static coupling

centralized, Degraded-Am I in This Antipattern?

contracts as, Contract coupling

in CQRS pattern, Team Topologies

in Nexus Architecture pattern, Nexus Architecture Pattern

service mesh as, How Do I Manage Cross-Cutting Concerns?

create, read, update, delete (CRUD) operations, Command Query Responsibility Segregation (CQRS) Pattern-Applying the Pattern, Degraded

critical path, Applying the Pattern, Cache in the critical path-Cache in the critical path

cross-cutting concerns, How Do I Manage Cross-Cutting Concerns?-Agentic Mesh, LLMs, and MCP

agentic mesh, LLMs, and MCP, Agentic Mesh, LLMs, and MCP-Agentic Mesh, LLMs, and MCP

in aspect-oriented programming, Aspect-Oriented Architecture?

data mesh and cooperative quantum, Data Mesh and Cooperative Quantum-Data Mesh and Cooperative Quantum

in Nexus Architecture pattern, Nexus Architecture, Aspect-Oriented Architecture?

service mesh and sidecar, Service Mesh and Sidecar-Service Mesh and Sidecar

cross-service transactions, Service Sharding Pattern-How Should I Implement Atomicity in My Distributed Architectures? (see also distributed transactions)

CRUD (create, read, update, delete) operations, Command Query Responsibility Segregation (CQRS) Pattern-Applying the Pattern, Degraded

Cunningham, Ward, Repairing a distributed Stovepipe Architecture

D

dashboard interface, Manual Intervention (Dashboard)

data

analytical, Data Mesh and Cooperative Quantum-Data Mesh and Cooperative Quantum

in event payloads, How Much Data Should I Include in the Event Payload?-How Much Data Should I Include in the Event Payload?, How Much Data Should I Include in the Event Payload?

new requirements, When changes involve new data requirements-When changes involve new data requirements

operational, Data Mesh and Cooperative Quantum

partial, Applying the Pattern

reads and writes optimization, How Can I Optimize My Data for Reads and Writes?

relationships as granularity integrators, Data relationships

specialized query languages, The Place for Specialized Data Query Languages

stale, Handling Cache Failures

data access, How to Govern This Pattern

data consistency, Data Consistency, When Using Caching, How Do I Optimize Data Consistency?, Faking Parts of ACID

data fidelity, Trade-Offs

data granularity, Why Is This a Pitfall?

data integrity, Atomicity, Data Integrity

data lakes, Data Mesh and Cooperative Quantum

data loss, Pros

data mesh, Data Mesh and Cooperative Quantum-Data Mesh and Cooperative Quantum, Applying the Pattern (see also agentic mesh; service mesh)

data model, Command Query Responsibility Segregation (CQRS) Pattern-Applying the Pattern

data topology, Styles Versus Patterns

data warehouses, Data Mesh and Cooperative Quantum

database topology

database-per-service, Data Access Considerations, Database-Per-Service Database Topology

domain-based, Domain-Based Database Topology-Domain-Based Database Topology

and event contract choice, Influences

in Key-Based Event Contract pattern, Domain-Based Database Topology-Domain-Based Database Topology

monolithic, Monolithic Database Topology

databases (see also ACID transactions)

connection pools, How to Govern This Pattern

responsiveness, How to Govern This Pattern

transactions, Database transactions

triggers, Should I Capture My Data as an Event Stream?

datastore indexes, How Can I Optimize My Data for Reads and Writes?

DDD (Domain-Driven Design), Implementing the Pattern, Identifying Events and Aggregates, When Should I Use Global Coupling?, Events Versus Messages

dead letter queue (DLQ), Client Acknowledgement Mode, Dead letter queue

debuggability, Debuggability

decisions based on past experiences, Should I Only Base My Decisions on Past Experiences?-Should I Only Base My Decisions on Past Experiences?

decomposition strategies, Implementing the Pattern

decoupled architectures, Temporal coupling

dedicated database topology (see database topology)

Dehghani, Zhamak, Qualitative Analysis, Data Mesh and Cooperative Quantum

delivery mode

- asynchronous send, Event chaining
- auto-acknowledgement, Receiving a message
- client acknowledgement, Client Acknowledgement Mode-Client Acknowledgement Mode, Cons
- persistent, Message Persistence

dependency graph, Why Is This an Antipattern?

deployability, Deployability

deployment model, Styles Versus Patterns

derived events, Event chaining

design complexity, Degraded

design patterns, Architecture Patterns Versus Design Patterns

Design Patterns (Gamma et al.), Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls

Deutsch, L. Peter, Governing Communication

development costs, Degraded

distance coupling (see coupling distance)

distributed architectures, Styles Versus Patterns

distributed governance strategy, How to Govern This Pattern-How to Govern This Pattern

distributed state machine, Applying the Pattern (see also consolidated state machine)

distributed stovepipe architecture, repairing, Repairing a distributed Stovepipe Architecture

distributed transactions, How Do I Roll Back a Distributed Transaction?-How Do I Roll Back a Distributed Transaction?, Why Is This a Pitfall?, Database transactions, When changes require service orchestration (see also cross-service transactions)

DLQ (dead letter queue), Client Acknowledgement Mode, Dead letter queue

domain abstraction, Domain Alignment, Degraded, Why Is This an Antipattern? (see also technical abstraction)

domain alignment, Domain Alignment

domain drift, How to Govern This Pattern

domain events, Identifying Events and Aggregates-Domain Events, How to Govern This Pattern

domain failure, Implementing the Pattern

domain partitioning, Pros, Applying the Pattern

domain services, What Happens When Everything Works, How to Avoid This Pitfall-How to Avoid This Pitfall, Pros

domain-based database topology, Domain-Based Database Topology-Domain-Based Database Topology

domain-centric teams, Data Mesh and Cooperative Quantum

Domain-Driven Design (DDD), Implementing the Pattern, Identifying Events and Aggregates, When Should I Use Global Coupling?, Events Versus Messages

dual write problem, Cons

duplicate message processing, Cons

"duplication is better than coupling" mantra, Reusability

durability, Durability-Durability

dynamic coupling

- in Choreography pattern, Architecture Quantum
- defined, Dynamic coupling-Dynamic coupling
- in Domain-Broker pattern, Architecture Quantum
- in Event Sourcing pattern, Architecture Quantum
- in Explicit Workflow State pattern, Influences
- in Implicit Workflow State pattern, Influences
- in Nexus Architecture pattern, Influences
- in Orchestration pattern, Architecture Quantum
- in Service Sharding pattern, How Should I Implement Atomicity in My Distributed Architectures?, Architecture Quantum

E

EDAs (event-driven architectures)

- last-participant support, Last Participant Support
- as single architecture quantum, How Many Brokers Should I Have in My Distributed Architecture?
- and stamp coupling, Influences
- tracking workflow state, Implicit Workflow State Pattern-How Should I Manage My Workflow State?

use of events and messages, Events Versus Messages

versus message-based systems, How Do I Prevent Message Loss?-Event-Driven Versus Message-Based Systems

versus microservices, Events Versus Messages

workflows in, Pros

elasticity, Elasticity

emergent design, Why Is This an Antipattern?-Why Is This an Antipattern?

Encino Man (film), Should I Only Base My Decisions on Past Experiences?

enterprise architects, How to Govern This Pattern-How to Govern This Pattern

Enterprise Integration Patterns (Woolf and Hohpe), Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls

Enterprise Service Bus (ESB), How Should I Manage My Workflows?, When Should I Use Global Coupling?

epic saga, Building Qualitative Analysis

error console, Compensating Update Failures

error handling

defined, Error Handling

failure of compensating updates, Degraded, Failure to Achieve the Compensating Update-Failure to Achieve the Compensating Update

lacking designated component, Degraded

logic residing in domain services, What Happens When an Error Occurs?, What Happens When an Error Occurs?

reuse cost of smearing, Cons

timing and responsibility at scale, Improved

error queues, Workflow Event Processor-Workflow Event Processor

error rates, How to Govern This Pattern

ESB (Enterprise Service Bus), How Should I Manage My Workflows?, When Should I Use Global Coupling?

Evans, Eric, Events Versus Messages

Event Consumer, Applying the Pattern, Pros

event contracts, How Much Data Should I Include in the Event Payload?, Cons, How to Govern This Pattern, Pros

event dispatcher

callback processing by, Update the Allocation Map

dispatching algorithm of, Assign a Message to a Thread-Assign a Message to a Thread

handing messages to threads, Process Messages

responsibilities of, Applying the Pattern

event processors, Implementing the Pattern, Implementing the Pattern

event store, Applying the Pattern, Event Store Structure-Event Store Structure

event-driven architectures (see EDAs)

events

anemic, Anemic Events-Anemic Events

chaining, Event chaining-Event chaining, Contextual Events Pattern, Extensible Events Pattern

contextual derived, Applying the Pattern

data in payload, How Much Data Should I Include in the Event Payload?-How Much Data Should I Include in the Event Payload?, How Much Data Should I Include in the Event Payload?

derived, Event chaining

domain, Identifying Events and Aggregates-Domain Events, How to Govern This Pattern

generic derived, How Many Events Should I Trigger from a Single Service?-How Many Events Should I Trigger from a Single Service?

granularity, Event Granularity

identifying, Identifying Events and Aggregates-Identifying Events and Aggregates

integration, Architecture Quantum

without listeners, Should I Trigger an Event If No One Is Listening for It (Yet)?-Should I Trigger an Event If No One Is Listening for It (Yet)?

log, Event Sourcing Pattern

noise, Cons

objects, Applying the Pattern

out-of-order, Cons

publishing, Persistence, Degraded

reading, Reading Events-Reading Events

replay, Should I Capture My Data as an Event Stream?, Event Volume and Scale

schemas, Architecture Quantum

significant, Should I Trigger an Event If No One Is Listening for It (Yet)?

from single service, How Many Events Should I Trigger from a Single Service?-How Many Events Should I Trigger from a Single Service?

streams, Event Store Structure

uncommitted, Event generation

versus messages, Events Versus Messages-Events Versus Messages

volume and scale, Event Volume and Scale

writing

command handler, Command handler

event generation, Event generation

overview of, Writing Events-Writing Events

persistence, Persistence

validation, Validation

eventual consistency

as BASE property, ACID Versus BASE

in Cache Aside pattern, Caching Topologies

in Compensating Updates pattern, What Level of Consistency Is Proper?

in CQRS pattern, How to Govern This Pattern

in Event Sourcing pattern, Cons, Consistency Model and Atomicity, How to Govern This Pattern

in Read-Through/Write-Through Cache pattern, Influences

evolvability, Evolvability

expectedVersion number, Writing Events

external state, Applying the Pattern-Applying the Pattern

externally facing APIs, How Can I Support Multiple User-Interface Types?

F

fail fast, How to Govern This Pattern

fairy tale saga, Building Qualitative Analysis

Fallacies of Distributed Computing (Richards and Ford), Why Is This a Pitfall?

fault tolerance, Fault Tolerance

fidelity of captured history, Should I Capture My Data as an Event Stream?

FIFO (first in, first out) queues, Message Processing Order, How Do I Prioritize My Messages?, Degraded

First Law of Software Architecture, Trade-Offs, Why Is This an Antipattern?, Build Your Own Pattern Analysis

five-star scale, Qualitative Analysis-Building Qualitative Analysis

flexibility, Flexibility

Ford, Neal, Building Quantitative Analysis, Evolvability, Why Is This a Pitfall?, Stars and Other Icons

forensic debugging, Debuggability

Fowler, Martin, Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls, Antipatterns

Fraser, Brendan, Should I Only Base My Decisions on Past Experiences?

Front Orchestrator, Front Controller Pattern

function-level scalability, Pros

Fundamentals of Software Architecture (Richards et al.)

architectural styles covered in, Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls

architectural styles partitioned in, Applying the Pattern

capabilities list in, Capability and Context Cross-Reference

comparison charts in, Qualitative Analysis

composite capabilities defined in, Agility

cost ratings in, When not to use stars

First Law coined in, Trade-Offs

styles versus patterns distinction in, Architectural Styles

G

Gall, John, Degraded, Why Is This an Antipattern?

Gall's Law, Why Is This an Antipattern?

Gamma, Erich, Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls

Gang of Four book, Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls

GDPR (General Data Protection Regulation), Cons (see also right to be forgotten)

general-purpose patterns, Choosing Capabilities

generative AI, How Is This Book Different from Other Patterns Books?, Agentic Code Generation, How to Govern This Pattern, Agentic Mesh, LLMs, and MCP

generic derived events, How Many Events Should I Trigger from a Single Service?-How Many Events Should I Trigger from a Single Service?

global coupling, When Should I Use Global Coupling?-When Should I Use Global Coupling?

global orchestrator, How Should I Manage My Workflows?

global sequencing mechanism, Event Store Structure

godlike event processor, Cumulative Responsibility

governance

- architectural, How Is This Book Different from Other Patterns Books?, Am I in This Antipattern?, From Qualitative to Quantitative Analysis, Automating Architectural Governance: Fitness Functions-Automating Architectural Governance: Fitness Functions
- automating, Automating Architectural Governance: Fitness Functions-Automating Architectural Governance: Fitness Functions

governance mesh (see service mesh)

granularity

- in Backend for Frontend pattern, Influences
- data, Why Is This a Pitfall?
- event, Event Granularity
- in Nexus Architecture pattern, Influences
- in Orchestration pattern, How Should I Manage My Workflows?, Granularity
- service, Why Is This a Pitfall?-Why Is This a Pitfall?, Shared code, Service Granularity, How to Govern This Pattern

granularity disintegrators

code volatility, Code volatility

evolvability, Evolvability

fault tolerance, Fault tolerance

overview of, Granularity Disintegrators

scalability and throughput, Scalability and throughput

security, Security

service scope and function, Service scope and function-Service scope and function

granularity integrators, How to Avoid This Pitfall, Granularity integrators-Workflow and choreography

GraphQL, The Place for Specialized Data Query Languages, GraphQL-GraphQL

greenfield systems, Greenfield Systems-Greenfield Systems

H

Haerder, T., ACID Versus BASE

hashtags, Using a Zettelkasten (Slip Box)

Helm, Richard, Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls

history tables, Should I Capture My Data as an Event Stream?-Should I Capture My Data as an Event Stream?

Hohpe, Gregor, Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls

hooks, in system, Improved

Horror Story saga, Applying the Pattern (see also Phone Tag saga)

hybrid workflows, Hybrids

I

immutability, Schema Evolution, Schema Evolution

impedance mismatch, How Can I Optimize My Data for Reads and Writes?

implementation coupling, Why Is This a Pitfall?

in-memory replicated caching (see replicated in-memory cache)

independent deployment, Defined

influences, How Is This Book Different from Other Patterns Books?, Identifying Influences

infrastructure costs, Degraded

initial time to market, Initial Time to Market

initiating event, How Much Data Should I Include in the Event Payload?

initiating event owner, Pros

insignificant events (see significant events)

integration events, Architecture Quantum

integration test, The Cache Is Not in the Critical Path, Cache in the critical path

interconnectivity, Cons

interservice calls, Repairing a distributed Stovepipe Architecture

intraservice communication, Intraservice Communication

isolation, Isolation-Isolation, Faking Parts of ACID

J

JMS (Java Message Service), How Do I Prioritize My Messages?

Johnson, Ralph, Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls

join points

- architecture quanta and, Influences
- in data mesh, Data Mesh and Cooperative Quantum
- explicit and orthogonal, Aspect-Oriented Architecture?
- governance of, How to Govern This Pattern
- misaligned coupling at, Cons
- operational independence and, Nexus Architecture
- overlapping architectures via, Applying the Pattern
- sidecar as, Service Mesh and Sidecar

K

Kafka, How to Govern This Pattern (see also streaming brokers)

key-based payloads, Key-Based Payloads

knowledge pyramid, How to Avoid This Pitfall-How to Avoid This Pitfall

Koenig, Andrew, The Value of Patterns

Kua, Patrick, Building Quantitative Analysis

L

last participant support, Last Participant Support

layered monolithic architecture, Domain Alignment, Layered Architecture

layers

closed, Why Is This an Antipattern?

governance of, Am I in This Antipattern?

opening, Opening Layers-Consider a Different Architecture Style

lazy loading, Cache Aside Pattern, Reads

legacy technologies, Cons

LLMs (large language models), How Is This Book Different from Other Patterns Books?, Agentic Mesh, LLMs, and MCP-Agentic Mesh, LLMs, and MCP, Applying the Pattern

loader, Applying the Pattern, Loader and store implementation

logging, How to Govern This Pattern, How Do I Manage Cross-Cutting Concerns?

logical architecture, Why Is This an Antipattern?, Greenfield Systems

loose contracts

in contract coupling, Contract coupling

in Data-Based Event Contract pattern, Applying the Pattern

in Key-Based Event Contract pattern, Applying the Pattern

in Stamp Coupling antipattern, How Specific Should My Contract Be?, Contract Management, Loose Contracts

loose coupling, Static coupling

M

macro variant, Macro Variant-Macro Variant, How to Govern This Pattern

maintainability, Maintainability

Martin, Robert C., Repairing a distributed Stovepipe Architecture

master data management, When Should I Use Global Coupling? (see also single source of truth)

materialized views (see projections)

maximum threshold, Applying the Pattern

MCP, Agentic Mesh, LLMs, and MCP-Agentic Mesh, LLMs, and MCP

mean time to recovery (MTTR), Degraded, Degraded

mean time to start (MTTS), Elasticity, Degraded, Fault tolerance, Should I Make Everything a Microservice?

mediator pattern, Implementing the Pattern

message brokers, The message broker crashes, Cons, Degraded, How Can I Improve Message Throughput and Capacity?, Events Versus Messages (see also broker coupling; streaming brokers)

message processing order, Message Processing Order (see also FIFO queues)

message throughput, Throughput, Scalability and throughput, How Can I Improve Throughput and Scalability?-How Can I Improve Throughput and Scalability?

message volume (see request volume)

message-based systems, Event-Driven Versus Message-Based Systems-Event-Driven Versus Message-Based Systems, Last Participant Support

messages

- acknowledgement, Last Participant Support
- context, How Can I Guarantee Message Processing Order?
- guaranteeing processing order, How Can I Guarantee Message Processing Order?-How Can I Guarantee Message Processing Order?

high-priority versus normal-priority, How Do I Prioritize My Messages?

hub, Cumulative Responsibility

IDs, Cons

improving throughput and capacity, How Can I Improve Message Throughput and Capacity?

loss prevention

- event chaining, Event chaining-Event chaining
- event-driven versus message-based systems, How Do I Prevent Message Loss?-Event-Driven Versus Message-Based Systems
- message broker crashes, The message broker crashes
- receiving messages, Receiving a message

nonpersisted, The message broker crashes

normal-priority, Applying the Pattern

persistence of, Message Persistence

priority header property, How Do I Prioritize My Messages?, Cons

producers, Message Persistence

redelivery, Client Acknowledgement Mode

storage in memory, Degraded

in Thread Delegate pattern

- assigning to thread, Assign a Message to a Thread-Assign a Message to a Thread
- overview of, Applying the Pattern-Applying the Pattern

processing, Process Messages-Process Messages

running out of threads, Running Out of Threads

updating allocation map, Update the Allocation Map

versus events, Events Versus Messages-Events Versus Messages

meta-workflow state, How Should I Manage My Workflow State?

metadata, How Is This Book Different from Other Patterns Books?, How to Use Chapter 2, "Capability and Context Cross-Reference", Improved/degraded

metapattern, Applying the Pattern

microkernel architecture, Adaptability

microservice architectures, Styles Versus Patterns

microservices, Architecture Quantum, Events Versus Messages (see also service-based architecture)

migration strategy, Implementing the Pattern

mini-monoliths, Cons

misaligned coupling, Cons

modular monolith, Domain Alignment, Consider a Different Architecture Style

modularity, Modularity

monitors, Automating Architectural Governance: Fitness Functions

monolithic database topology, Monolithic Database Topology

monolithic stovepipe architecture, Repairing a monolithic stovepipe architecture-Repairing a monolithic stovepipe architecture

monotonicity, Event Store Structure

MTTR (mean time to recovery), Degraded, Degraded

MTTS (mean time to start), Elasticity, Degraded, Fault tolerance, Should I Make Everything a Microservice?

multithreading, How Can I Improve Throughput and Scalability?

N

Newman, Sam, How to Avoid This Pitfall

nonfunctional requirements (see architectural capabilities)

nonpersisted messages, The message broker crashes

O

O'Reilly, Barry, Iterative Architecture and Residuality Theory

objective integrity check, Automating Architectural Governance: Fitness Functions

observability, How to Govern This Pattern, How to Govern This Pattern, How to Govern This Pattern, Applying the Pattern

Obsidian, Using a Zettelkasten (Slip Box)-Using a Zettelkasten (Slip Box)

operational capabilities

- in Architecture by Implication antipattern, When Do I Need an Architecture?, Am I in This Antipattern?
- in Async Chain of Persistence pattern, Cons
- bound by synchronous calls, Dynamic coupling
- in Choreography pattern, Pros
- in Frozen Caveman pitfall, Operational capabilities
- in Key-Based Event Contract pattern, Cons

in Multi-Broker pattern, Pros

in Nexus Architecture pattern, How Do I Manage Cross-Cutting Concerns?

in Orchestration pattern, Cons

in Service Sharding pattern, Pros

in Stovepipe Architecture antipattern, Am I in This Antipattern?

in Supervisor Consumer pattern, Pros

in Workflow Event pattern, Pros

operational complexity, Cons

operational costs, Degraded

operational coupling, How Do I Manage Cross-Cutting Concerns?-Service Mesh and Sidecar

operational data, Data Mesh and Cooperative Quantum (see also analytical data)

optimistic concurrency control, Writing Events, Handling Concurrency

orchestrator

compensating updates dispatched by, What Happens When Everything Works Correctly?-What Happens When Everything Works Correctly?

coordinating role of, How Should I Manage My Workflows?

error pathways handled by, The Case Where Something Breaks

Front, Front Controller Pattern

global, How Should I Manage My Workflows?

messages versus events, The Case Where Everything Works

as processing bottleneck, Cons

out-of-memory errors, How to Govern This Pattern

out-of-order events, Message Processing Order, Cons

overengineering, Iterative Architecture and Residuality Theory

ownership model, BFF Granularity

P

Parsons, Rebecca, Building Quantitative Analysis

partial data, Applying the Pattern

partitioning

domain, Pros

technical, Layered Architecture, Consider a Different Architecture Style, Improved

technical versus domain, Applying the Pattern

past experiences, Should I Only Base My Decisions on Past Experiences?-Should I Only Base My Decisions on Past Experiences?

pattern analysis

building capability list, Patterns by Capability

evaluating trade-offs

choosing capabilities, Choosing Capabilities

evaluating own patterns, Evaluating Your Own Patterns-Improved/degraded

identifying influences, Identifying Influences

overview of, Evaluating Trade-Offs for Patterns-Evaluating Trade-Offs for Patterns

stars and other icons, Stars and Other Icons-Reflect what you want to measure

using Zettelkasten (slip box), Using a Zettelkasten (Slip Box)-Using a Zettelkasten (Slip Box)

qualitative and quantitative, From Qualitative to Quantitative Analysis

value of patterns, The Value of Patterns

patterns

catalogs of, The Value of Patterns, Fundamental Definitions

levels of abstraction, How Is This Book Different from Other Patterns Books?

names of, Capability and Context Cross-Reference

spectrum of, The Pattern, Antipattern, Pitfall Spectrum

value of, The Value of Patterns

versus styles, Styles Versus Patterns-Styles Versus Patterns

Patterns in Enterprise Application Architecture (Fowler), Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls

payload contracts (see event contracts)

payload size, Pros, Payload Size and Message Volumes (see also request volume)

Payment Card Industry Data Security Standard (PCI DSS), Implementing the Pattern

permissive contracts (see loose contracts)

personal radar, Developing a Personal Radar-Developing a Personal Radar

Phone Tag saga, Applying the Pattern (see also Horror Story saga)

physical architecture, Styles Versus Patterns, Greenfield Systems, Brownfield Systems

pitfalls, Pitfalls, Trade-Offs

plumbing of system, How Do I Manage Cross-Cutting Concerns?

point-to-point communication, Events Versus Messages

predictability, Pros

primary capabilities, Patterns by Capability

priority queue, Applying the Pattern

priority thread, Applying the Pattern

problem resolution, How to Govern This Pattern

process-related capabilities, When Do I Need an Architecture?, Why Is This an Antipattern?, Am I in This Antipattern?, Process capabilities, Am I in This Antipattern?

processor threads

- FIFO queue of, Applying the Pattern
- holding off after errors, Process Messages
- multiple contexts per, Running Out of Threads
- queueing responsibility of, Assign a Message to a Thread

projections, Related Patterns, Antipatterns, and Pitfalls, Applying the Pattern, Reading Events, Cons, Event Volume and Scale

protocol agnosticism, The Place for Specialized Data Query Languages

protocols, Contract coupling

pseudocode, Applying the Pattern

pseudotransactional workflows, Pros

publish-and-subscribe communication, Events Versus Messages

Q

qualitative analysis (see also trade-off analysis)

- building, Building Qualitative Analysis-Building Qualitative Analysis
- overview of, Qualitative Analysis-Qualitative Analysis
- verified with fitness functions, From Qualitative to Quantitative Analysis
- versus quantitative analysis, Evaluating Trade-Offs for Patterns

quantitative analysis, Building Quantitative Analysis, Evaluating Trade-Offs for Patterns, From Qualitative to Quantitative Analysis

quantum of one, Defined

query patterns, Cons, Degraded

query requests, URL filtering for, URL Filtering for Query Requests

query side of data model, Applying the Pattern

queue depth, How to Govern This Pattern, Applying the Pattern-Applying the Pattern, How to Govern This Pattern

R

radar rings, Developing a Personal Radar

rate of contract change (see contracts)

Reactive Architecture, Applying the Pattern

reactive fitness function, Failure to Achieve the Compensating Update

Reactive Manifesto, The, Applying the Pattern

reactive verification mechanism, Governing Communication

read datastore, Applying the Pattern

read models (see projections)

read-your-own-writes challenge, The read-your-own-writes challenge

reads

- in caching, Reads-Reads
- optimization of data for, How Can I Optimize My Data for Reads and Writes?
- in Read-Through/Write-Through Cache pattern, Reads-Reads

referential integrity, Consistency

rehydration (see events)

reliability, Pros, Cons

remote cache, Caching Topologies, Influences

replicated in-memory cache

- in Cache Aside pattern, Caching Topologies
- in CQRS pattern, A note on eventual consistency
- in Key-Based Event Contract pattern, Cons
- in Read-Through/Write-Through Cache pattern, Influences

repositories, Applying the Pattern, Writing Events

request coalescing, Cons

request volume, Pros, Payload Size and Message Volumes (see also payload size)

Residuality Theory, Iterative Architecture and Residuality Theory-Iterative Architecture and Residuality Theory

residues, Iterative Architecture and Residuality Theory

resilience patterns, Improved

responsiveness, Responsiveness, How Do I Improve Both Performance and Scalability?, How to Govern This Pattern

retry strategy, Handling Concurrency-Handling Concurrency

reusability, Reusability

reuse philosophy, Styles Versus Patterns

Reuter, A., ACID Versus BASE

reverse index, How to Use Chapter 2, "Capability and Context Cross-Reference", Capability and Context Cross-Reference, Choosing Capabilities

Richards, Mark, Why Is This a Pitfall?, Iterative Architecture and Residuality Theory, Stars and Other Icons

right to be forgotten, Cons

Role and Responsibility Model, Repairing a distributed Stovepipe Architecture

root entity, Identifying Events and Aggregates

round-robin algorithm, Applying the Pattern

S

Sadalage, Pramod, Qualitative Analysis, Building Quantitative Analysis

sagas, Applying the Pattern

scalability

defined, Scalability

function-level, Pros

as granularity disintegrator, Scalability and throughput

improving, How Do I Improve Both Performance and Scalability?, How Can I Improve Throughput and Scalability?-How Can I Improve Throughput and Scalability?

qualitative star ratings, Building Qualitative Analysis

schema evolution, Schema Evolution-Schema Evolution

security, Pros, Pros, Process capabilities, Security

semantic coupling, Why Is This a Pitfall?

separation of concerns, Should I Capture My Data as an Event Stream?, Pros, Pros

sequence numbers, Event Store Structure, Writing Events

service atomicity, Service Granularity (see also granularity integrators)

service granularity, Why Is This a Pitfall?-Why Is This a Pitfall?, Shared code, Service Granularity, How to Govern This Pattern

service mesh, How to Govern This Pattern, Service Mesh and Sidecar-Service Mesh and Sidecar, Applying the Pattern (see also agentic mesh; data mesh)

service-based architecture, How to Avoid This Pitfall-How to Avoid This Pitfall (see also domain services)

service-level indicators (SLIs), How to Govern This Pattern

service-level objectives (SLOs), Implementing the Pattern, How to Govern This Pattern

service-oriented architecture (SOA), Degraded (see also Enterprise Service Bus)

services

- adding behavior to existing, When adding behavior to existing services-When adding behavior to existing services
- coarse-grained, How to Avoid This Pitfall
- consolidating, When changes require service orchestration-When changes involve new data requirements
- determining size of, Are My Services Too Small?
- domain, What Happens When Everything Works, How to Avoid This Pitfall-How to Avoid This Pitfall, Pros
- instances, Supervisor Consumer Pattern
- orchestration, When changes require service orchestration-When changes require service orchestration
- reliability metrics, Pros
- scope and function, Service scope and function-Service scope and function
- shared, Degraded
- triggering events, How Many Events Should I Trigger from a Single Service?-How Many Events Should I Trigger from a Single Service?

shared code, Shared code, Repairing a monolithic stovepipe architecture

shared kernel, BFF Granularity

shared libraries, Degraded

shared services, Degraded

side effects, How Should I Manage My Workflow State?

sidecar

- domain components in, Applying the Pattern
- ownership models, How to Govern This Pattern
- and service mesh, Service Mesh and Sidecar-Service Mesh and Sidecar

Sidecar pattern, Service Mesh and Sidecar-Service Mesh and Sidecar, Data Mesh and Cooperative Quantum

significant events, Should I Trigger an Event If No One Is Listening for It (Yet)? (see also derived events)

simplicity, Simplicity, Reflect what you want to measure

single backend, Backend for Frontend Pattern, How Can I Support Multiple User-Interface Types?

single broker, Domain-Broker Pattern

single consumer, How Can I Guarantee Message Processing Order?

single in-memory cache, Caching Topologies, Influences

single point of failure, How Many Brokers Should I Have in My Distributed Architecture?, Improved, Cons, Influences

single source of truth, Pros

Single-Responsibility Principle (SRP), Repairing a distributed Stovepipe Architecture

slip box (see Zettelkasten)

SLIs (service-level indicators), How to Govern This Pattern

SLOs (service-level objectives), Implementing the Pattern, How to Govern This Pattern

snapshots, Reading Events, Event Volume and Scale, Related Patterns, Antipatterns, and Pitfalls, Related Patterns, Antipatterns, and Pitfalls

SOA (service-oriented architecture), Degraded

soft state, ACID Versus BASE

software architecture, Architecture by Implication Antipattern

Software Architecture: The Hard Parts (Richards et al.)

- granularity disintegrators and integrators in, How to Avoid This Pitfall, How to Avoid This Antipattern, Service Granularity
- qualitative analysis of sagas in, Qualitative Analysis-Building Qualitative Analysis
- refactoring, How to Avoid This Antipattern
- refactorings described in, Repairing a monolithic stovepipe architecture
- saga types analyzed in, Applying the Pattern
- Sysops Squad kata in, Why Is This an Antipattern?
- transactional forces covered in, Influences, How Should I Manage My Workflow State?, Influences, How Should I Implement Atomicity in My Distributed Architectures?

source of truth, The Cache Is Not in the Critical Path, Should I Capture My Data as an Event Stream?, Applying the Pattern, Cache in the critical path

spectrum of patterns, antipatterns, and pitfalls, The Pattern, Antipattern, Pitfall Spectrum

SRP (Single-Responsibility Principle), Repairing a distributed Stovepipe Architecture

stale data, Handling Cache Failures

star schema, Data Mesh and Cooperative Quantum

stars and icons, Stars and Other Icons-Reflect what you want to measure

state component, Testability

state holder (see workflow state owner)

state tracker (see workflow state owner)

state-change event channel, Applying the Pattern-Applying the Pattern

static analysis tools, How to Govern This Pattern

static coupling

- in Choreography pattern, Architecture Quantum
- defined, Static coupling-Static coupling
- in Domain-Broker pattern, Architecture Quantum
- in Event Sourcing pattern, Architecture Quantum
- in Explicit Workflow State pattern, Influences
- in Implicit Workflow State pattern, Influences
- in Nexus Architecture pattern, Influences
- in Orchestration pattern, Architecture Quantum
- in quantum scope, Events Versus Messages
- in Service Sharding pattern, How Should I Implement Atomicity in My Distributed Architectures?, Architecture Quantum

store component, Applying the Pattern

store implementation, Loader and store implementation

Strategy Design pattern, Architecture Patterns Versus Design Patterns

streaming brokers, How to Govern This Pattern

stressor analysis, Iterative Architecture and Residuality Theory

stressors, Iterative Architecture and Residuality Theory

strict contracts, Applying the Pattern, How Specific Should My Contract Be?, Contract Management, Contract coupling

strong consistency, Caching Topologies, Consistency Model and Atomicity, Write-Behind Cache Pattern Variant, Influences

structural refactoring patterns, Repairing a monolithic stovepipe architecture

styles

    architectural, Building Qualitative Analysis, Stars and Other Icons, Events Versus Messages

    versus patterns, Styles Versus Patterns-Styles Versus Patterns

Super Supervisor component, Macro Variant-Macro Variant

Supervisor component, Applying the Pattern-Applying the Pattern

Swarm of Gnats trap, Applying the Pattern

switching logic, Applying the Pattern

synchronization mechanism, Applying the Pattern-A note on eventual consistency, Team Topologies

synchronous communication

    in Choreography pattern, Communication (Synchronous Versus Asynchronous)-Degraded

in Compensating Updates pattern, What Level of Consistency Is Proper?

creating operational coupling, Dynamic coupling

in Key-Based Event Contract pattern, Domain-Based Database Topology

in Orchestration pattern, The Case Where Everything Works, Communication (Synchronous Versus Asynchronous)-Degraded

synchronous send, Synchronous Send, Degraded

Sysops Squad architectural kata, Why Is This an Antipattern?-Why Is This an Antipattern?, Why Is This a Pitfall?

system of record, Cons

system-to-system integration, Integration Architecture

systems integration, Influences

T

team coordination, Consider a Different Architecture Style, Influences, Pros

team topologies

in Async Chain of Persistence pattern, Influences

in Backend for Frontend pattern, Influences

in CQRS pattern, Team Topologies

in Nexus Architecture pattern, Pros

technical abstraction, Why Is This an Antipattern? (see also domain abstraction)

technical breadth, How to Avoid This Pitfall-How to Avoid This Pitfall

technical debt, Cons

technical depth, How to Avoid This Pitfall-How to Avoid This Pitfall

technical partitioning

in layered architectures, Layered Architecture, Consider a Different Architecture Style

in Nexus Architecture pattern, Improved

versus domain partitioning, Applying the Pattern

technology radar (see personal radar)

temporal coupling

defined, Temporal coupling-Temporal coupling

in Domain-Broker pattern, Architecture Quantum

in Event Sourcing pattern, Architecture Quantum

in Implicit Workflow State pattern, Influences

in Nexus Architecture pattern, Influences

in Service Sharding pattern, How Should I Implement Atomicity in My Distributed Architectures?, Architecture Quantum

terminology, Common Definitions, Fundamental Definitions-Automating Architectural Governance: Fitness Functions

testability, Testability

testing frameworks, Automating Architectural Governance: Fitness Functions

Third Law of Software Architecture, The Pattern, Antipattern, Pitfall Spectrum

Thoughtworks Technology Radar, Developing a Personal Radar (see also personal radar)

thread and memory usage, How Can I Improve Throughput and Scalability?, How to Govern This Pattern

thread saturation, How to Govern This Pattern, How to Govern This Pattern

throughput, Throughput, Scalability and throughput, How Can I Improve Throughput and Scalability?-How Can I Improve Throughput and Scalability?

thundering herd, Cons

tight coupling, Static coupling

time travel, Pros

time-to-live (TTL), Conflicting Patterns, Pros, Influences

tracing, How to Govern This Pattern

trade-off analysis, How Is This Book Different from Other Patterns Books?, Why Is This a Pitfall? (see also qualitative analysis; quantitative analysis)

trade-off charts, Architecture Patterns Versus Design Patterns, When not to use stars

trade-offs for patterns evaluation

- choosing capabilities, Choosing Capabilities
- evaluating own patterns, Evaluating Your Own Patterns-Improved/degraded
- identifying influences, Identifying Influences
- overview of, Evaluating Trade-Offs for Patterns-Evaluating Trade-Offs for Patterns

stars and other icons, Stars and Other Icons-Reflect what you want to measure

using Zettelkasten, Using a Zettelkasten (Slip Box)-Using a Zettelkasten (Slip Box)

transactional context (see atomic context)

transactional coupling (see atomic coupling)

Transactional Outbox pattern, A note on eventual consistency

transactions

ACID, Influences, Compensating Updates Pattern, Database transactions, Influences

boundary, Identifying Events and Aggregates

cross-service, Service Sharding Pattern-How Should I Implement Atomicity in My Distributed Architectures?

database, Database transactions

defined, ACID Versus BASE

distributed, How Do I Roll Back a Distributed Transaction?, Why Is This a Pitfall?, Database transactions, When changes require service orchestration

TTL (time-to-live), Conflicting Patterns, Pros, Influences

20-minute rule, How to Avoid This Pitfall

U

ubiquitous language, The Value of Patterns

UIs (user interfaces), How Can I Support Multiple User-Interface Types?-How Can I Support Multiple User-Interface Types?

uncommitted events, Event generation

unhappy path, Applying the Pattern-The Case Where Something Breaks

upcasting, Schema Evolution

update requests, Applying the Pattern

updates

in caching, Writes/Updates-Writes/Updates

compensating, Atomicity (Data Consistency and Integrity), Implementing the Pattern, Failure to Achieve the Compensating Update-Failure to Achieve the Compensating Update, Why Is This a Pitfall?

in Read-Through/Write-Through Cache pattern, Writes/Updates-Writes/Updates

URL filtering, URL Filtering for Query Requests (see also GraphQL)

V

versioning, Cons, Schema Evolution-Schema Evolution, Architecture Quantum, Contract Management

Vlissides, John, Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls

volatility

in Grains of Sand pitfall, Code volatility, Shared code

in Microservice All the Things pitfall, How to Avoid This Pitfall

in Nexus Architecture pattern, Service Mesh and Sidecar

in Overcentralization antipattern, Why Is This an Antipattern?-Cons, Am I in This Antipattern?

of reuse targets, Reusability

volatility-based decomposition, How to Avoid This Antipattern

W

Wire Tap pattern, Payload Size and Message Volumes (see also architecture fitness functions)

Woolf, Bobby, Introduction: Fundamental Patterns and Antipatterns…with Some Pitfalls

Workflow Processor, Applying the Pattern

workflow state

in Choreography pattern, What Happens When an Error Occurs?

in Cross-Cutting State pattern, How Should I Manage My Workflow State?-How Should I Manage My Workflow State?

in Domain-Broker pattern, Workflow State

in Explicit Workflow State pattern, How Should I Manage My Workflow State?-How Should I Manage My Workflow State?, Pros

in Front Controller pattern, How Do I Handle Mediation Without an Explicit Orchestrator?, Applying the Pattern

in Implicit Workflow State pattern, How Should I Manage My Workflow State?-How Should I Manage My Workflow State?

in Orchestration pattern, How Should I Manage My Workflows?

workflow state owner

in Domain-Broker pattern, Workflow State

in Explicit Workflow State pattern, Applying the Pattern, Degraded

in Implicit Workflow State pattern, Applying the Pattern, Applying the Pattern, Degraded

workflows

communication, What Happens When Everything Works-What Happens When Everything Works

complex, Pros

defined, Workflow

event processor, Workflow Event Processor-Workflow Event Processor

as granularity integrators, Workflow and choreography

hybrid, Hybrids

logic, What Happens When an Error Occurs?

management of, How Should I Manage My Workflows?, How Should I Manage My Workflows?-How Should I Manage My Workflows?

pseudotransactional, Pros

write datastore, Applying the Pattern

Write-Around Caching pattern, Write-Around Caching Pattern Variant-Write-Around Caching Pattern Variant

Write-Behind Cache pattern, Write-Behind Cache Pattern Variant-Write-Behind Cache Pattern Variant, How to Govern This Pattern

writer (see store component)

writes

in caching, Writes/Updates-Writes/Updates

optimization of data for, How Can I Optimize My Data for Reads and Writes?

in Read-Through/Write-Through Cache pattern, Writes/Updates-Writes/Updates

X

x-max-priority property, How Do I Prioritize My Messages?

Z

Zettelkasten, Using a Zettelkasten (Slip Box)-Using a Zettelkasten (Slip Box)

About the Authors

Mark Richards is an experienced, hands-on software architect involved in the architecture, design, and implementation of microservice architectures and other distributed systems in a variety of technologies. He has been in the software industry since 1983 and has significant experience and expertise in application, integration, and enterprise architecture. Mark is the founder of *DeveloperToArchitect.com*, a free website devoted to helping developers in the journey to becoming a software architect. In addition to hands-on consulting, Mark has authored numerous technical books and videos, including his two latest books, *Fundamentals of Software Architecture* and *Software Architecture: The Hard Parts*, which he coauthored with Neal Ford. Mark has spoken at hundreds of conferences and user groups around the world on a variety of enterprise-related technical topics.

Neal Ford is an independent software consultant, bestselling author, and international speaker. He is an internationally recognized expert on software development and delivery, especially in the intersection of agile engineering techniques and software architecture. Neal has authored magazine articles, nine books (and counting), dozens of video presentations, and spoken at hundreds of developer conferences worldwide. His topics include software architecture, continuous delivery, functional programming, and cutting-edge software engineering, and his work includes a business-focused book and video on improving technical presentations. He is an innovator in building concrete verification mechanisms for software architecture, for humans or agents. Check out his website at *nealford.com*.

Raju Gandhi is the founder of DefMacro Software as well as a consultant, author, teacher, and speaker at conferences around the world. As a software developer and teacher, he believes in keeping things simple, preferring to understand and explain the "why" as opposed to the "how." Raju blogs at *LooselyTyped.com* and lives in Ohio with his wife, Michelle, their three children, Mason, Micah, and Delphine, and furry family members Buddy, Skye, and Princess Zara. Find his contact information at *Rajugandhi.com*. He's always looking to make new friends.

Colophon

The animal on the cover of *Software Architecture Patterns, Antipatterns, and Pitfalls* is a rufous-bellied seedsnipe (*Attagis gayi*), a stout, hardy shorebird native to the high-elevation alpine terrain of the Andes Mountains across western South America.

The seedsnipe primarily inhabits extreme environments often reaching up to the permanent snow line — ranging from 4,000 to 5,500 meters in the north down to 1,000 meters in southern Patagonia — where it frequents bogs, alpine cushion plant communities, and rocky slopes.

Measuring approximately 10 to 12 inches in length and weighing just under 1 pound, on average, it is the largest member of the seedsnipe family. This bird has a plump, partridge-like body with short legs and pointed wings. Both sexes look alike, displaying intricately scalloped rufous-brown and blackish patterns across their upperparts, which provide camouflage against montane gravel, paired with a warm, rust-colored underbelly.

Unlike most shorebirds, seedsnipes have adapted to a herbivorous diet, feeding primarily on plant matter such as the seeds, leaves, and buds of hardy alpine flora. They are strong fliers that take off in rapid, erratic zigzag patterns when startled, often emitting a musical call.

During the breeding season, which generally runs from September to December depending on latitude, pairs form and nest on the ground. The female lays a clutch of typically four eggs in a shallow scrape lined with sparse plant debris. The chicks hatch precocial, covered in cryptic down, and are quickly able to walk and forage under parental supervision.

The International Union for Conservation of Nature (IUCN) classifies the rufous-bellied seedsnipe as a species of Least Concern. Although local populations face occasional pressure from hunting near mining communities, its overall population remains stable across remote, protected habitats. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The color illustration is by Monica Kamsvaag, based on an antique line engraving from Lydekker's *Royal Natural History*. 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.