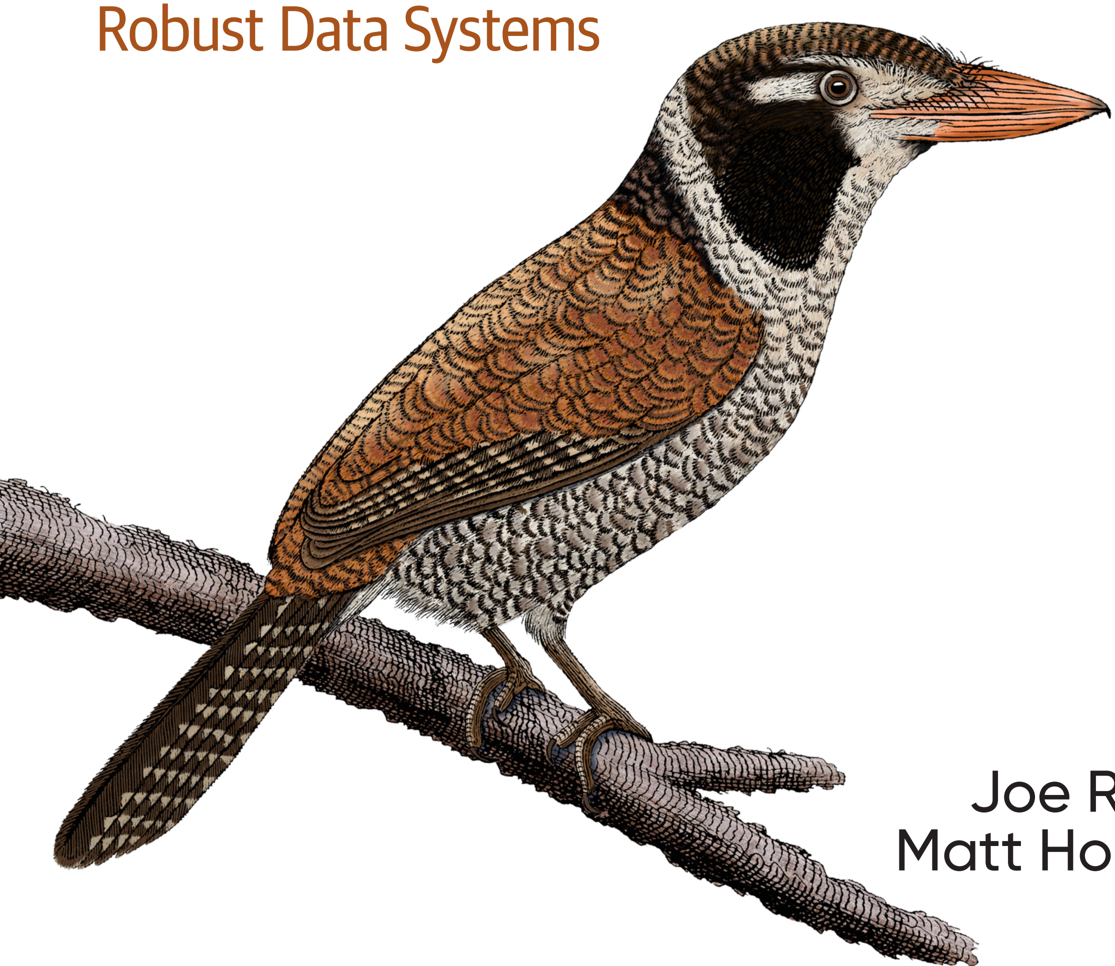
O'REILLY®

Fundamentals of Data Engineering

Plan and Build Robust Data Systems



Joe Reis &
Matt Housley

O'REILLY®

Fundamentals of Data Engineering

Data engineering has grown rapidly in the past decade, leaving many software engineers, data scientists, and analysts looking for a comprehensive view of this practice. With this practical book, you'll learn how to plan and build systems to serve the needs of your organization and customers by evaluating the best technologies available through the framework of the data engineering lifecycle.

Authors Joe Reis and Matt Housley walk you through the data engineering lifecycle and show you how to stitch together a variety of cloud technologies to serve the needs of downstream data consumers. You'll understand how to apply the concepts of data generation, ingestion, orchestration, transformation, storage, and governance that are critical in any data environment regardless of the underlying technology.

This book will help you:

- Get a concise overview of the entire data engineering landscape
- Assess data engineering problems using an end-to-end framework of best practices
- Cut through marketing hype when choosing data technologies, architecture, and processes
- Use the data engineering lifecycle to design and build a robust architecture
- Incorporate data governance and security across the data engineering lifecycle

Joe Reis is a "recovering data scientist," and a data engineer and architect.

Matt Housley is a data engineering consultant and cloud specialist.

> "The world of data has been evolving for a while now. First there were designers. Then database administrators. Then CIOs. Then data architects. This book signals the next step in the evolution and maturity of the industry. It is a must read for anyone who takes their profession and career honestly."
>
> **—Bill Inmon**
> creator of the data warehouse

> "*Fundamentals of Data Engineering* is a great introduction to the business of moving, processing, and handling data. I'd highly recommend it for anyone wanting to get up to speed in data engineering or analytics, or for existing practitioners who want to fill in any gaps in their understanding."
>
> **—Jordan Tigani**
> founder and CEO, MotherDuck, and founding engineer and cocreator of BigQuery

DATA

US $69.99 CAN $87.99
ISBN: 978-1-098-10830-4

9 781098 108304 56999

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

Praise for *Fundamentals of Data Engineering*

The world of data has been evolving for a while now. First there were designers. Then database administrators. Then CIOs. Then data architects. This book signals the next step in the evolution and maturity of the industry. It is a must read for anyone who takes their profession and career honestly.

—Bill Inmon, creator of the data warehouse

Fundamentals of Data Engineering is a great introduction to the business of moving, processing, and handling data. It explains the taxonomy of data concepts, without focusing too heavily on individual tools or vendors, so the techniques and ideas should outlast any individual trend or product. I'd highly recommend it for anyone wanting to get up to speed in data engineering or analytics, or for existing practitioners who want to fill in any gaps in their understanding.

—Jordan Tigani, founder and CEO, MotherDuck, and founding engineer and cocreator of BigQuery

If you want to lead in your industry, you must build the capabilities required to provide exceptional customer and employee experiences. This is not just a technology problem. It's a people opportunity. And it will transform your business. Data engineers are at the center of this transformation. But today the discipline is misunderstood. This book will demystify data engineering and become your ultimate guide to succeeding with data.

—Bruno Aziza, Head of Data Analytics, Google Cloud

What a book! Joe and Matt are giving you the answer to the question, "What must I understand to do data engineering?" Whether you are getting started as a data engineer or strengthening your skills, you are not looking for yet another technology handbook. You are seeking to learn more about the underlying principles and the core concepts of the role, its responsibilities, its technical and organizational environment, its mission—that's exactly what Joe and Matt offer in this book.

—Andy Petrella, founder of Kensu

This is the missing book in data engineering. A wonderfully thorough account of what it takes to be a good practicing data engineer, including thoughtful real-life considerations. I'd recommend all future education of data professionals include Joe and Matt's work.

—Sarah Krasnik, data engineering leader

It is incredible to realize the breadth of knowledge a data engineer must have. But don't let it scare you. This book provides a great foundational overview of various architectures, approaches, methodologies, and patterns that anyone working with data needs to be aware of. But what is even more valuable is that this book is full of golden nuggets of wisdom, best-practice advice, and things to consider when making decisions related to data engineering. It is a must read for both experienced and new data engineers.

—Veronika Durgin, data and analytics leader

I was honored and humbled to be asked by Joe and Matt to help technical review their masterpiece of data knowledge, *Fundamentals of Data Engineering*. Their ability to break down the key components that are critical to anyone wanting to move into a data engineering role is second to none. Their writing style makes the information easy to absorb, and they leave no stone unturned. It was an absolute pleasure to work with some of the best thought leaders in the data space. I can't wait to see what they do next.

—Chris Tabb, cofounder of LEIT DATA

Fundamentals of Data Engineering is the first book to take an in-depth and holistic look into the requirements of today's data engineer. As you'll see, the book dives into the critical areas of data engineering including skill sets, tools, and architectures used to manage, move, and curate data in today's complex technical environments.

More importantly, Joe and Matt convey their master of understanding data engineering and take the time to further dive into the more nuanced areas of data engineering and make it relatable to the reader. Whether you're a manager, experienced data engineer, or someone wanting to get into the space, this book provides practical insight into today's data engineering landscape.

—Jon King, Principal Data Architect

Two things will remain relevant to data engineers in 2042: SQL and this book. Joe and Matt cut through the hype around tools to extract the slowly changing dimensions of our discipline. Whether you're starting your journey with data or adding stripes to your black belt, *Fundamentals of Data Engineering* lays the foundation for mastery.

—Kevin Hu, CEO of Metaplane

In a field that is rapidly changing, with new technology solutions popping up constantly, Joe and Matt provide clear, timeless guidance, focusing on the core concepts and foundational knowledge required to excel as a data engineer. This book is jam packed with information that will empower you to ask the right questions, understand trade-offs, and make the best decisions when designing your data architecture and implementing solutions across the data engineering lifecycle. Whether you're just considering becoming a data engineer or have been in the field for years, I guarantee you'll learn something from this book!

—Julie Price, Senior Product Manager, SingleStore

Fundamentals of Data Engineering isn't just an instruction manual—it teaches you how to think like a data engineer. Part history lesson, part theory, and part acquired knowledge from Joe and Matt's decades of experience, the book has definitely earned its place on every data professional's bookshelf.

—Scott Breitenother, founder and CEO, Brooklyn Data Co.

There is no other book that so comprehensively covers what it means to be a data engineer. Joe and Matt dive deep into responsibilities, impacts, architectural choices, and so much more. Despite talking about such complex topics, the book is easy to read and digest. A very powerful combination.

—Danny Leybzon, MLOps Architect

I wish this book was around years ago when I started working with data engineers. The wide coverage of the field makes the involved roles clear and builds empathy with the many roles it takes to build a competent data discipline.

—Tod Hansmann, VP Engineering

A must read and instant classic for anyone in the data engineering field. This book fills a gap in the current knowledge base, discussing fundamental topics not found in other books. You will gain understanding of foundational concepts and insight into historical context about data engineering that will set up anyone to succeed.

—Matthew Sharp, Data and ML Engineer

Data engineering is the foundation of every analysis, machine learning model, and data product, so it is critical that it is done well. There are countless manuals, books, and references for each of the technologies used by data engineers, but very few (if any) resources that provide a holistic view of what it means to work as a data engineer. This book fills a critical need in the industry and does it well, laying the foundation for new and working data engineers to be successful and effective in their roles. This is the book that I'll be recommending to anyone who wants to work with data at any level.

—*Tobias Macey, host of* The Data Engineering Podcast

Fundamentals of Data Engineering

Plan and Build Robust Data Systems

Joe Reis and Matt Housley

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Fundamentals of Data Engineering

by Joe Reis and Matt Housley

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: Jessica Haberman
Development Editor: Michele Cronin
Production Editor: Gregory Hyman
Copyeditor: Sharon Wilkey
Proofreader: Amnet Systems, LLC

Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

July 2022: First Edition

Revision History for the First Edition

2022-06-22: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098108304* for release details.

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

978-1-098-10830-4

[LSI]

Table of Contents

Preface. xiii

Part I. Foundation and Building Blocks

1. Data Engineering Described. 3
What Is Data Engineering? 3
Data Engineering Defined 4
The Data Engineering Lifecycle 5
Evolution of the Data Engineer 6
Data Engineering and Data Science 11
Data Engineering Skills and Activities 13
Data Maturity and the Data Engineer 13
The Background and Skills of a Data Engineer 17
Business Responsibilities 18
Technical Responsibilities 19
The Continuum of Data Engineering Roles, from A to B 21
Data Engineers Inside an Organization 22
Internal-Facing Versus External-Facing Data Engineers 23
Data Engineers and Other Technical Roles 24
Data Engineers and Business Leadership 28
Conclusion 31
Additional Resources 32

2. The Data Engineering Lifecycle. 33
What Is the Data Engineering Lifecycle? 33
The Data Lifecycle Versus the Data Engineering Lifecycle 34
Generation: Source Systems 35

Storage 38
Ingestion 39
Transformation 43
Serving Data 44
Major Undercurrents Across the Data Engineering Lifecycle 48
Security 49
Data Management 50
DataOps 59
Data Architecture 64
Orchestration 64
Software Engineering 66
Conclusion 68
Additional Resources 69

3. Designing Good Data Architecture. 71
What Is Data Architecture? 71
Enterprise Architecture Defined 72
Data Architecture Defined 75
“Good” Data Architecture 76
Principles of Good Data Architecture 77
Principle 1: Choose Common Components Wisely 78
Principle 2: Plan for Failure 79
Principle 3: Architect for Scalability 80
Principle 4: Architecture Is Leadership 80
Principle 5: Always Be Architecting 81
Principle 6: Build Loosely Coupled Systems 81
Principle 7: Make Reversible Decisions 83
Principle 8: Prioritize Security 84
Principle 9: Embrace FinOps 85
Major Architecture Concepts 87
Domains and Services 87
Distributed Systems, Scalability, and Designing for Failure 88
Tight Versus Loose Coupling: Tiers, Monoliths, and Microservices 90
User Access: Single Versus Multitenant 94
Event-Driven Architecture 95
Brownfield Versus Greenfield Projects 96
Examples and Types of Data Architecture 98
Data Warehouse 98
Data Lake 101
Convergence, Next-Generation Data Lakes, and the Data Platform 102
Modern Data Stack 103
Lambda Architecture 104

Kappa Architecture 105
The Dataflow Model and Unified Batch and Streaming 105
Architecture for IoT 106
Data Mesh 109
Other Data Architecture Examples 110
Who's Involved with Designing a Data Architecture? 111
Conclusion 111
Additional Resources 111

4. Choosing Technologies Across the Data Engineering Lifecycle. 115
Team Size and Capabilities 116
Speed to Market 117
Interoperability 117
Cost Optimization and Business Value 118
Total Cost of Ownership 118
Total Opportunity Cost of Ownership 119
FinOps 120
Today Versus the Future: Immutable Versus Transitory Technologies 120
Our Advice 122
Location 123
On Premises 123
Cloud 124
Hybrid Cloud 127
Multicloud 128
Decentralized: Blockchain and the Edge 129
Our Advice 129
Cloud Repatriation Arguments 130
Build Versus Buy 132
Open Source Software 133
Proprietary Walled Gardens 137
Our Advice 138
Monolith Versus Modular 139
Monolith 139
Modularity 140
The Distributed Monolith Pattern 142
Our Advice 142
Serverless Versus Servers 143
Serverless 143
Containers 144
How to Evaluate Server Versus Serverless 145
Our Advice 146
Optimization, Performance, and the Benchmark Wars 147

Big Data...for the 1990s 148
Nonsensical Cost Comparisons 148
Asymmetric Optimization 148
Caveat Emptor 149
Undercurrents and Their Impacts on Choosing Technologies 149
Data Management 149
DataOps 149
Data Architecture 150
Orchestration Example: Airflow 150
Software Engineering 151
Conclusion 151
Additional Resources 151

Part II. The Data Engineering Lifecycle in Depth

5. **Data Generation in Source Systems. 155**
Sources of Data: How Is Data Created? 156
Source Systems: Main Ideas 156
Files and Unstructured Data 156
APIs 157
Application Databases (OLTP Systems) 157
Online Analytical Processing System 159
Change Data Capture 159
Logs 160
Database Logs 161
CRUD 162
Insert-Only 162
Messages and Streams 163
Types of Time 164
Source System Practical Details 165
Databases 166
APIs 174
Data Sharing 176
Third-Party Data Sources 177
Message Queues and Event-Streaming Platforms 177
Whom You'll Work With 181
Undercurrents and Their Impact on Source Systems 183
Security 183
Data Management 184
DataOps 184
Data Architecture 185

Orchestration 186
Software Engineering 187
Conclusion 187
Additional Resources 188

6. Storage. 189
Raw Ingredients of Data Storage 191
Magnetic Disk Drive 191
Solid-State Drive 193
Random Access Memory 194
Networking and CPU 195
Serialization 195
Compression 196
Caching 197
Data Storage Systems 197
Single Machine Versus Distributed Storage 198
Eventual Versus Strong Consistency 198
File Storage 199
Block Storage 202
Object Storage 205
Cache and Memory-Based Storage Systems 211
The Hadoop Distributed File System 211
Streaming Storage 212
Indexes, Partitioning, and Clustering 213
Data Engineering Storage Abstractions 215
The Data Warehouse 215
The Data Lake 216
The Data Lakehouse 216
Data Platforms 217
Stream-to-Batch Storage Architecture 217
Big Ideas and Trends in Storage 218
Data Catalog 218
Data Sharing 219
Schema 219
Separation of Compute from Storage 220
Data Storage Lifecycle and Data Retention 223
Single-Tenant Versus Multitenant Storage 226
Whom You'll Work With 227
Undercurrents 228
Security 228
Data Management 228
DataOps 229

Data Architecture 230
Orchestration 230
Software Engineering 230
Conclusion 230
Additional Resources 231

7. Ingestion. 233
What Is Data Ingestion? 234
Key Engineering Considerations for the Ingestion Phase 235
Bounded Versus Unbounded Data 236
Frequency 237
Synchronous Versus Asynchronous Ingestion 238
Serialization and Deserialization 239
Throughput and Scalability 239
Reliability and Durability 240
Payload 241
Push Versus Pull Versus Poll Patterns 244
Batch Ingestion Considerations 244
Snapshot or Differential Extraction 246
File-Based Export and Ingestion 246
ETL Versus ELT 246
Inserts, Updates, and Batch Size 247
Data Migration 247
Message and Stream Ingestion Considerations 248
Schema Evolution 248
Late-Arriving Data 248
Ordering and Multiple Delivery 248
Replay 249
Time to Live 249
Message Size 249
Error Handling and Dead-Letter Queues 249
Consumer Pull and Push 250
Location 250
Ways to Ingest Data 250
Direct Database Connection 251
Change Data Capture 252
APIs 254
Message Queues and Event-Streaming Platforms 255
Managed Data Connectors 256
Moving Data with Object Storage 257
EDI 257
Databases and File Export 257

Practical Issues with Common File Formats 258
Shell 258
SSH 259
SFTP and SCP 259
Webhooks 259
Web Interface 260
Web Scraping 260
Transfer Appliances for Data Migration 261
Data Sharing 262
Whom You'll Work With 262
Upstream Stakeholders 262
Downstream Stakeholders 263
Undercurrents 263
Security 264
Data Management 264
DataOps 266
Orchestration 268
Software Engineering 268
Conclusion 268
Additional Resources 269

8. Queries, Modeling, and Transformation. 271
Queries 272
What Is a Query? 273
The Life of a Query 274
The Query Optimizer 275
Improving Query Performance 275
Queries on Streaming Data 281
Data Modeling 287
What Is a Data Model? 288
Conceptual, Logical, and Physical Data Models 289
Normalization 290
Techniques for Modeling Batch Analytical Data 294
Modeling Streaming Data 307
Transformations 309
Batch Transformations 310
Materialized Views, Federation, and Query Virtualization 323
Streaming Transformations and Processing 326
Whom You'll Work With 329
Upstream Stakeholders 329
Downstream Stakeholders 330
Undercurrents 330

Security 330
Data Management 331
DataOps 332
Data Architecture 333
Orchestration 333
Software Engineering 333
Conclusion 334
Additional Resources 335

9. Serving Data for Analytics, Machine Learning, and Reverse ETL. 337
General Considerations for Serving Data 338
Trust 338
What's the Use Case, and Who's the User? 339
Data Products 340
Self-Service or Not? 341
Data Definitions and Logic 342
Data Mesh 343
Analytics 344
Business Analytics 344
Operational Analytics 346
Embedded Analytics 348
Machine Learning 349
What a Data Engineer Should Know About ML 350
Ways to Serve Data for Analytics and ML 351
File Exchange 351
Databases 352
Streaming Systems 354
Query Federation 354
Data Sharing 355
Semantic and Metrics Layers 355
Serving Data in Notebooks 356
Reverse ETL 358
Whom You'll Work With 360
Undercurrents 360
Security 361
Data Management 362
DataOps 362
Data Architecture 363
Orchestration 363
Software Engineering 364
Conclusion 365
Additional Resources 365

Part III. Security, Privacy, and the Future of Data Engineering

10. Security and Privacy. 369
People 370
The Power of Negative Thinking 370
Always Be Paranoid 370
Processes 371
Security Theater Versus Security Habit 371
Active Security 371
The Principle of Least Privilege 372
Shared Responsibility in the Cloud 372
Always Back Up Your Data 372
An Example Security Policy 373
Technology 374
Patch and Update Systems 374
Encryption 375
Logging, Monitoring, and Alerting 375
Network Access 376
Security for Low-Level Data Engineering 377
Conclusion 378
Additional Resources 378

11. The Future of Data Engineering. 379
The Data Engineering Lifecycle Isn't Going Away 380
The Decline of Complexity and the Rise of Easy-to-Use Data Tools 380
The Cloud-Scale Data OS and Improved Interoperability 381
"Enterprisey" Data Engineering 383
Titles and Responsibilities Will Morph... 384
Moving Beyond the Modern Data Stack, Toward the Live Data Stack 385
The Live Data Stack 385
Streaming Pipelines and Real-Time Analytical Databases 386
The Fusion of Data with Applications 387
The Tight Feedback Between Applications and ML 388
Dark Matter Data and the Rise of...Spreadsheets?! 388
Conclusion 389

A. Serialization and Compression Technical Details. 391

B. Cloud Networking. 399

Index. 403

Preface

How did this book come about? The origin is deeply rooted in our journey from data science into data engineering. We often jokingly refer to ourselves as *recovering data scientists*. We both had the experience of being assigned to data science projects, then struggling to execute these projects due to a lack of proper foundations. Our journey into data engineering began when we undertook data engineering tasks to build foundations and infrastructure.

With the rise of data science, companies splashed out lavishly on data science talent, hoping to reap rich rewards. Very often, data scientists struggled with basic problems that their background and training did not address—data collection, data cleansing, data access, data transformation, and data infrastructure. These are problems that data engineering aims to solve.

What This Book Isn't

Before we cover what this book is about and what you'll get out of it, let's quickly cover what this book *isn't*. This book isn't about data engineering using a particular tool, technology, or platform. While many excellent books approach data engineering technologies from this perspective, these books have a short shelf life. Instead, we focus on the fundamental concepts behind data engineering.

What This Book Is About

This book aims to fill a gap in current data engineering content and materials. While there's no shortage of technical resources that address specific data engineering tools and technologies, people struggle to understand how to assemble these components into a coherent whole that applies in the real world. This book connects the dots of the end-to-end data lifecycle. It shows you how to stitch together various technologies to serve the needs of downstream data consumers such as analysts, data scientists, and machine learning engineers. This book works as a complement to O'Reilly books that cover the details of particular technologies, platforms, and programming languages.

The big idea of this book is the *data engineering lifecycle*: data generation, storage, ingestion, transformation, and serving. Since the dawn of data, we've seen the rise and fall of innumerable specific technologies and vendor products, but the data engineering lifecycle stages have remained essentially unchanged. With this framework, the reader will come away with a sound understanding for applying technologies to real-world business problems.

Our goal here is to map out principles that reach across two axes. First, we wish to distill data engineering into principles that can encompass *any relevant technology*. Second, we wish to present principles that will stand the test of *time*. We hope that these ideas reflect lessons learned across the data technology upheaval of the last twenty years and that our mental framework will remain useful for a decade or more into the future.

One thing to note: we unapologetically take a cloud-first approach. We view the cloud as a fundamentally transformative development that will endure for decades; most on-premises data systems and workloads will eventually move to cloud hosting. We assume that infrastructure and systems are *ephemeral* and *scalable*, and that data engineers will lean toward deploying managed services in the cloud. That said, most concepts in this book will translate to non-cloud environments.

Who Should Read This Book

Our primary intended audience for this book consists of technical practitioners, mid- to senior-level software engineers, data scientists, or analysts interested in moving into data engineering; or data engineers working in the guts of specific technologies, but wanting to develop a more comprehensive perspective. Our secondary target audience consists of data stakeholders who work adjacent to technical practitioners—e.g., a data team lead with a technical background overseeing a team of data engineers, or a director of data warehousing wanting to migrate from on-premises technology to a cloud-based solution.

Ideally, you're curious and want to learn—why else would you be reading this book? You stay current with data technologies and trends by reading books and articles on data warehousing/data lakes, batch and streaming systems, orchestration, modeling, management, analysis, developments in cloud technologies, etc. This book will help you weave what you've read into a complete picture of data engineering across technologies and paradigms.

Prerequisites

We assume a good deal of familiarity with the types of data systems found in a corporate setting. In addition, we assume that readers have some familiarity with SQL and Python (or some other programming language), and experience with cloud services.

Numerous resources are available for aspiring data engineers to practice Python and SQL. Free online resources abound (blog posts, tutorial sites, YouTube videos), and many new Python books are published every year.

The cloud provides unprecedented opportunities to get hands-on experience with data tools. We suggest that aspiring data engineers set up accounts with cloud services such as AWS, Azure, Google Cloud Platform, Snowflake, Databricks, etc. Note that many of these platforms have *free tier* options, but readers should keep a close eye on costs and work with small quantities of data and single node clusters as they study.

Developing familiarity with corporate data systems outside of a corporate environment remains difficult, and this creates certain barriers for aspiring data engineers who have yet to land their first data job. This book can help. We suggest that data novices read for high-level ideas and then look at materials in the Additional Resources section at the end of each chapter. On a second read through, note any unfamiliar terms and technologies. You can utilize Google, Wikipedia, blog posts, YouTube videos, and vendor sites to become familiar with new terms and fill gaps in your understanding.

What You'll Learn and How It Will Improve Your Abilities

This book aims to help you build a solid foundation for solving real-world data engineering problems.

By the end of this book you will understand:

- How data engineering impacts your current role (data scientist, software engineer, or data team lead)
- How to cut through the marketing hype and choose the right technologies, data architecture, and processes
- How to use the data engineering lifecycle to design and build a robust architecture
- Best practices for each stage of the data lifecycle

And you will be able to:

- Incorporate data engineering principles in your current role (data scientist, analyst, software engineer, data team lead, etc.)
- Stitch together a variety of cloud technologies to serve the needs of downstream data consumers
- Assess data engineering problems with an end-to-end framework of best practices
- Incorporate data governance and security across the data engineering lifecycle

Navigating This Book

This book is composed of four parts:

- Part I, "Foundation and Building Blocks"
- Part II, "The Data Engineering Lifecycle in Depth"
- Part III, "Security, Privacy, and the Future of Data Engineering"
- Appendices A and B: covering serialization and compression, and cloud networking, respectively

In Part I, we begin by defining data engineering in Chapter 1, then map out the data engineering lifecycle in Chapter 2. In Chapter 3, we discuss *good architecture*. In Chapter 4, we introduce a framework for choosing the right technology—while we frequently see technology and architecture conflated, these are in fact very different topics.

Part II builds on Chapter 2 to cover the data engineering lifecycle in depth; each lifecycle stage—data generation, storage, ingestion, transformation and serving—is covered in its own chapter. Part II is arguably the heart of the book, and the other chapters exist to support the core ideas covered here.

Part III covers additional topics. In Chapter 10, we discuss *security and privacy*. While security has always been an important part of the data engineering profession, it has only become more critical with the rise of for profit hacking and state sponsored cyber attacks. And what can we say of privacy? The era of corporate privacy nihilism is over—no company wants to see its name appear in the headline of an article on sloppy privacy practices. Reckless handling of personal data can also have significant legal ramifications with the advent of GDPR, CCPA, and other regulations. In short, security and privacy must be top priorities in any data engineering work.

In the course of working in data engineering, doing research for this book and interviewing numerous experts, we thought a good deal about where the field is going in the near and long term. Chapter 11 outlines our highly speculative ideas on the future of data engineering. By its nature, the future is a slippery thing. Time will tell if some of our ideas are correct. We would love to hear from our readers on how their visions of the future agree with or differ from our own.

In the appendices, we cover a handful of technical topics that are extremely relevant to the day-to-day practice of data engineering but didn't fit into the main body of the text. Specifically, engineers need to understand serialization and compression (see Appendix A) both to work directly with data files and to assess performance considerations in data systems, and cloud networking (see Appendix B) is a critical topic as data engineering shifts into the cloud.

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.

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/fundamentals-of-data*.

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

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

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

Follow us on Twitter: *https://twitter.com/oreillymedia*

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

Acknowledgments

When we started writing this book, we were warned by many people that we faced a hard task. A book like this has a lot of moving parts, and due to its comprehensive view of the field of data engineering, it required a ton of research, interviews, discussions, and deep thinking. We won't claim to have captured every nuance of data engineering, but we hope that the results resonate with you. Numerous individuals contributed to our efforts, and we're grateful for the support we received from many experts.

First, thanks to our amazing crew of technical reviewers. They slogged through many readings and gave invaluable (and often ruthlessly blunt) feedback. This book would be a fraction of itself without their efforts. In no particular order, we give endless thanks to Bill Inmon, Andy Petrella, Matt Sharp, Tod Hanseman, Chris Tabb, Danny Lebzyon, Martin Kleppman, Scott Lorimor, Nick Schrock, Lisa Steckman, Veronika Durgin, and Alex Woolford.

Second, we've had a unique opportunity to talk with the leading experts in the field of data on our live shows, podcasts, meetups, and endless private calls. Their ideas helped shape our book. There are too many people to name individually, but we'd like to give shoutouts to Jordan Tigani, Zhamak Dehghani, Ananth Packkildurai, Shruti Bhat, Eric Tschetter, Benn Stancil, Kevin Hu, Michael Rogove, Ryan Wright, Adi Polak, Shinji Kim, Andreas Kretz, Egor Gryaznov, Chad Sanderson, Julie Price, Matt Turck, Monica Rogati, Mars Lan, Pardhu Gunnam, Brian Suk, Barr Moses, Lior Gavish, Bruno Aziza, Gian Merlino, DeVaris Brown, Todd Beauchene, Tudor Girba, Scott Taylor, Ori Rafael, Lee Edwards, Bryan Offutt, Ollie Hughes, Gilbert Eijkelenboom, Chris Bergh, Fabiana Clemente, Andreas Kretz, Ori Reshef, Nick Singh, Mark Balkenende, Kenten Danas, Brian Olsen, Lior Gavish, Rhaghu Murthy, Greg Coquillo, David Aponte, Demetrios Brinkmann, Sarah Catanzaro, Michel Tricot, Levi Davis, Ted Walker, Carlos Kemeny, Josh Benamram, Chanin Nantasenamat, George Firican, Jordan Goldmeir, Minhaaj Rehmam, Luigi Patruno, Vin Vashista, Danny Ma, Jesse Anderson, Alessya Visnjic, Vishal Singh, Dave Langer, Roy Hasson, Todd Odess, Che Sharma, Scott Breitenother, Ben Taylor, Thom Ives, John Thompson, Brent Dykes, Josh Tobin, Mark Kosiba, Tyler Pugliese, Douwe Maan, Martin Traverso, Curtis Kowalski, Bob Davis, Koo Ping Shung, Ed Chenard, Matt Sciorma, Tyler Folkman, Jeff Baird, Tejas Manohar, Paul Singman, Kevin Stumpf, Willem Pineaar, and Michael Del Balso from Tecton, Emma Dahl, Harpreet Sahota, Ken Jee, Scott Taylor, Kate Strachnyi, Kristen Kehrer, Taylor Miller, Abe Gong, Ben Castleton, Ben Rogojan, David Mertz, Emmanuel Raj, Andrew Jones, Avery Smith, Brock Cooper, Jeff Larson, Jon King, Holden Ackerman, Miriah Peterson, Felipe Hoffa, David Gonzalez, Richard Wellman, Susan Walsh, Ravit Jain, Lauren Balik, Mikiko Bazeley, Mark Freeman, Mike Wimmer, Alexey Shchedrin, Mary Clair Thompson, Julie Burroughs, Jason Pedley, Freddy Drennan, Jason Pedley, Kelly and Matt Phillipps, Brian Campbell, Faris Chebib, Dylan Gregerson, Ken Myers, Jake Carter, Seth Paul, Ethan Aaron, and many others.

If you're not mentioned specifically, don't take it personally. You know who you are. Let us know and we'll get you on the next edition.

We'd also like to thank the Ternary Data team (Colleen McAuley, Maike Wells, Patrick Dahl, Aaron Hunsaker, and others), our students, and the countless people around the world who've supported us. It's a great reminder the world is a very small place.

Working with the O'Reilly crew was amazing! Special thanks to Jess Haberman for having confidence in us during the book proposal process, our amazing and extremely patient development editors Nicole Taché and Michele Cronin for invaluable editing, feedback, and support. Thank you also to the superb production team at O'Reilly (Greg and crew).

Joe would like to thank his family—Cassie, Milo, and Ethan—for letting him write a book. They had to endure a ton, and Joe promises to never write a book again. ;)

Matt would like to thank his friends and family for their enduring patience and support. He's still hopeful that Seneca will deign to give a five-star review after a good deal of toil and missed family time around the holidays.

PART I

Foundation and Building Blocks

CHAPTER 3

Designing Good Data Architecture

Good data architecture provides seamless capabilities across every step of the data lifecycle and undercurrent. We'll begin by defining *data architecture* and then discuss components and considerations. We'll then touch on specific batch patterns (data warehouses, data lakes), streaming patterns, and patterns that unify batch and streaming. Throughout, we'll emphasize leveraging the capabilities of the cloud to deliver scalability, availability, and reliability.

What Is Data Architecture?

Successful data engineering is built upon rock-solid data architecture. This chapter aims to review a few popular architecture approaches and frameworks, and then craft our opinionated definition of what makes "good" data architecture. Indeed, we won't make everyone happy. Still, we will lay out a pragmatic, domain-specific, working definition for *data architecture* that we think will work for companies of vastly different scales, business processes, and needs.

What is data architecture? When you stop to unpack it, the topic becomes a bit murky; researching data architecture yields many inconsistent and often outdated definitions. It's a lot like when we defined *data engineering* in Chapter 1—there's no consensus. In a field that is constantly changing, this is to be expected. So what do we mean by *data architecture* for the purposes of this book? Before defining the term, it's essential to understand the context in which it sits. Let's briefly cover enterprise architecture, which will frame our definition of data architecture.

Enterprise Architecture Defined

Enterprise architecture has many subsets, including business, technical, application, and data (Figure 3-1). As such, many frameworks and resources are devoted to enterprise architecture. In truth, architecture is a surprisingly controversial topic.

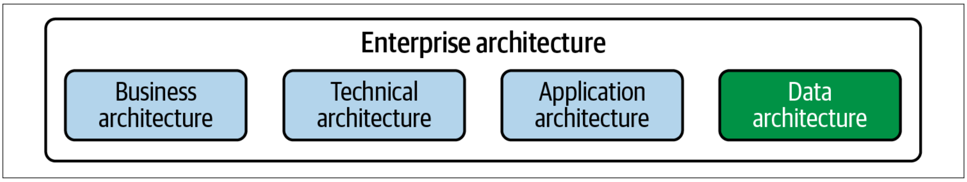


Figure 3-1. Data architecture is a subset of enterprise architecture

The term *enterprise* gets mixed reactions. It brings to mind sterile corporate offices, command-and-control/waterfall planning, stagnant business cultures, and empty catchphrases. Even so, we can learn some things here.

Before we define and describe *enterprise architecture*, let's unpack this term. Let's look at how enterprise architecture is defined by some significant thought leaders: TOGAF, Gartner, and EABOK.

TOGAF's definition

TOGAF is *The Open Group Architecture Framework*, a standard of The Open Group. It's touted as the most widely used architecture framework today. Here's the TOGAF definition:[1]

> The term "enterprise" in the context of "enterprise architecture" can denote an entire enterprise—encompassing all of its information and technology services, processes, and infrastructure—or a specific domain within the enterprise. In both cases, the architecture crosses multiple systems, and multiple functional groups within the enterprise.

Gartner's definition

Gartner is a global research and advisory company that produces research articles and reports on trends related to enterprises. Among other things, it is responsible for the (in)famous Gartner Hype Cycle. Gartner's definition is as follows:[2]

> Enterprise architecture (EA) is a discipline for proactively and holistically leading enterprise responses to disruptive forces by identifying and analyzing the execution of change toward desired business vision and outcomes. EA delivers value by presenting

[1] The Open Group, *TOGAF Version 9.1*, *https://oreil.ly/A1H67*.

[2] Gartner Glossary, s.v. "Enterprise Architecture (EA)," *https://oreil.ly/SWwQF*.

> business and IT leaders with signature-ready recommendations for adjusting policies and projects to achieve targeted business outcomes that capitalize on relevant business disruptions.

EABOK's definition

EABOK is the *Enterprise Architecture Book of Knowledge*, an enterprise architecture reference produced by the MITRE Corporation. EABOK was released as an incomplete draft in 2004 and has not been updated since. Though seemingly obsolete, EABOK is frequently referenced in descriptions of enterprise architecture; we found many of its ideas helpful while writing this book. Here's the EABOK definition:[3]

> Enterprise Architecture (EA) is an organizational model; an abstract representation of an Enterprise that aligns strategy, operations, and technology to create a roadmap for success.

Our definition

We extract a few common threads in these definitions of enterprise architecture: change, alignment, organization, opportunities, problem-solving, and migration. Here is our definition of *enterprise architecture*, one that we feel is more relevant to today's fast-moving data landscape:

> Enterprise architecture is the design of systems to support *change in the enterprise*, achieved by *flexible and reversible decisions* reached through careful *evaluation of trade-offs*.

Here, we touch on some key areas we'll return to throughout the book: flexible and reversible decisions, change management, and evaluation of trade-offs. We discuss each theme at length in this section and then make the definition more concrete in the latter part of the chapter by giving various examples of data architecture.

Flexible and reversible decisions are essential for two reasons. First, the world is constantly changing, and predicting the future is impossible. Reversible decisions allow you to adjust course as the world changes and you gather new information. Second, there is a natural tendency toward enterprise ossification as organizations grow. Adopting a culture of reversible decisions helps overcome this tendency by reducing the risk attached to a decision.

Jeff Bezos is credited with the idea of one-way and two-way doors.[4] A *one-way door* is a decision that is almost impossible to reverse. For example, Amazon could have

[3] EABOK Consortium website, *https://eabok.org*.

[4] Jeff Haden, "Amazon Founder Jeff Bezos: This Is How Successful People Make Such Smart Decisions," *Inc.*, December 3, 2018, *https://oreil.ly/QwIm0*.

decided to sell AWS or shut it down. It would be nearly impossible for Amazon to rebuild a public cloud with the same market position after such an action.

On the other hand, a *two-way door* is an easily reversible decision: you walk through and proceed if you like what you see in the room or step back through the door if you don't. Amazon might decide to require the use of DynamoDB for a new microservices database. If this policy doesn't work, Amazon has the option of reversing it and refactoring some services to use other databases. Since the stakes attached to each reversible decision (two-way door) are low, organizations can make more decisions, iterating, improving, and collecting data rapidly.

Change management is closely related to reversible decisions and is a central theme of enterprise architecture frameworks. Even with an emphasis on reversible decisions, enterprises often need to undertake large initiatives. These are ideally broken into smaller changes, each one a reversible decision in itself. Returning to Amazon, we note a five-year gap (2007 to 2012) from the publication of a paper on the DynamoDB concept to Werner Vogels's announcement of the DynamoDB service on AWS. Behind the scenes, teams took numerous small actions to make DynamoDB a concrete reality for AWS customers. Managing such small actions is at the heart of change management.

Architects are not simply mapping out IT processes and vaguely looking toward a distant, utopian future; they actively solve business problems and create new opportunities. Technical solutions exist not for their own sake but in support of business goals. Architects identify problems in the current state (poor data quality, scalability limits, money-losing lines of business), define desired future states (agile data-quality improvement, scalable cloud data solutions, improved business processes), and realize initiatives through execution of small, concrete steps. It bears repeating:

> *Technical solutions exist not for their own sake but in support of business goals.*

We found significant inspiration in *Fundamentals of Software Architecture* by Mark Richards and Neal Ford (O'Reilly). They emphasize that trade-offs are inevitable and ubiquitous in the engineering space. Sometimes the relatively fluid nature of software and data leads us to believe that we are freed from the constraints that engineers face in the hard, cold physical world. Indeed, this is partially true; patching a software bug is much easier than redesigning and replacing an airplane wing. However, digital systems are ultimately constrained by physical limits such as latency, reliability, density, and energy consumption. Engineers also confront various nonphysical limits, such as characteristics of programming languages and frameworks, and practical constraints in managing complexity, budgets, etc. Magical thinking culminates in poor engineering. Data engineers must account for trade-offs at every step to design an optimal system while minimizing high-interest technical debt.

Let's reiterate one central point in our enterprise architecture definition: enterprise architecture balances flexibility and trade-offs. This isn't always an easy balance, and architects must constantly assess and reevaluate with the recognition that the world is dynamic. Given the pace of change that enterprises are faced with, organizations—and their architecture—cannot afford to stand still.

Data Architecture Defined

Now that you understand enterprise architecture, let's dive into data architecture by establishing a working definition that will set the stage for the rest of the book. *Data architecture* is a subset of enterprise architecture, inheriting its properties: processes, strategy, change management, and evaluating trade-offs. Here are a couple of definitions of data architecture that influence our definition.

TOGAF's definition

TOGAF defines data architecture as follows:[5]

> A description of the structure and interaction of the enterprise's major types and sources of data, logical data assets, physical data assets, and data management resources.

DAMA's definition

The DAMA *DMBOK* defines data architecture as follows:[6]

> Identifying the data needs of the enterprise (regardless of structure) and designing and maintaining the master blueprints to meet those needs. Using master blueprints to guide data integration, control data assets, and align data investments with business strategy.

Our definition

Considering the preceding two definitions and our experience, we have crafted our definition of *data architecture*:

> Data architecture is the design of systems to support the evolving data needs of an enterprise, achieved by flexible and reversible decisions reached through a careful evaluation of trade-offs.

How does data architecture fit into data engineering? Just as the data engineering lifecycle is a subset of the data lifecycle, data engineering architecture is a subset of general data architecture. *Data engineering architecture* is the systems and frameworks

[5] The Open Group, *TOGAF Version 9.1*, *https://oreil.ly/A1H67*.

[6] *DAMA - DMBOK: Data Management Body of Knowledge*, 2nd ed. (Technics Publications, 2017).

that make up the key sections of the data engineering lifecycle. We'll use *data architecture* interchangeably with *data engineering architecture* throughout this book.

Other aspects of data architecture that you should be aware of are operational and technical (Figure 3-2). *Operational architecture* encompasses the functional requirements of what needs to happen related to people, processes, and technology. For example, what business processes does the data serve? How does the organization manage data quality? What is the latency requirement from when the data is produced to when it becomes available to query? *Technical architecture* outlines how data is ingested, stored, transformed, and served along the data engineering lifecycle. For instance, how will you move 10 TB of data every hour from a source database to your data lake? In short, operational architecture describes *what* needs to be done, and technical architecture details *how* it will happen.

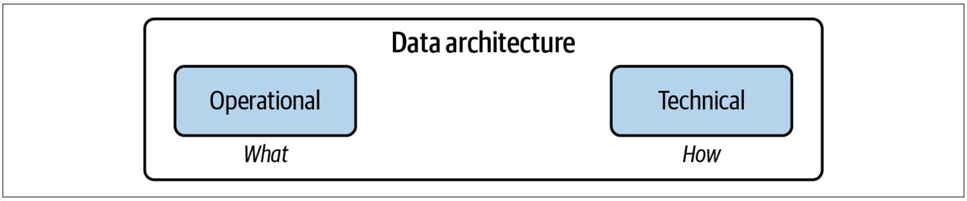


Figure 3-2. Operational and technical data architecture

Now that we have a working definition of data architecture, let's cover the elements of "good" data architecture.

"Good" Data Architecture

> Never shoot for the best architecture, but rather the least worst architecture.
>
> —Mark Richards and Neal Ford[7]

According to Grady Booch, "Architecture represents the significant design decisions that shape a system, where *significant* is measured by cost of change." Data architects aim to make significant decisions that will lead to good architecture at a basic level.

What do we mean by "good" data architecture? To paraphrase an old cliche, you know good when you see it. *Good data architecture* serves business requirements with a common, widely reusable set of building blocks while maintaining flexibility and making appropriate trade-offs. Bad architecture is authoritarian and tries to cram a bunch of one-size-fits-all decisions into a big ball of mud.

[7] Mark Richards and Neal Ford, *Fundamentals of Software Architecture* (Sebastopol, CA: O'Reilly, 2020), *https://oreil.ly/hpCp0*.

Agility is the foundation for good data architecture; it acknowledges that the world is fluid. *Good data architecture is flexible and easily maintainable*. It evolves in response to changes within the business and new technologies and practices that may unlock even more value in the future. Businesses and their use cases for data are always evolving. The world is dynamic, and the pace of change in the data space is accelerating. Last year's data architecture that served you well might not be sufficient for today, let alone next year.

Bad data architecture is tightly coupled, rigid, overly centralized, or uses the wrong tools for the job, hampering development and change management. Ideally, by designing architecture with reversibility in mind, changes will be less costly.

The undercurrents of the data engineering lifecycle form the foundation of good data architecture for companies at any stage of data maturity. Again, these undercurrents are security, data management, DataOps, data architecture, orchestration, and software engineering.

Good data architecture is a living, breathing thing. It's never finished. In fact, per our definition, change and evolution are central to the meaning and purpose of data architecture. Let's now look at the principles of good data architecture.

Principles of Good Data Architecture

This section takes a 10,000-foot view of good architecture by focusing on principles—key ideas useful in evaluating major architectural decisions and practices. We borrow inspiration for our architecture principles from several sources, especially the AWS Well-Architected Framework and Google Cloud's Five Principles for Cloud-Native Architecture.

The AWS Well-Architected Framework consists of six pillars:

- Operational excellence
- Security
- Reliability
- Performance efficiency
- Cost optimization
- Sustainability

Google Cloud's Five Principles for Cloud-Native Architecture are as follows:

- Design for automation.
- Be smart with state.
- Favor managed services.

- Practice defense in depth.
- Always be architecting.

We advise you to carefully study both frameworks, identify valuable ideas, and determine points of disagreement. We'd like to expand or elaborate on these pillars with these principles of data engineering architecture:

1. Choose common components wisely.
2. Plan for failure.
3. Architect for scalability.
4. Architecture is leadership.
5. Always be architecting.
6. Build loosely coupled systems.
7. Make reversible decisions.
8. Prioritize security.
9. Embrace FinOps.

Principle 1: Choose Common Components Wisely

One of the primary jobs of a data engineer is to choose common components and practices that can be used widely across an organization. When architects choose well and lead effectively, common components become a fabric facilitating team collaboration and breaking down silos. Common components enable agility within and across teams in conjunction with shared knowledge and skills.

Common components can be anything that has broad applicability within an organization. Common components include object storage, version-control systems, observability, monitoring and orchestration systems, and processing engines. Common components should be accessible to everyone with an appropriate use case, and teams are encouraged to rely on common components already in use rather than reinventing the wheel. Common components must support robust permissions and security to enable sharing of assets among teams while preventing unauthorized access.

Cloud platforms are an ideal place to adopt common components. For example, compute and storage separation in cloud data systems allows users to access a shared storage layer (most commonly object storage) using specialized tools to access and query the data needed for specific use cases.

Choosing common components is a balancing act. On the one hand, you need to focus on needs across the data engineering lifecycle and teams, utilize common components that will be useful for individual projects, and simultaneously facilitate interoperation and collaboration. On the other hand, architects should avoid decisions that will hamper the productivity of engineers working on domain-specific problems by forcing them into one-size-fits-all technology solutions. Chapter 4 provides additional details.

Principle 2: Plan for Failure

> Everything fails, all the time.
>
> —Werner Vogels, CTO of Amazon Web Services[8]

Modern hardware is highly robust and durable. Even so, any hardware component will fail, given enough time. To build highly robust data systems, you must consider failures in your designs. Here are a few key terms for evaluating failure scenarios; we describe these in greater detail in this chapter and throughout the book:

Availability
: The percentage of time an IT service or component is in an operable state.

Reliability
: The system's probability of meeting defined standards in performing its intended function during a specified interval.

Recovery time objective
: The maximum acceptable time for a service or system outage. The recovery time objective (RTO) is generally set by determining the business impact of an outage. An RTO of one day might be fine for an internal reporting system. A website outage of just five minutes could have a significant adverse business impact on an online retailer.

Recovery point objective
: The acceptable state after recovery. In data systems, data is often lost during an outage. In this setting, the recovery point objective (RPO) refers to the maximum acceptable data loss.

Engineers need to consider acceptable reliability, availability, RTO, and RPO in designing for failure. These will guide their architecture decisions as they assess possible failure scenarios.

[8] UberPulse, "Amazon.com CTO: Everything Fails," YouTube video, 3:03, *https://oreil.ly/vDVlX*.

Principle 3: Architect for Scalability

Scalability in data systems encompasses two main capabilities. First, scalable systems can *scale up* to handle significant quantities of data. We might need to spin up a large cluster to train a model on a petabyte of customer data or scale out a streaming ingestion system to handle a transient load spike. Our ability to scale up allows us to handle extreme loads temporarily. Second, scalable systems can *scale down*. Once the load spike ebbs, we should automatically remove capacity to cut costs. (This is related to principle 9.) An *elastic system* can scale dynamically in response to load, ideally in an automated fashion.

Some scalable systems can also *scale to zero*: they shut down completely when not in use. Once the large model-training job completes, we can delete the cluster. Many serverless systems (e.g., serverless functions and serverless online analytical processing, or OLAP, databases) can automatically scale to zero.

Note that deploying inappropriate scaling strategies can result in overcomplicated systems and high costs. A straightforward relational database with one failover node may be appropriate for an application instead of a complex cluster arrangement. Measure your current load, approximate load spikes, and estimate load over the next several years to determine if your database architecture is appropriate. If your startup grows much faster than anticipated, this growth should also lead to more available resources to rearchitect for scalability.

Principle 4: Architecture Is Leadership

Data architects are responsible for technology decisions and architecture descriptions and disseminating these choices through effective leadership and training. Data architects should be highly technically competent but delegate most individual contributor work to others. Strong leadership skills combined with high technical competence are rare and extremely valuable. The best data architects take this duality seriously.

Note that leadership does not imply a command-and-control approach to technology. It was not uncommon in the past for architects to choose one proprietary database technology and force every team to house their data there. We oppose this approach because it can significantly hinder current data projects. Cloud environments allow architects to balance common component choices with flexibility that enables innovation within projects.

Returning to the notion of technical leadership, Martin Fowler describes a specific archetype of an ideal software architect, well embodied in his colleague Dave Rice:[9]

> In many ways, the most important activity of *Architectus Oryzus* is to mentor the development team, to raise their level so they can take on more complex issues. Improving the development team's ability gives an architect much greater leverage than being the sole decision-maker and thus running the risk of being an architectural bottleneck.

An ideal data architect manifests similar characteristics. They possess the technical skills of a data engineer but no longer practice data engineering day to day; they mentor current data engineers, make careful technology choices in consultation with their organization, and disseminate expertise through training and leadership. They train engineers in best practices and bring the company's engineering resources together to pursue common goals in both technology and business.

As a data engineer, you should practice architecture leadership and seek mentorship from architects. Eventually, you may well occupy the architect role yourself.

Principle 5: Always Be Architecting

We borrow this principle directly from Google Cloud's Five Principles for Cloud-Native Architecture. Data architects don't serve in their role simply to maintain the existing state; instead, they constantly design new and exciting things in response to changes in business and technology. Per the EABOK, an architect's job is to develop deep knowledge of the *baseline architecture* (current state), develop a *target architecture*, and map out a *sequencing plan* to determine priorities and the order of architecture changes.

We add that modern architecture should not be command-and-control or waterfall but collaborative and agile. The data architect maintains a target architecture and sequencing plans that change over time. The target architecture becomes a moving target, adjusted in response to business and technology changes internally and worldwide. The sequencing plan determines immediate priorities for delivery.

Principle 6: Build Loosely Coupled Systems

> When the architecture of the system is designed to enable teams to test, deploy, and change systems without dependencies on other teams, teams require little communication to get work done. In other words, both the architecture and the teams are loosely coupled.
>
> —Google DevOps tech architecture guide[10]

[9] Martin Fowler, "Who Needs an Architect?" *IEEE Software*, July/August 2003, *https://oreil.ly/wAMmZ*.

[10] Google Cloud, "DevOps Tech: Architecture," Cloud Architecture Center, *https://oreil.ly/j4MT1*.

In 2002, Bezos wrote an email to Amazon employees that became known as the Bezos API Mandate:[11]

1. All teams will henceforth expose their data and functionality through service interfaces.
2. Teams must communicate with each other through these interfaces.
3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols—doesn't matter.
5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

The advent of Bezos's API Mandate is widely viewed as a watershed moment for Amazon. Putting data and services behind APIs enabled the loose coupling and eventually resulted in AWS as we know it now. Google's pursuit of loose coupling allowed it to grow its systems to an extraordinary scale.

For software architecture, a loosely coupled system has the following properties:

1. Systems are broken into many small components.
2. These systems interface with other services through abstraction layers, such as a messaging bus or an API. These abstraction layers hide and protect internal details of the service, such as a database backend or internal classes and method calls.
3. As a consequence of property 2, internal changes to a system component don't require changes in other parts. Details of code updates are hidden behind stable APIs. Each piece can evolve and improve separately.
4. As a consequence of property 3, there is no waterfall, global release cycle for the whole system. Instead, each component is updated separately as changes and improvements are made.

Notice that we are talking about *technical systems*. We need to think bigger. Let's translate these technical characteristics into organizational characteristics:

[11] "The Bezos API Mandate: Amazon's Manifesto for Externalization," Nordic APIs, January 19, 2021, *https://oreil.ly/vIs8m*.

1. Many small teams engineer a large, complex system. Each team is tasked with engineering, maintaining, and improving some system components.
2. These teams publish the abstract details of their components to other teams via API definitions, message schemas, etc. Teams need not concern themselves with other teams' components; they simply use the published API or message specifications to call these components. They iterate their part to improve their performance and capabilities over time. They might also publish new capabilities as they are added or request new stuff from other teams. Again, the latter happens without teams needing to worry about the internal technical details of the requested features. Teams work together through *loosely coupled communication*.
3. As a consequence of characteristic 2, each team can rapidly evolve and improve its component independently of the work of other teams.
4. Specifically, characteristic 3 implies that teams can release updates to their components with minimal downtime. Teams release continuously during regular working hours to make code changes and test them.

Loose coupling of both technology and human systems will allow your data engineering teams to more efficiently collaborate with one another and with other parts of the company. This principle also directly facilitates principle 7.

Principle 7: Make Reversible Decisions

The data landscape is changing rapidly. Today's hot technology or stack is tomorrow's afterthought. Popular opinion shifts quickly. You should aim for reversible decisions, as these tend to simplify your architecture and keep it agile.

As Fowler wrote, "One of an architect's most important tasks is to remove architecture by finding ways to eliminate irreversibility in software designs."[12] What was true when Fowler wrote this in 2003 is just as accurate today.

As we said previously, Bezos refers to reversible decisions as "two-way doors." As he says, "If you walk through and don't like what you see on the other side, you can't get back to before. We can call these Type 1 decisions. But most decisions aren't like that—they are changeable, reversible—they're two-way doors." Aim for two-way doors whenever possible.

Given the pace of change—and the decoupling/modularization of technologies across your data architecture—always strive to pick the best-of-breed solutions that work for today. Also, be prepared to upgrade or adopt better practices as the landscape evolves.

[12] Fowler, "Who Needs an Architect?"

Principle 8: Prioritize Security

Every data engineer must assume responsibility for the security of the systems they build and maintain. We focus now on two main ideas: zero-trust security and the shared responsibility security model. These align closely to a cloud-native architecture.

Hardened-perimeter and zero-trust security models

To define *zero-trust security*, it's helpful to start by understanding the traditional hard-perimeter security model and its limitations, as detailed in Google Cloud's Five Principles:[13]

> Traditional architectures place a lot of faith in perimeter security, crudely a hardened network perimeter with "trusted things" inside and "untrusted things" outside. Unfortunately, this approach has always been vulnerable to insider attacks, as well as external threats such as spear phishing.

The 1996 film *Mission Impossible* presents a perfect example of the hard-perimeter security model and its limitations. In the movie, the CIA hosts highly sensitive data on a storage system inside a room with extremely tight physical security. Ethan Hunt infiltrates CIA headquarters and exploits a human target to gain physical access to the storage system. Once inside the secure room, he can exfiltrate data with relative ease.

For at least a decade, alarming media reports have made us aware of the growing menace of security breaches that exploit human targets inside hardened organizational security perimeters. Even as employees work on highly secure corporate networks, they remain connected to the outside world through email and mobile devices. External threats effectively become internal threats.

In a cloud-native environment, the notion of a hardened perimeter erodes further. All assets are connected to the outside world to some degree. While virtual private cloud (VPC) networks can be defined with no external connectivity, the API control plane that engineers use to define these networks still faces the internet.

The shared responsibility model

Amazon emphasizes the shared responsibility model, which divides security into the security *of* the cloud and security *in* the cloud. AWS is responsible for the security of the cloud:[14]

[13] Tom Grey, "5 Principles for Cloud-Native Architecture—What It Is and How to Master It," Google Cloud blog, June 19, 2019, *https://oreil.ly/4NkGf*.

[14] Amazon Web Services, "Security in AWS WAF," AWS WAF documentation, *https://oreil.ly/rEFoU*.

> AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely.

AWS users are responsible for security in the cloud:

> Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

In general, all cloud providers operate on some form of this shared responsibility model. They secure their services according to published specifications. Still, it is ultimately the user's responsibility to design a security model for their applications and data and leverage cloud capabilities to realize this model.

Data engineers as security engineers

In the corporate world today, a command-and-control approach to security is quite common, wherein security and networking teams manage perimeters and general security practices. The cloud pushes this responsibility out to engineers who are not explicitly in security roles. Because of this responsibility, in conjunction with more general erosion of the hard security perimeter, all data engineers should consider themselves security engineers.

Failure to assume these new implicit responsibilities can lead to dire consequences. Numerous data breaches have resulted from the simple error of configuring Amazon S3 buckets with public access.[15] Those who handle data must assume that they are ultimately responsible for securing it.

Principle 9: Embrace FinOps

Let's start by considering a couple of definitions of FinOps. First, the FinOps Foundation offers this:[16]

> FinOps is an evolving cloud financial management discipline and cultural practice that enables organizations to get maximum business value by helping engineering, finance, technology, and business teams to collaborate on data-driven spending decisions.

In addition, J. R. Sorment and Mike Fuller provide the following definition in *Cloud FinOps*:[17]

> The term "FinOps" typically refers to the emerging professional movement that advocates a collaborative working relationship between DevOps and Finance, resulting in

[15] Ericka Chickowski, "Leaky Buckets: 10 Worst Amazon S3 Breaches," Bitdefender *Business Insights* blog, Jan 24, 2018, *https://oreil.ly/pFEFO*.

[16] FinOps Foundation, "What Is FinOps," *https://oreil.ly/wJFVn*.

[17] J. R. Storment and Mike Fuller, *Cloud FinOps* (Sebastapol, CA: O'Reilly, 2019), *https://oreil.ly/QV6vF*.

> an iterative, data-driven management of infrastructure spending (i.e., lowering the unit economics of cloud) while simultaneously increasing the cost efficiency and, ultimately, the profitability of the cloud environment.

The cost structure of data has evolved dramatically during the cloud era. In an on-premises setting, data systems are generally acquired with a capital expenditure (described more in Chapter 4) for a new system every few years in an on-premises setting. Responsible parties have to balance their budget against desired compute and storage capacity. Overbuying entails wasted money, while underbuying means hampering future data projects and driving significant personnel time to control system load and data size; underbuying may require faster technology refresh cycles, with associated extra costs.

In the cloud era, most data systems are pay-as-you-go and readily scalable. Systems can run on a cost-per-query model, cost-per-processing-capacity model, or another variant of a pay-as-you-go model. This approach can be far more efficient than the capital expenditure approach. It is now possible to scale up for high performance, and then scale down to save money. However, the pay-as-you-go approach makes spending far more dynamic. The new challenge for data leaders is to manage budgets, priorities, and efficiency.

Cloud tooling necessitates a set of processes for managing spending and resources. In the past, data engineers thought in terms of performance engineering—maximizing the performance for data processes on a fixed set of resources and buying adequate resources for future needs. With FinOps, engineers need to learn to think about the cost structures of cloud systems. For example, what is the appropriate mix of AWS spot instances when running a distributed cluster? What is the most appropriate approach for running a sizable daily job in terms of cost-effectiveness and performance? When should the company switch from a pay-per-query model to reserved capacity?

FinOps evolves the operational monitoring model to monitor spending on an ongoing basis. Rather than simply monitor requests and CPU utilization for a web server, FinOps might monitor the ongoing cost of serverless functions handling traffic, as well as spikes in spending trigger alerts. Just as systems are designed to fail gracefully in excessive traffic, companies may consider adopting hard limits for spending, with graceful failure modes in response to spending spikes.

Ops teams should also think in terms of cost attacks. Just as a distributed denial-of-service (DDoS) attack can block access to a web server, many companies have discovered to their chagrin that excessive downloads from S3 buckets can drive spending through the roof and threaten a small startup with bankruptcy. When sharing data publicly, data teams can address these issues by setting requester-pays policies, or simply monitoring for excessive data access spending and quickly removing access if spending begins to rise to unacceptable levels.

As of this writing, FinOps is a recently formalized practice. The FinOps Foundation was started only in 2019.[18] However, we highly recommend you start thinking about FinOps early, before you encounter high cloud bills. Start your journey with the FinOps Foundation and O'Reilly's *Cloud FinOps*. We also suggest that data engineers involve themselves in the community process of creating FinOps practices for data engineering— in such a new practice area, a good deal of territory is yet to be mapped out.

Now that you have a high-level understanding of good data architecture principles, let's dive a bit deeper into the major concepts you'll need to design and build good data architecture.

Major Architecture Concepts

If you follow the current trends in data, it seems like new types of data tools and architectures are arriving on the scene every week. Amidst this flurry of activity, we must not lose sight of the main goal of all of these architectures: to take data and transform it into something useful for downstream consumption.

Domains and Services

> Domain: A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.
>
> —Eric Evans[19]

Before diving into the components of the architecture, let's briefly cover two terms you'll see come up very often: domain and services. A *domain* is the real-world subject area for which you're architecting. A *service* is a set of functionality whose goal is to accomplish a task. For example, you might have a sales order-processing service whose task is to process orders as they are created. The sales order-processing service's only job is to process orders; it doesn't provide other functionality, such as inventory management or updating user profiles.

A domain can contain multiple services. For example, you might have a sales domain with three services: orders, invoicing, and products. Each service has particular tasks that support the sales domain. Other domains may also share services (Figure 3-3). In this case, the accounting domain is responsible for basic accounting functions: invoicing, payroll, and accounts receivable (AR). Notice the accounting domain shares the invoice service with the sales domain since a sale generates an invoice,

[18] "FinOps Foundation Soars to 300 Members and Introduces New Partner Tiers for Cloud Service Providers and Vendors," Business Wire, June 17, 2019, *https://oreil.ly/XcwYO*.

[19] Eric Evans, *Domain-Driven Design Reference: Definitions and Pattern Summaries* (March 2015), *https://oreil.ly/pQ9oq*.

and accounting must keep track of invoices to ensure that payment is received. Sales and accounting own their respective domains.

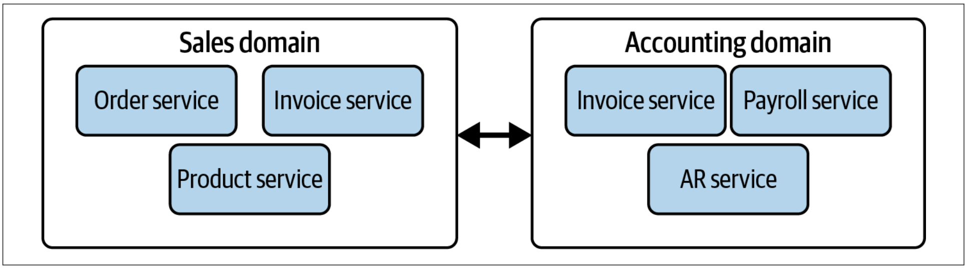


Figure 3-3. Two domains (sales and accounting) share a common service (invoices), and sales and accounting own their respective domains

When thinking about what constitutes a domain, focus on what the domain represents in the real world and work backward. In the preceding example, the sales domain should represent what happens with the sales function in your company. When architecting the sales domain, avoid cookie-cutter copying and pasting from what other companies do. Your company's sales function likely has unique aspects that require specific services to make it work the way your sales team expects.

Identify what should go in the domain. When determining what the domain should encompass and what services to include, the best advice is to simply go and talk with users and stakeholders, listen to what they're saying, and build the services that will help them do their job. Avoid the classic trap of architecting in a vacuum.

Distributed Systems, Scalability, and Designing for Failure

The discussion in this section is related to our second and third principles of data engineering architecture discussed previously: plan for failure and architect for scalability. As data engineers, we're interested in four closely related characteristics of data systems (availability and reliability were mentioned previously, but we reiterate them here for completeness):

Scalability
: Allows us to increase the capacity of a system to improve performance and handle the demand. For example, we might want to scale a system to handle a high rate of queries or process a huge data set.

Elasticity
: The ability of a scalable system to scale dynamically; a highly elastic system can automatically scale up and down based on the current workload. Scaling up is critical as demand increases, while scaling down saves money in a cloud environment. Modern systems sometimes scale to zero, meaning they can automatically shut down when idle.

Availability

The percentage of time an IT service or component is in an operable state.

Reliability

The system's probability of meeting defined standards in performing its intended function during a specified interval.

> See PagerDuty's "Why Are Availability and Reliability Crucial?" web page for definitions and background on availability and reliability.

How are these characteristics related? If a system fails to meet performance requirements during a specified interval, it may become unresponsive. Thus low reliability can lead to low availability. On the other hand, dynamic scaling helps ensure adequate performance without manual intervention from engineers—elasticity improves reliability.

Scalability can be realized in a variety of ways. For your services and domains, does a single machine handle everything? A single machine can be scaled vertically; you can increase resources (CPU, disk, memory, I/O). But there are hard limits to possible resources on a single machine. Also, what happens if this machine dies? Given enough time, some components will eventually fail. What's your plan for backup and failover? Single machines generally can't offer high availability and reliability.

We utilize a distributed system to realize higher overall scaling capacity and increased availability and reliability. *Horizontal scaling* allows you to add more machines to satisfy load and resource requirements (Figure 3-4). Common horizontally scaled systems have a leader node that acts as the main point of contact for the instantiation, progress, and completion of workloads. When a workload is started, the leader node distributes tasks to the worker nodes within its system, completing the tasks and returning the results to the leader node. Typical modern distributed architectures also build in redundancy. Data is replicated so that if a machine dies, the other machines can pick up where the missing server left off; the cluster may add more machines to restore capacity.

Distributed systems are widespread in the various data technologies you'll use across your architecture. Almost every cloud data warehouse object storage system you use has some notion of distribution under the hood. Management details of the distributed system are typically abstracted away, allowing you to focus on high-level architecture instead of low-level plumbing. However, we highly recommend that you learn more about distributed systems because these details can be extremely helpful in understanding and improving the performance of your pipelines; Martin Kleppmann's *Designing Data-Intensive Applications* (O'Reilly) is an excellent resource.

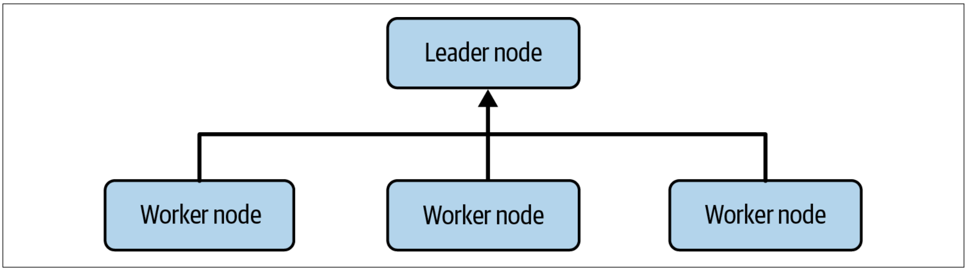


Figure 3-4. A simple horizontal distributed system utilizing a leader-follower architecture, with one leader node and three worker nodes

Tight Versus Loose Coupling: Tiers, Monoliths, and Microservices

When designing a data architecture, you choose how much interdependence you want to include within your various domains, services, and resources. On one end of the spectrum, you can choose to have extremely centralized dependencies and workflows. Every part of a domain and service is vitally dependent upon every other domain and service. This pattern is known as *tightly coupled*.

On the other end of the spectrum, you have decentralized domains and services that do not have strict dependence on each other, in a pattern known as *loose coupling*. In a loosely coupled scenario, it's easy for decentralized teams to build systems whose data may not be usable by their peers. Be sure to assign common standards, ownership, responsibility, and accountability to the teams owning their respective domains and services. Designing "good" data architecture relies on trade-offs between the tight and loose coupling of domains and services.

It's worth noting that many of the ideas in this section originate in software development. We'll try to retain the context of these big ideas' original intent and spirit—keeping them agnostic of data—while later explaining some differences you should be aware of when applying these concepts to data specifically.

Architecture tiers

As you develop your architecture, it helps to be aware of architecture tiers. Your architecture has layers—data, application, business logic, presentation, and so forth —and you need to know how to decouple these layers. Because tight coupling of modalities presents obvious vulnerabilities, keep in mind how you structure the layers of your architecture to achieve maximum reliability and flexibility. Let's look at single-tier and multitier architecture.

Single tier. In a *single-tier architecture*, your database and application are tightly coupled, residing on a single server (Figure 3-5). This server could be your laptop or a single virtual machine (VM) in the cloud. The tightly coupled nature means if the server, the database, or the application fails, the entire architecture fails. While single-tier architectures are good for prototyping and development, they are not advised for production environments because of the obvious failure risks.

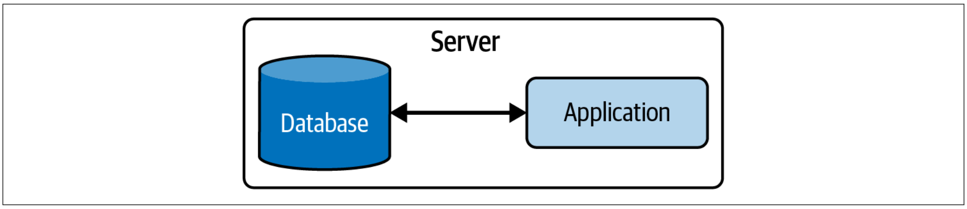


Figure 3-5. Single-tier architecture

Even when single-tier architectures build in redundancy (for example, a failover replica), they present significant limitations in other ways. For instance, it is often impractical (and not advisable) to run analytics queries against production application databases. Doing so risks overwhelming the database and causing the application to become unavailable. A single-tier architecture is fine for testing systems on a local machine but is not advised for production uses.

Multitier. The challenges of a tightly coupled single-tier architecture are solved by decoupling the data and application. A *multitier* (also known as *n-tier*) architecture is composed of separate layers: data, application, business logic, presentation, etc. These layers are bottom-up and hierarchical, meaning the lower layer isn't necessarily dependent on the upper layers; the upper layers depend on the lower layers. The notion is to separate data from the application, and application from the presentation.

A common multitier architecture is a three-tier architecture, a widely used client-server design. A *three-tier architecture* consists of data, application logic, and presentation tiers (Figure 3-6). Each tier is isolated from the other, allowing for separation of concerns. With a three-tier architecture, you're free to use whatever technologies you prefer within each tier without the need to be monolithically focused.

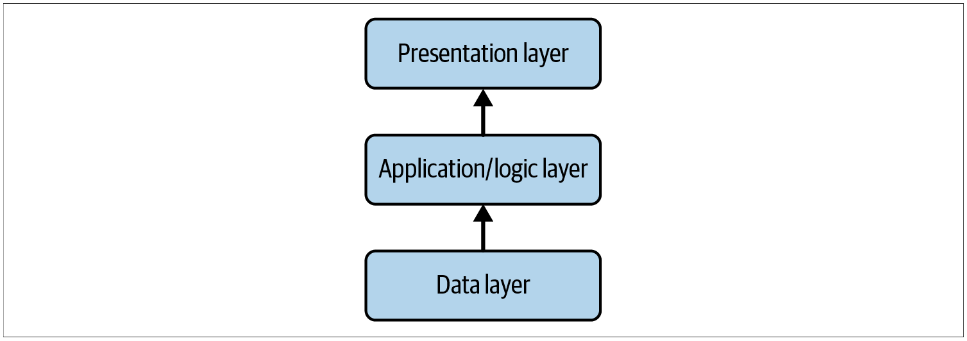


Figure 3-6. A three-tier architecture

We've seen many single-tier architectures in production. Single-tier architectures offer simplicity but also severe limitations. Eventually, an organization or application outgrows this arrangement; it works well until it doesn't. For instance, in a single-tier architecture, the data and logic layers share and compete for resources (disk, CPU, and memory) in ways that are simply avoided in a multitier architecture. Resources are spread across various tiers. Data engineers should use tiers to evaluate their layered architecture and the way dependencies are handled. Again, start simple and bake in evolution to additional tiers as your architecture becomes more complex.

In a multitier architecture, you need to consider separating your layers and the way resources are shared within layers when working with a distributed system. Distributed systems under the hood power many technologies you'll encounter across the data engineering lifecycle. First, think about whether you want resource contention with your nodes. If not, exercise a *shared-nothing architecture*: a single node handles each request, meaning other nodes do not share resources such as memory, disk, or CPU with this node or with each other. Data and resources are isolated to the node. Alternatively, various nodes can handle multiple requests and share resources but at the risk of resource contention. Another consideration is whether nodes should share the same disk and memory accessible by all nodes. This is called a *shared disk architecture* and is common when you want shared resources if a random node failure occurs.

Monoliths

The general notion of a monolith includes as much as possible under one roof; in its most extreme version, a monolith consists of a single codebase running on a single machine that provides both the application logic and user interface.

Coupling within monoliths can be viewed in two ways: technical coupling and domain coupling. *Technical coupling* refers to architectural tiers, while *domain coupling* refers to the way domains are coupled together. A monolith has varying degrees of coupling among technologies and domains. You could have an application with various layers decoupled in a multitier architecture but still share multiple domains. Or, you could have a single-tier architecture serving a single domain.

The tight coupling of a monolith implies a lack of modularity of its components. Swapping out or upgrading components in a monolith is often an exercise in trading one pain for another. Because of the tightly coupled nature, reusing components across the architecture is difficult or impossible. When evaluating how to improve a monolithic architecture, it's often a game of whack-a-mole: one component is improved, often at the expense of unknown consequences with other areas of the monolith.

Data teams will often ignore solving the growing complexity of their monolith, letting it devolve into a big ball of mud.

Chapter 4 provides a more extensive discussion comparing monoliths to distributed technologies. We also discuss the *distributed monolith*, a strange hybrid that emerges when engineers build distributed systems with excessive tight coupling.

Microservices

Compared with the attributes of a monolith—interwoven services, centralization, and tight coupling among services—microservices are the polar opposite. *Microservices architecture* comprises separate, decentralized, and loosely coupled services. Each service has a specific function and is decoupled from other services operating within its domain. If one service temporarily goes down, it won't affect the ability of other services to continue functioning.

A question that comes up often is how to convert your monolith into many microservices (Figure 3-7). This completely depends on how complex your monolith is and how much effort it will be to start extracting services out of it. It's entirely possible that your monolith cannot be broken apart, in which case, you'll want to start creating a new parallel architecture that has the services decoupled in a microservices-friendly manner. We don't suggest an entire refactor but instead break out services. The monolith didn't arrive overnight and is a technology issue as an organizational one. Be sure you get buy-in from stakeholders of the monolith if you plan to break it apart.

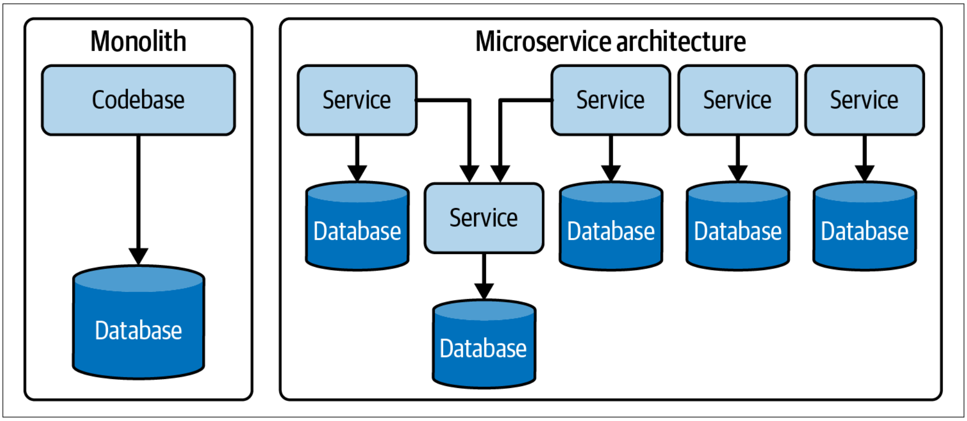


Figure 3-7. An extremely monolithic architecture runs all functionality inside a single codebase, potentially colocating a database on the same host server

If you'd like to learn more about breaking apart a monolith, we suggest reading the fantastic, pragmatic guide *Software Architecture: The Hard Parts* by Neal Ford et al. (O'Reilly).

Considerations for data architecture

As we mentioned at the start of this section, the concepts of tight versus loose coupling stem from software development, with some of these concepts dating back over 20 years. Though architectural practices in data are now adopting those from software development, it's still common to see very monolithic, tightly coupled data architectures. Some of this is due to the nature of existing data technologies and the way they integrate.

For example, data pipelines might consume data from many sources ingested into a central data warehouse. The central data warehouse is inherently monolithic. A move toward a microservices equivalent with a data warehouse is to decouple the workflow with domain-specific data pipelines connecting to corresponding domain-specific data warehouses. For example, the sales data pipeline connects to the sales-specific data warehouse, and the inventory and product domains follow a similar pattern.

Rather than dogmatically preach microservices over monoliths (among other arguments), we suggest you pragmatically use loose coupling as an ideal, while recognizing the state and limitations of the data technologies you're using within your data architecture. Incorporate reversible technology choices that allow for modularity and loose coupling whenever possible.

As you can see in Figure 3-7, you separate the components of your architecture into different layers of concern in a vertical fashion. While a multitier architecture solves the technical challenges of decoupling shared resources, it does not address the complexity of sharing domains. Along the lines of single versus multitiered architecture, you should also consider how you separate the domains of your data architecture. For example, your analyst team might rely on data from sales and inventory. The sales and inventory domains are different and should be viewed as separate.

One approach to this problem is centralization: a single team is responsible for gathering data from all domains and reconciling it for consumption across the organization. (This is a common approach in traditional data warehousing.) Another approach is the *data mesh*. With the data mesh, each software team is responsible for preparing its data for consumption across the rest of the organization. We'll say more about the data mesh later in this chapter.

Our advice: monoliths aren't necessarily bad, and it might make sense to start with one under certain conditions. Sometimes you need to move fast, and it's much simpler to start with a monolith. Just be prepared to break it into smaller pieces eventually; don't get too comfortable.

User Access: Single Versus Multitenant

As a data engineer, you have to make decisions about sharing systems across multiple teams, organizations, and customers. In some sense, all cloud services are multitenant, although this multitenancy occurs at various grains. For example, a cloud compute instance is usually on a shared server, but the VM itself provides some degree of isolation. Object storage is a multitenant system, but cloud vendors guarantee security and isolation so long as customers configure their permissions correctly.

Engineers frequently need to make decisions about multitenancy at a much smaller scale. For example, do multiple departments in a large company share the same data warehouse? Does the organization share data for multiple large customers within the same table?

We have two factors to consider in multitenancy: performance and security. With multiple large tenants within a cloud system, will the system support consistent performance for all tenants, or will there be a noisy neighbor problem? (That is, will high usage from one tenant degrade performance for other tenants?) Regarding security, data from different tenants must be properly isolated. When a company has multiple external customer tenants, these tenants should not be aware of one another, and engineers must prevent data leakage. Strategies for data isolation vary by system. For instance, it is often perfectly acceptable to use multitenant tables and isolate data through views. However, you must make certain that these views cannot leak data. Read vendor or project documentation to understand appropriate strategies and risks.

Event-Driven Architecture

Your business is rarely static. Things often happen in your business, such as getting a new customer, a new order from a customer, or an order for a product or service. These are all examples of *events* that are broadly defined as something that happened, typically a change in the *state* of something. For example, a new order might be created by a customer, or a customer might later make an update to this order.

An event-driven workflow (Figure 3-8) encompasses the ability to create, update, and asynchronously move events across various parts of the data engineering lifecycle. This workflow boils down to three main areas: event production, routing, and consumption. An event must be produced and routed to something that consumes it without tightly coupled dependencies among the producer, event router, and consumer.

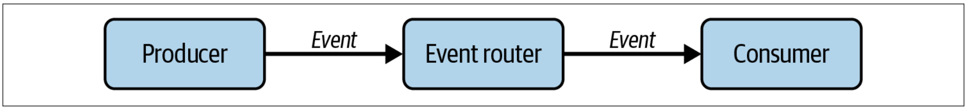


Figure 3-8. In an event-driven workflow, an event is produced, routed, and then consumed

An event-driven architecture (Figure 3-9) embraces the event-driven workflow and uses this to communicate across various services. The advantage of an event-driven architecture is that it distributes the state of an event across multiple services. This is helpful if a service goes offline, a node fails in a distributed system, or you'd like multiple consumers or services to access the same events. Anytime you have loosely coupled services, this is a candidate for event-driven architecture. Many of the examples we describe later in this chapter incorporate some form of event-driven architecture.

You'll learn more about event-driven streaming and messaging systems in Chapter 5.

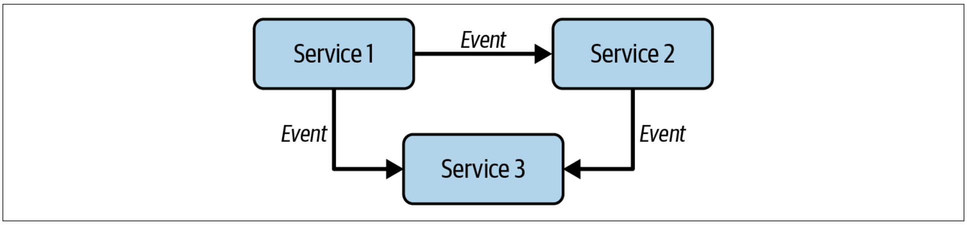


Figure 3-9. In an event-driven architecture, events are passed between loosely coupled services

Brownfield Versus Greenfield Projects

Before you design your data architecture project, you need to know whether you're starting with a clean slate or redesigning an existing architecture. Each type of project requires assessing trade-offs, albeit with different considerations and approaches. Projects roughly fall into two buckets: brownfield and greenfield.

Brownfield projects

Brownfield projects often involve refactoring and reorganizing an existing architecture and are constrained by the choices of the present and past. Because a key part of architecture is change management, you must figure out a way around these limitations and design a path forward to achieve your new business and technical objectives. Brownfield projects require a thorough understanding of the legacy architecture and the interplay of various old and new technologies. All too often, it's easy to criticize a prior team's work and decisions, but it is far better to dig deep, ask questions, and understand why decisions were made. Empathy and context go a long way in helping you diagnose problems with the existing architecture, identify opportunities, and recognize pitfalls.

You'll need to introduce your new architecture and technologies and deprecate the old stuff at some point. Let's look at a couple of popular approaches. Many teams jump headfirst into an all-at-once or big-bang overhaul of the old architecture, often figuring out deprecation as they go. Though popular, we don't advise this approach because of the associated risks and lack of a plan. This path often leads to disaster, with many irreversible and costly decisions. Your job is to make reversible, high-ROI decisions.

A popular alternative to a direct rewrite is the strangler pattern: new systems slowly and incrementally replace a legacy architecture's components.[20] Eventually, the legacy architecture is completely replaced. The attraction to the strangler pattern is its targeted and surgical approach of deprecating one piece of a system at a time. This allows for flexible and reversible decisions while assessing the impact of the deprecation on dependent systems.

It's important to note that deprecation might be "ivory tower" advice and not practical or achievable. Eradicating legacy technology or architecture might be impossible if you're at a large organization. Someone, somewhere, is using these legacy components. As someone once said, "Legacy is a condescending way to describe something that makes money."

If you can deprecate, understand there are numerous ways to deprecate your old architecture. It is critical to demonstrate value on the new platform by gradually increasing its maturity to show evidence of success and then follow an exit plan to shut down old systems.

Greenfield projects

On the opposite end of the spectrum, a *greenfield project* allows you to pioneer a fresh start, unconstrained by the history or legacy of a prior architecture. Greenfield projects tend to be easier than brownfield projects, and many data architects and engineers find them more fun! You have the opportunity to try the newest and coolest tools and architectural patterns. What could be more exciting?

You should watch out for some things before getting too carried away. We see teams get overly exuberant with shiny object syndrome. They feel compelled to reach for the latest and greatest technology fad without understanding how it will impact the value of the project. There's also a temptation to do *resume-driven development*, stacking up impressive new technologies without prioritizing the project's ultimate goals.[21] Always prioritize requirements over building something cool.

Whether you're working on a brownfield or greenfield project, always focus on the tenets of "good" data architecture. Assess trade-offs, make flexible and reversible decisions, and strive for positive ROI.

[20] Martin Fowler, "StranglerFigApplication," June 29, 2004, *https://oreil.ly/PmqxB*.

[21] Mike Loukides, "Resume Driven Development," *O'Reilly Radar*, October 13, 2004, *https://oreil.ly/BUHa8*.

Now, we'll look at examples and types of architectures—some established for decades (the data warehouse), some brand-new (the data lakehouse), and some that quickly came and went but still influence current architecture patterns (Lambda architecture).

Examples and Types of Data Architecture

Because data architecture is an abstract discipline, it helps to reason by example. In this section, we outline prominent examples and types of data architecture that are popular today. Though this set of examples is by no means exhaustive, the intention is to expose you to some of the most common data architecture patterns and to get you thinking about the requisite flexibility and trade-off analysis needed when designing a good architecture for your use case.

Data Warehouse

A *data warehouse* is a central data hub used for reporting and analysis. Data in a data warehouse is typically highly formatted and structured for analytics use cases. It's among the oldest and most well-established data architectures.

In 1989, Bill Inmon originated the notion of the data warehouse, which he described as "a subject-oriented, integrated, nonvolatile, and time-variant collection of data in support of management's decisions."[22] Though technical aspects of the data warehouse have evolved significantly, we feel this original definition still holds its weight today.

In the past, data warehouses were widely used at enterprises with significant budgets (often in the millions of dollars) to acquire data systems and pay internal teams to provide ongoing support to maintain the data warehouse. This was expensive and labor-intensive. Since then, the scalable, pay-as-you-go model has made cloud data warehouses accessible even to tiny companies. Because a third-party provider manages the data warehouse infrastructure, companies can do a lot more with fewer people, even as the complexity of their data grows.

It's worth noting two types of data warehouse architecture: organizational and technical. The *organizational data warehouse architecture* organizes data associated with certain business team structures and processes. The *technical data warehouse architecture* reflects the technical nature of the data warehouse, such as MPP. A company can have a data warehouse without an MPP system or run an MPP system that is not organized as a data warehouse. However, the technical and organizational architectures have existed in a virtuous cycle and are frequently identified with each other.

[22] H. W. Inmon, *Building the Data Warehouse* (Hoboken: Wiley, 2005).

The organizational data warehouse architecture has two main characteristics:

Separates online analytical processing (OLAP) from production databases (online transaction processing)

This separation is critical as businesses grow. Moving data into a separate physical system directs load away from production systems and improves analytics performance.

Centralizes and organizes data

Traditionally, a data warehouse pulls data from application systems by using ETL. The extract phase pulls data from source systems. The transformation phase cleans and standardizes data, organizing and imposing business logic in a highly modeled form. (Chapter 8 covers transformations and data models.) The load phase pushes data into the data warehouse target database system. Data is loaded into multiple data marts that serve the analytical needs for specific lines or business and departments. Figure 3-10 shows the general workflow. The data warehouse and ETL go hand in hand with specific business structures, including DBA and ETL developer teams that implement the direction of business leaders to ensure that data for reporting and analytics corresponds to business processes.

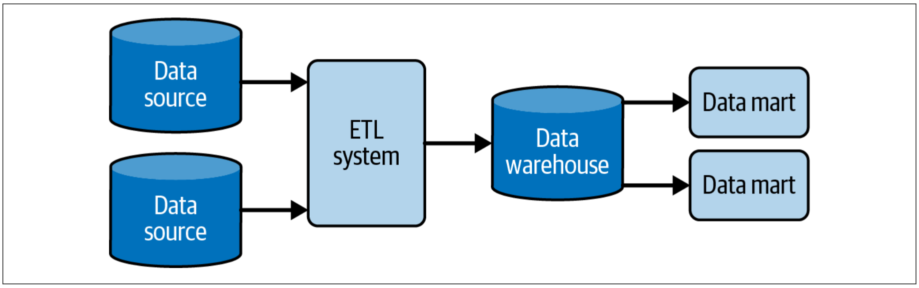


Figure 3-10. Basic data warehouse with ETL

Regarding the technical data warehouse architecture, the first MPP systems in the late 1970s became popular in the 1980s. MPPs support essentially the same SQL semantics used in relational application databases. Still, they are optimized to scan massive amounts of data in parallel and thus allow high-performance aggregation and statistical calculations. In recent years, MPP systems have increasingly shifted from a row-based to a columnar architecture to facilitate even larger data and queries, especially in cloud data warehouses. MPPs are indispensable for running performant queries for large enterprises as data and reporting needs grow.

One variation on ETL is ELT. With the ELT data warehouse architecture, data gets moved more or less directly from production systems into a staging area in the data warehouse. Staging in this setting indicates that the data is in a raw form. Rather than using an external system, transformations are handled directly in the data warehouse. The intention is to take advantage of the massive computational power of cloud data warehouses and data processing tools. Data is processed in batches, and transformed output is written into tables and views for analytics. Figure 3-11 shows the general process. ELT is also popular in a streaming arrangement, as events are streamed from a CDC process, stored in a staging area, and then subsequently transformed within the data warehouse.

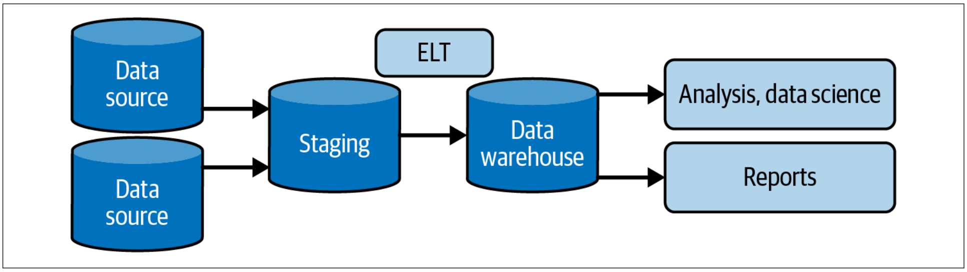


Figure 3-11. ELT—extract, load, and transform

A second version of ELT was popularized during big data growth in the Hadoop ecosystem. This is *transform-on-read ELT*, which we discuss in "Data Lake" on page 101.

The cloud data warehouse

Cloud data warehouses represent a significant evolution of the on-premises data warehouse architecture and have thus led to significant changes to the organizational architecture. Amazon Redshift kicked off the cloud data warehouse revolution. Instead of needing to appropriately size an MPP system for the next several years and sign a multimillion-dollar contract to procure the system, companies had the option of spinning up a Redshift cluster on demand, scaling it up over time as data and analytics demand grew. They could even spin up new Redshift clusters on demand to serve specific workloads and quickly delete clusters when they were no longer needed.

Google BigQuery, Snowflake, and other competitors popularized the idea of separating compute from storage. In this architecture, data is housed in object storage, allowing virtually limitless storage. This also gives users the option to spin up computing power on demand, providing ad hoc big data capabilities without the long-term cost of thousands of nodes.

Cloud data warehouses expand the capabilities of MPP systems to cover many big data use cases that required a Hadoop cluster in the very recent past. They can readily process petabytes of data in a single query. They typically support data structures that allow the storage of tens of megabytes of raw text data per row or extremely rich and complex JSON documents. As cloud data warehouses (and data lakes) mature, the line between the data warehouse and the data lake will continue to blur.

So significant is the impact of the new capabilities offered by cloud data warehouses that we might consider jettisoning the term *data warehouse* altogether. Instead, these services are evolving into a new data platform with much broader capabilities than those offered by a traditional MPP system.

Data marts

A *data mart* is a more refined subset of a warehouse designed to serve analytics and reporting, focused on a single suborganization, department, or line of business; every department has its own data mart, specific to its needs. This is in contrast to the full data warehouse that serves the broader organization or business.

Data marts exist for two reasons. First, a data mart makes data more easily accessible to analysts and report developers. Second, data marts provide an additional stage of transformation beyond that provided by the initial ETL or ELT pipelines. This can significantly improve performance if reports or analytics queries require complex joins and aggregations of data, especially when the raw data is large. Transform processes can populate the data mart with joined and aggregated data to improve performance for live queries. Figure 3-12 shows the general workflow. We discuss data marts, and modeling data for data marts, in Chapter 8.

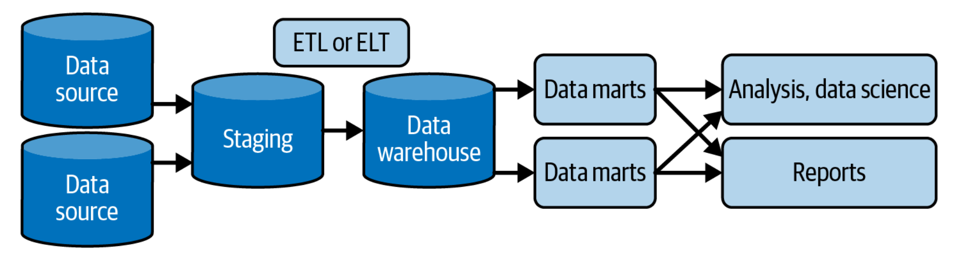


Figure 3-12. ETL or ELT plus data marts

Data Lake

Among the most popular architectures that appeared during the big data era is the *data lake*. Instead of imposing tight structural limitations on data, why not simply dump all of your data—structured and unstructured—into a central location? The data lake promised to be a democratizing force, liberating the business to drink from a fountain of limitless data. The first-generation data lake, "data lake 1.0," made solid contributions but generally failed to deliver on its promise.

Data lake 1.0 started with HDFS. As the cloud grew in popularity, these data lakes moved to cloud-based object storage, with extremely cheap storage costs and virtually limitless storage capacity. Instead of relying on a monolithic data warehouse where storage and compute are tightly coupled, the data lake allows an immense amount of data of any size and type to be stored. When this data needs to be queried or transformed, you have access to nearly unlimited computing power by spinning up a cluster on demand, and you can pick your favorite data-processing technology for the task at hand—MapReduce, Spark, Ray, Presto, Hive, etc.

Despite the promise and hype, data lake 1.0 had serious shortcomings. The data lake became a dumping ground; terms such as *data swamp*, *dark data*, and *WORN* were coined as once-promising data projects failed. Data grew to unmanageable sizes, with little in the way of schema management, data cataloging, and discovery tools. In addition, the original data lake concept was essentially write-only, creating huge headaches with the arrival of regulations such as GDPR that required targeted deletion of user records.

Processing data was also challenging. Relatively banal data transformations such as joins were a huge headache to code as MapReduce jobs. Later frameworks such as Pig and Hive somewhat improved the situation for data processing but did little to address the basic problems of data management. Simple data manipulation language (DML) operations common in SQL—deleting or updating rows—were painful to implement, generally achieved by creating entirely new tables. While big data engineers radiated a particular disdain for their counterparts in data warehousing, the latter could point out that data warehouses provided basic data management capabilities out of the box, and that SQL was an efficient tool for writing complex, performant queries and transformations.

Data lake 1.0 also failed to deliver on another core promise of the big data movement. Open source software in the Apache ecosystem was touted as a means to avoid multimillion-dollar contracts for proprietary MPP systems. Cheap, off-the-shelf hardware would replace custom vendor solutions. In reality, big data costs ballooned as the complexities of managing Hadoop clusters forced companies to hire large teams of engineers at high salaries. Companies often chose to purchase licensed, customized versions of Hadoop from vendors to avoid the exposed wires and sharp edges of the raw Apache codebase and acquire a set of scaffolding tools to make Hadoop more user-friendly. Even companies that avoided managing Hadoop clusters using cloud storage had to spend big on talent to write MapReduce jobs.

We should be careful not to understate the utility and power of first-generation data lakes. Many organizations found significant value in data lakes—especially huge, heavily data-focused Silicon Valley tech companies like Netflix and Facebook. These companies had the resources to build successful data practices and create their custom Hadoop-based tools and enhancements. But for many organizations, data lakes turned into an internal superfund site of waste, disappointment, and spiraling costs.

Convergence, Next-Generation Data Lakes, and the Data Platform

In response to the limitations of first-generation data lakes, various players have sought to enhance the concept to fully realize its promise. For example, Databricks introduced the notion of a *data lakehouse*. The lakehouse incorporates the controls, data management, and data structures found in a data warehouse while still housing data in object storage and supporting a variety of query and transformation engines. In particular, the data lakehouse supports atomicity, consistency, isolation, and durability (ACID) transactions, a big departure from the original data lake, where you simply pour in data and never update or delete it. The term *data lakehouse* suggests a convergence between data lakes and data warehouses.

The technical architecture of cloud data warehouses has evolved to be very similar to a data lake architecture. Cloud data warehouses separate compute from storage, support petabyte-scale queries, store a variety of unstructured data and semistructured objects, and integrate with advanced processing technologies such as Spark or Beam.

We believe that the trend of convergence will only continue. The data lake and the data warehouse will still exist as different architectures. In practice, their capabilities will converge so that few users will notice a boundary between them in their day-to-day work. We now see several vendors offering *data platforms* that combine data lake and data warehouse capabilities. From our perspective, AWS, Azure, Google Cloud, Snowflake, and Databricks are class leaders, each offering a constellation of tightly integrated tools for working with data, running the gamut from relational to completely unstructured. Instead of choosing between a data lake or data warehouse architecture, future data engineers will have the option to choose a converged data platform based on a variety of factors, including vendor, ecosystem, and relative openness.

Modern Data Stack

The *modern data stack* (Figure 3-13) is currently a trendy analytics architecture that highlights the type of abstraction we expect to see more widely used over the next several years. Whereas past data stacks relied on expensive, monolithic toolsets, the main objective of the modern data stack is to use cloud-based, plug-and-play, easy-to-use, off-the-shelf components to create a modular and cost-effective data architecture. These components include data pipelines, storage, transformation, data management/governance, monitoring, visualization, and exploration. The domain is still in flux, and the specific tools are changing and evolving rapidly, but the core aim will remain the same: to reduce complexity and increase modularization. Note that the notion of a modern data stack integrates nicely with the converged data platform idea from the previous section.

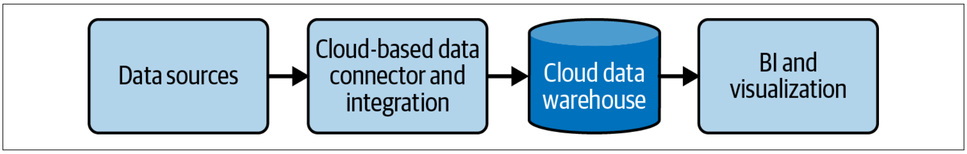


Figure 3-13. Basic components of the modern data stack

Key outcomes of the modern data stack are self-service (analytics and pipelines), agile data management, and using open source tools or simple proprietary tools with clear pricing structures. Community is a central aspect of the modern data stack as well. Unlike products of the past that had releases and roadmaps largely hidden from users, projects and companies operating in the modern data stack space typically have strong user bases and active communities that participate in the development by using the product early, suggesting features, and submitting pull requests to improve the code.

Regardless of where "modern" goes (we share our ideas in Chapter 11), we think the key concept of plug-and-play modularity with easy-to-understand pricing and implementation is the way of the future. Especially in analytics engineering, the modern data stack is and will continue to be the default choice of data architecture. Throughout the book, the architecture we reference contains pieces of the modern data stack, such as cloud-based and plug-and-play modular components.

Lambda Architecture

In the "old days" (the early to mid-2010s), the popularity of working with streaming data exploded with the emergence of Kafka as a highly scalable message queue and frameworks such as Apache Storm and Samza for streaming/real-time analytics. These technologies allowed companies to perform new types of analytics and modeling on large amounts of data, user aggregation and ranking, and product recommendations. Data engineers needed to figure out how to reconcile batch and streaming data into a single architecture. The Lambda architecture was one of the early popular responses to this problem.

In a *Lambda architecture* (Figure 3-14), you have systems operating independently of each other—batch, streaming, and serving. The source system is ideally immutable and append-only, sending data to two destinations for processing: stream and batch. In-stream processing intends to serve the data with the lowest possible latency in a "speed" layer, usually a NoSQL database. In the batch layer, data is processed and transformed in a system such as a data warehouse, creating precomputed and aggregated views of the data. The serving layer provides a combined view by aggregating query results from the two layers.

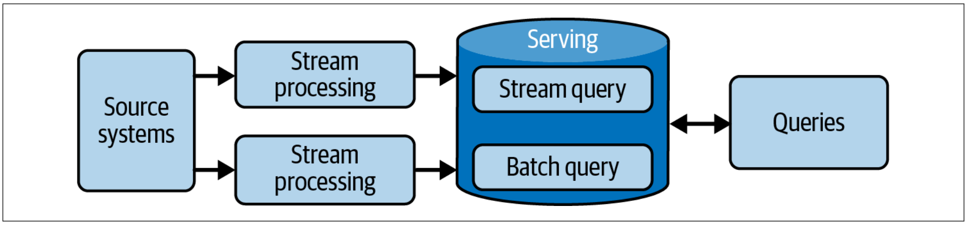


Figure 3-14. Lambda architecture

Lambda architecture has its share of challenges and criticisms. Managing multiple systems with different codebases is as difficult as it sounds, creating error-prone systems with code and data that are extremely difficult to reconcile.

We mention Lambda architecture because it still gets attention and is popular in search-engine results for data architecture. Lambda isn't our first recommendation if you're trying to combine streaming and batch data for analytics. Technology and practices have moved on.

Next, let's look at a reaction to Lambda architecture, the Kappa architecture.

Kappa Architecture

As a response to the shortcomings of Lambda architecture, Jay Kreps proposed an alternative called *Kappa architecture* (Figure 3-15).[23] The central thesis is this: why not just use a stream-processing platform as the backbone for all data handling—ingestion, storage, and serving? This facilitates a true event-based architecture. Real-time and batch processing can be applied seamlessly to the same data by reading the live event stream directly and replaying large chunks of data for batch processing.

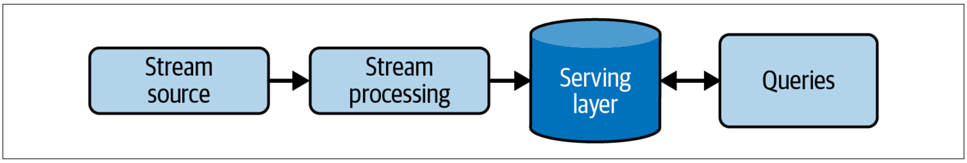


Figure 3-15. Kappa architecture

Though the original Kappa architecture article came out in 2014, we haven't seen it widely adopted. There may be a couple of reasons for this. First, streaming itself is still a bit of a mystery for many companies; it's easy to talk about, but harder than expected to execute. Second, Kappa architecture turns out to be complicated and expensive in practice. While some streaming systems can scale to huge data volumes, they are complex and expensive; batch storage and processing remain much more efficient and cost-effective for enormous historical datasets.

The Dataflow Model and Unified Batch and Streaming

Both Lambda and Kappa sought to address limitations of the Hadoop ecosystem of the 2010s by trying to duct-tape together complicated tools that were likely not natural fits in the first place. The central challenge of unifying batch and streaming data remained, and Lambda and Kappa both provided inspiration and groundwork for continued progress in this pursuit.

[23] Jay Kreps, "Questioning the Lambda Architecture," *O'Reilly Radar*, July 2, 2014, *https://oreil.ly/wWR3n*.

One of the central problems of managing batch and stream processing is unifying multiple code paths. While the Kappa architecture relies on a unified queuing and storage layer, one still has to confront using different tools for collecting real-time statistics or running batch aggregation jobs. Today, engineers seek to solve this in several ways. Google made its mark by developing the Dataflow model and the Apache Beam framework that implements this model.

The core idea in the Dataflow model is to view all data as events, as the aggregation is performed over various types of windows. Ongoing real-time event streams are *unbounded data*. Data batches are simply bounded event streams, and the boundaries provide a natural window. Engineers can choose from various windows for real-time aggregation, such as sliding or tumbling. Real-time and batch processing happens in the same system using nearly identical code.

The philosophy of "batch as a special case of streaming" is now more pervasive. Various frameworks such as Flink and Spark have adopted a similar approach.

Architecture for IoT

The *Internet of Things* (IoT) is the distributed collection of devices, aka *things*—computers, sensors, mobile devices, smart home devices, and anything else with an internet connection. Rather than generating data from direct human input (think data entry from a keyboard), IoT data is generated from devices that collect data periodically or continuously from the surrounding environment and transmit it to a destination. IoT devices are often low-powered and operate in low-resource/low bandwidth environments.

While the concept of IoT devices dates back at least a few decades, the smartphone revolution created a massive IoT swarm virtually overnight. Since then, numerous new IoT categories have emerged, such as smart thermostats, car entertainment systems, smart TVs, and smart speakers. The IoT has evolved from a futurist fantasy to a massive data engineering domain. We expect IoT to become one of the dominant ways data is generated and consumed, and this section goes a bit deeper than the others you've read.

Having a cursory understanding of IoT architecture will help you understand broader data architecture trends. Let's briefly look at some IoT architecture concepts.

Devices

Devices (also known as *things*) are the physical hardware connected to the internet, sensing the environment around them and collecting and transmitting data to a downstream destination. These devices might be used in consumer applications like a doorbell camera, smartwatch, or thermostat. The device might be an AI-powered camera that monitors an assembly line for defective components, a GPS tracker to record vehicle locations, or a Raspberry Pi programmed to download the latest tweets and brew your coffee. Any device capable of collecting data from its environment is an IoT device.

Devices should be minimally capable of collecting and transmitting data. However, the device might also crunch data or run ML on the data it collects before sending it downstream—edge computing and edge machine learning, respectively.

A data engineer doesn't necessarily need to know the inner details of IoT devices but should know what the device does, the data it collects, any edge computations or ML it runs before transmitting the data, and how often it sends data. It also helps to know the consequences of a device or internet outage, environmental or other external factors affecting data collection, and how these may impact the downstream collection of data from the device.

Interfacing with devices

A device isn't beneficial unless you can get its data. This section covers some of the key components necessary to interface with IoT devices in the wild.

IoT gateway. An *IoT gateway* is a hub for connecting devices and securely routing devices to the appropriate destinations on the internet. While you can connect a device directly to the internet without an IoT gateway, the gateway allows devices to connect using extremely little power. It acts as a way station for data retention and manages an internet connection to the final data destination.

New low-power WiFi standards are designed to make IoT gateways less critical in the future, but these are just rolling out now. Typically, a swarm of devices will utilize many IoT gateways, one at each physical location where devices are present (Figure 3-16).

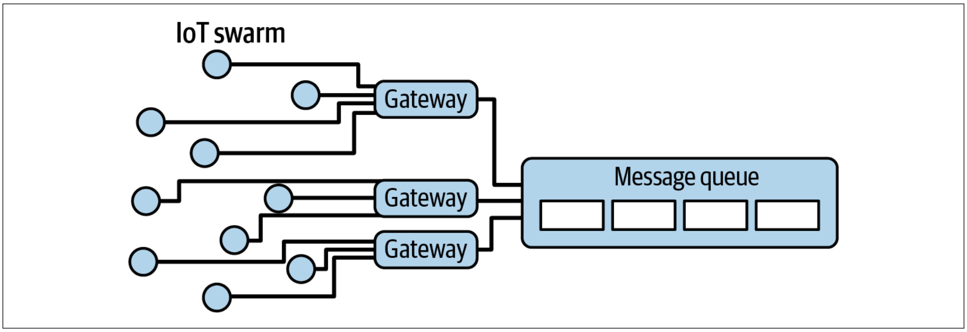


Figure 3-16. A device swarm (circles), IoT gateways, and message queue with messages (rectangles within the queue)

Ingestion. *Ingestion* begins with an IoT gateway, as discussed previously. From there, events and measurements can flow into an event ingestion architecture.

Of course, other patterns are possible. For instance, the gateway may accumulate data and upload it in batches for later analytics processing. In remote physical environments, gateways may not have connectivity to a network much of the time. They may upload all data only when they are brought into the range of a cellular or WiFi network. The point is that the diversity of IoT systems and environments presents complications—e.g., late-arriving data, data structure and schema disparities, data corruption, and connection disruption—that engineers must account for in their architectures and downstream analytics.

Storage. Storage requirements will depend a great deal on the latency requirement for the IoT devices in the system. For example, for remote sensors collecting scientific data for analysis at a later time, batch object storage may be perfectly acceptable. However, near real-time responses may be expected from a system backend that constantly analyzes data in a home monitoring and automation solution. In this case, a message queue or time-series database is more appropriate. We discuss storage systems in more detail in Chapter 6.

Serving. Serving patterns are incredibly diverse. In a batch scientific application, data might be analyzed using a cloud data warehouse and then served in a report. Data will be presented and served in numerous ways in a home-monitoring application. Data will be analyzed in the near time using a stream-processing engine or queries in a time-series database to look for critical events such as a fire, electrical outage, or break-in. Detection of an anomaly will trigger alerts to the homeowner, the fire department, or other entity. A batch analytics component also exists—for example, a monthly report on the state of the home.

One significant serving pattern for IoT looks like reverse ETL (Figure 3-17), although we tend not to use this term in the IoT context. Think of this scenario: data from sensors on manufacturing devices is collected and analyzed. The results of these measurements are processed to look for optimizations that will allow equipment to operate more efficiently. Data is sent back to reconfigure the devices and optimize them.

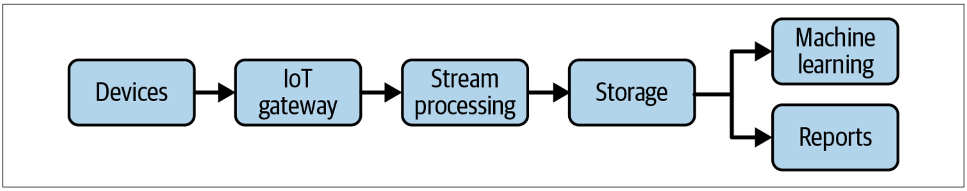


Figure 3-17. IoT serving pattern for downstream use cases

Scratching the surface of the IoT

IoT scenarios are incredibly complex, and IoT architecture and systems are also less familiar to data engineers who may have spent their careers working with business data. We hope that this introduction will encourage interested data engineers to learn more about this fascinating and rapidly evolving specialization.

Data Mesh

The *data mesh* is a recent response to sprawling monolithic data platforms, such as centralized data lakes and data warehouses, and "the great divide of data," wherein the landscape is divided between operational data and analytical data.[24] The data mesh attempts to invert the challenges of centralized data architecture, taking the concepts of domain-driven design (commonly used in software architectures) and applying them to data architecture. Because the data mesh has captured much recent attention, you should be aware of it.

A big part of the data mesh is decentralization, as Zhamak Dehghani noted in her groundbreaking article on the topic:[25]

> In order to decentralize the monolithic data platform, we need to reverse how we think about data, its locality, and ownership. Instead of flowing the data from domains into a centrally owned data lake or platform, domains need to host and serve their domain datasets in an easily consumable way.

Dehghani later identified four key components of the data mesh:[26]

- Domain-oriented decentralized data ownership and architecture
- Data as a product
- Self-serve data infrastructure as a platform
- Federated computational governance

Figure 3-18 shows a simplified version of a data mesh architecture. You can learn more about data mesh in Dehghani's book *Data Mesh* (O'Reilly).

[24] Zhamak Dehghani, "Data Mesh Principles and Logical Architecture," MartinFowler.com, December 3, 2020, *https://oreil.ly/ezWE7*.

[25] Zhamak Dehghani, "How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh," MartinFowler.com, May 20, 2019, *https://oreil.ly/SqMe8*.

[26] Zhamak Dehghani, "Data Mesh Principles and Logical Architecture."

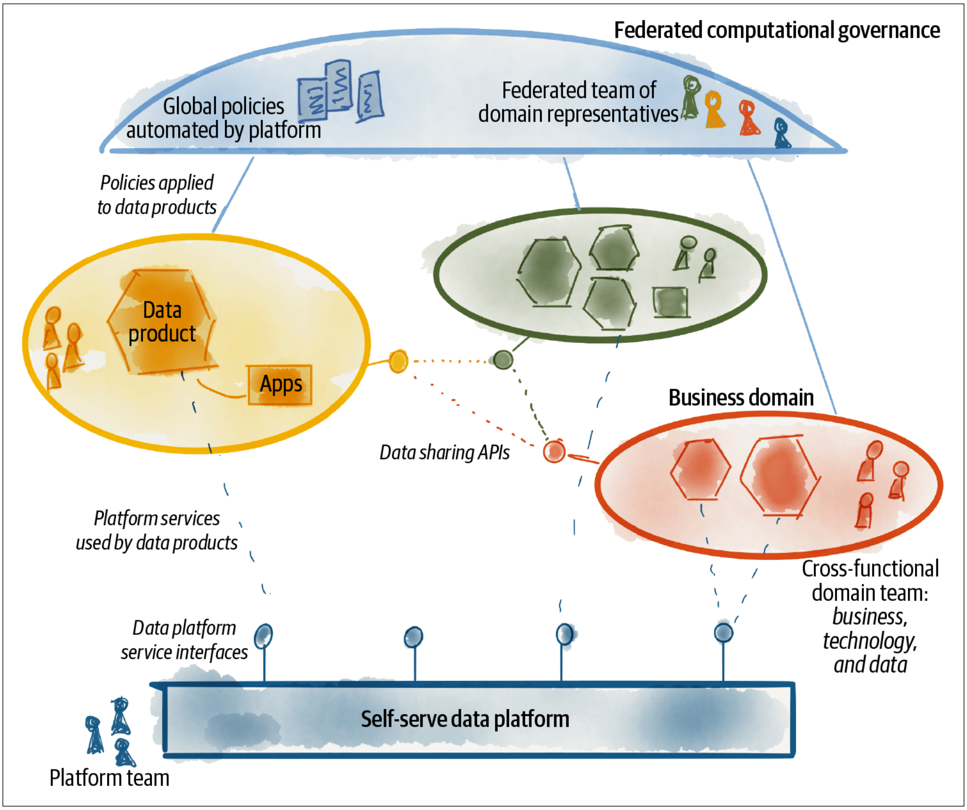

Federated computational governance
Global policies automated by platform
Federated team of domain representatives
Policies applied to data products
Data product
Apps
Data sharing APIs
Business domain
Platform services used by data products
Cross-functional domain team: business, technology, and data
Data platform service interfaces
Self-serve data platform
Platform team

Figure 3-18. Simplified example of a data mesh architecture. Source: From Data Mesh, *by Zhamak Dehghani. Copyright © 2022 Zhamak Dehghani. Published by O'Reilly Media, Inc. Used with permission.*

Other Data Architecture Examples

Data architectures have countless other variations, such as data fabric, data hub, scaled architecture, metadata-first architecture, event-driven architecture, live data stack (Chapter 11), and many more. And new architectures will continue to emerge as practices consolidate and mature, and tooling simplifies and improves. We've focused on a handful of the most critical data architecture patterns that are extremely well established, evolving rapidly, or both.

As a data engineer, pay attention to how new architectures may help your organization. Stay abreast of new developments by cultivating a high-level awareness of the data engineering ecosystem developments. Be open-minded and don't get emotionally attached to one approach. Once you've identified potential value, deepen your learning and make concrete decisions. When done right, minor tweaks—or major overhauls—in your data architecture can positively impact the business.

Who's Involved with Designing a Data Architecture?

Data architecture isn't designed in a vacuum. Bigger companies may still employ data architects, but those architects will need to be heavily in tune and current with the state of technology and data. Gone are the days of ivory tower data architecture. In the past, architecture was largely orthogonal to engineering. We expect this distinction will disappear as data engineering, and engineering in general, quickly evolves, becoming more agile, with less separation between engineering and architecture.

Ideally, a data engineer will work alongside a dedicated data architect. However, if a company is small or low in its level of data maturity, a data engineer might work double duty as an architect. Because data architecture is an undercurrent of the data engineering lifecycle, a data engineer should understand "good" architecture and the various types of data architecture.

When designing architecture, you'll work alongside business stakeholders to evaluate trade-offs. What are the trade-offs inherent in adopting a cloud data warehouse versus a data lake? What are the trade-offs of various cloud platforms? When might a unified batch/streaming framework (Beam, Flink) be an appropriate choice? Studying these choices in the abstract will prepare you to make concrete, valuable decisions.

Conclusion

You've learned how data architecture fits into the data engineering lifecycle and what makes for "good" data architecture, and you've seen several examples of data architectures. Because architecture is such a key foundation for success, we encourage you to invest the time to study it deeply and understand the trade-offs inherent in any architecture. You will be prepared to map out architecture that corresponds to your organization's unique requirements.

Next up, let's look at some approaches to choosing the right technologies to be used in data architecture and across the data engineering lifecycle.

Additional Resources

- "AnemicDomainModel" by Martin Fowler
- "Big Data Architectures" Azure documentation
- "BoundedContext" by Martin Fowler
- "A Brief Introduction to Two Data Processing Architectures—Lambda and Kappa for Big Data" by Iman Samizadeh
- "The Building Blocks of a Modern Data Platform" by Prukalpa
- "Choosing Open Wisely" by Benoit Dageville et al.

- "Choosing the Right Architecture for Global Data Distribution" Google Cloud Architecture web page
- "Column-Oriented DBMS" Wikipedia page
- "A Comparison of Data Processing Frameworks" by Ludovik Santos
- "The Cost of Cloud, a Trillion Dollar Paradox" by Sarah Wang and Martin Casado
- "The Curse of the Data Lake Monster" by Kiran Prakash and Lucy Chambers
- *Data Architecture: A Primer for the Data Scientist* by W. H. Inmon et al. (Academic Press)
- "Data Architecture: Complex vs. Complicated" by Dave Wells
- "Data as a Product vs. Data as a Service" by Justin Gage
- "The Data Dichotomy: Rethinking the Way We Treat Data and Services" by Ben Stopford
- "Data Fabric Architecture Is Key to Modernizing Data Management and Integration" by Ashutosh Gupta
- "Data Fabric Defined" by James Serra
- "Data Team Platform" by GitLab Data
- "Data Warehouse Architecture: Overview" by Roelant Vos
- "Data Warehouse Architecture" tutorial at Javatpoint
- "Defining Architecture" ISO/IEC/IEEE 42010 web page
- "The Design and Implementation of Modern Column-Oriented Database Systems" by Daniel Abadi et al.
- "Disasters I've Seen in a Microservices World" by Joao Alves
- "DomainDrivenDesign" by Martin Fowler
- "Down with Pipeline Debt: Introducing Great Expectations" by the Great Expectations project
- *EABOK* draft, edited by Paula Hagan
- EABOK website
- "EagerReadDerivation" by Martin Fowler
- "End-to-End Serverless ETL Orchestration in AWS: A Guide" by Rittika Jindal
- "Enterprise Architecture" Gartner Glossary definition
- "Enterprise Architecture's Role in Building a Data-Driven Organization" by Ashutosh Gupta
- "Event Sourcing" by Martin Fowler

- "Falling Back in Love with Data Pipelines" by Sean Knapp
- "Five Principles for Cloud-Native Architecture: What It Is and How to Master It" by Tom Grey
- "Focusing on Events" by Martin Fowler
- "Functional Data Engineering: A Modern Paradigm for Batch Data Processing" by Maxime Beauchemin
- "Google Cloud Architecture Framework" Google Cloud Architecture web page
- "How to Beat the Cap Theorem" by Nathan Marz
- "How to Build a Data Architecture to Drive Innovation—Today and Tomorrow" by Antonio Castro et al.
- "How TOGAF Defines Enterprise Architecture (EA)" by Avancier Limited
- The Information Management Body of Knowledge website
- "Introducing Dagster: An Open Source Python Library for Building Data Applications" by Nick Schrock
- "The Log: What Every Software Engineer Should Know About Real-Time Data's Unifying Abstraction" by Jay Kreps
- "Microsoft Azure IoT Reference Architecture" documentation
- Microsoft's "Azure Architecture Center"
- "Modern CI Is Too Complex and Misdirected" by Gregory Szorc
- "The Modern Data Stack: Past, Present, and Future" by Tristan Handy
- "Moving Beyond Batch vs. Streaming" by David Yaffe
- "A Personal Implementation of Modern Data Architecture: Getting Strava Data into Google Cloud Platform" by Matthew Reeve
- "Polyglot Persistence" by Martin Fowler
- "Potemkin Data Science" by Michael Correll
- "Principled Data Engineering, Part I: Architectural Overview" by Hussein Danish
- "Questioning the Lambda Architecture" by Jay Kreps
- "Reliable Microservices Data Exchange with the Outbox Pattern" by Gunnar Morling
- "ReportingDatabase" by Martin Fowler
- "The Rise of the Metadata Lake" by Prukalpa
- "Run Your Data Team Like a Product Team" by Emilie Schario and Taylor A. Murphy

- “Separating Utility from Value Add” by Ross Pettit
- “The Six Principles of Modern Data Architecture” by Joshua Klahr
- Snowflake’s “What Is Data Warehouse Architecture” web page
- “Software Infrastructure 2.0: A Wishlist” by Erik Bernhardsson
- “Staying Ahead of Data Debt” by Etai Mizrahi
- “Tactics vs. Strategy: SOA and the Tarpit of Irrelevancy” by Neal Ford
- “Test Data Quality at Scale with Deequ” by Dustin Lange et al.
- “Three-Tier Architecture” by IBM Education
- TOGAF framework website
- “The Top 5 Data Trends for CDOs to Watch Out for in 2021” by Prukalpa
- “240 Tables and No Documentation?” by Alexey Makhotkin
- “The Ultimate Data Observability Checklist” by Molly Vorwerck
- “Unified Analytics: Where Batch and Streaming Come Together; SQL and Beyond” Apache Flink Roadmap
- “UtilityVsStrategicDichotomy” by Martin Fowler
- “What Is a Data Lakehouse?” by Ben Lorica et al.
- “What Is Data Architecture? A Framework for Managing Data” by Thor Olavsrud
- “What Is the Open Data Ecosystem and Why It’s Here to Stay” by Casber Wang
- “What’s Wrong with MLOps?” by Laszlo Sragner
- “What the Heck Is Data Mesh” by Chris Riccomini
- “Who Needs an Architect” by Martin Fowler
- “Zachman Framework” Wikipedia page

About the Authors

Joe Reis is a business-minded data nerd who's worked in the data industry for 20 years, with responsibilities ranging from statistical modeling, forecasting, machine learning, data engineering, data architecture, and almost everything else in between. Joe is the CEO and cofounder of Ternary Data, a data engineering and architecture consulting firm based in Salt Lake City, Utah. In addition, he volunteers with several technology groups and teaches at the University of Utah. In his spare time, Joe likes to rock climb, produce electronic music, and take his kids on crazy adventures.

Matt Housley is a data engineering consultant and cloud specialist. After some early programming experience with Logo, Basic, and 6502 assembly, he completed a PhD in mathematics at the University of Utah. Matt then began working in data science, eventually specializing in cloud-based data engineering. He cofounded Ternary Data with Joe Reis, where he leverages his teaching experience to train future data engineers and advise teams on robust data architecture. Matt and Joe also pontificate on all things data on *The Monday Morning Data Chat*.

Colophon

The animal on the cover of *Fundamentals of Data Engineering* is the white-eared puffbird (*Nystalus chacuru*).

So named for the conspicuous patch of white at their ears, as well as for their fluffy plumage, these small, rotund birds are found across a wide swath of central South America, where they inhabit forest edges and savanna.

White-eared puffbirds are sit-and-wait hunters, perching in open spaces for long periods and feeding opportunistically on insects, lizards, and even small mammals that happen to come near. They are most often found alone or in pairs and are relatively quiet birds, vocalizing only rarely.

The International Union for Conservation of Nature has listed the white-eared puffbird as being of *least concern*, due, in part, to their extensive range and stable population. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from Shaw's *General Zoology*. 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.