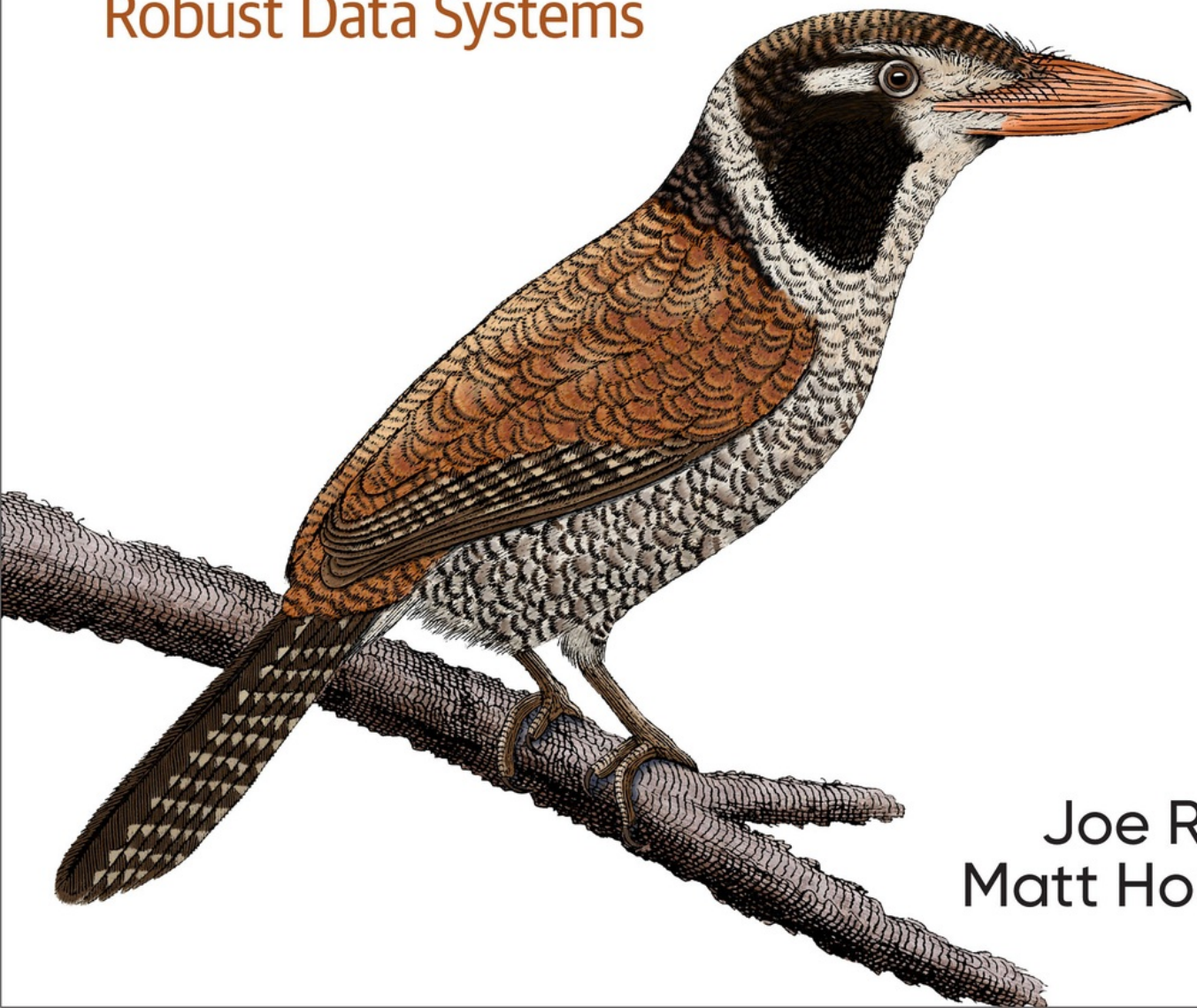
O'REILLY®

Fundamentals of Data Engineering

Plan and Build Robust Data Systems



Joe Reis &
Matt Housley

Fundamentals of Data Engineering

Plan and Build Robust Data Systems

Joe Reis and Matt Housley

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

June 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.

978-1-098-10830-4

[LSI]

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 try to 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 life cycle 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.

The Book Outline

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: cloud networking, serialization and compression

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 appendix, 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, cloud networking is a critical topic as data engineering shifts into the cloud, and engineers need to understand serialization and compression both to work directly with data files, and to assess performance considerations in data systems.

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.

`Constant width bold`

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

`Constant width italic`

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

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

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

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

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, 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 Bill Inmon, Jordan Tigani, Zhamak Dehghani, Shruti Bhat, Eric Tschetter, Benn Stancil, Kevin Hu, Michael Rogove, Ryan Wright, 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, Jake Carter, Jason Pedley, Kelly and Matt Phillipps, Brian Campbell, Faris Chebib, Dylan Gregerson, Ken Myers, 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, 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 crew 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 another 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 1. Data Engineering Described

If you work in data or software, you may have noticed data engineering emerging from the shadows and now sharing the stage with data science. Data engineering is one of the hottest fields in data and technology, and for a good reason. It builds the foundation for data science and analytics in production. This chapter explores what data engineering is, how the field was born and its evolution, the skills of data engineers, and with whom they work.

What Is Data Engineering?

Despite the current popularity of data engineering, there's a lot of confusion about what data engineering means and what data engineers do. Data engineering has existed in some form since companies started doing things with data—such as predictive analysis, descriptive analytics, and reports—and came into sharp focus alongside the rise of data science in the 2010s. For the purpose of this book, it's critical to define what *data engineering* and *data engineer* mean.

First, let's look at the landscape of how data engineering is described and develop some terminology we can use throughout this book. Endless definitions of *data engineering* exist. In early 2022, a Google exact-match search for "what is data engineering?" returns over 91,000 unique results. Before we give our definition, here are a few examples of how some experts in the field define data engineering:

Data engineering is a set of operations aimed at creating interfaces and mechanisms for the flow and access of information. It takes dedicated specialists—data engineers—to maintain data so that it remains available and usable by others. In short, data engineers set up and operate the organization's data infrastructure, preparing it for further analysis by data analysts and scientists.

—From "Data Engineering and Its Main Concepts" by AlexSoft[1]

The first type of data engineering is SQL-focused. The work and primary storage of the data is in relational databases. All of the data processing is done with SQL or a SQL-based language. Sometimes, this data processing is done with an ETL tool.[2] The second type of data engineering is Big Data–focused. The work and primary storage of the data is in Big Data technologies like Hadoop, Cassandra, and HBase. All of the data processing is done in Big Data frameworks like MapReduce, Spark, and Flink. While SQL is used, the primary processing is done with programming languages like Java, Scala, and Python.

—Jesse Anderson[3]

In relation to previously existing roles, the data engineering field could be thought of as a superset of business intelligence and data warehousing that brings more elements from software engineering. This discipline also integrates specialization around the operation of so-called "big data" distributed systems, along with concepts around the extended Hadoop ecosystem, stream processing, and in computation at scale.

—Maxime Beauchemin[4]

Data engineering is all about the movement, manipulation, and management of data.

—Lewis Gavin[5]

Wow! It's entirely understandable if you've been confused about data engineering. That's only a handful of definitions, and they contain an enormous range of opinions about the meaning of *data engineering*.

Data Engineering Defined

When we unpack the common threads of how various people define data engineering, an obvious pattern emerges: a data engineer gets data, stores it, and prepares it for consumption by data scientists, analysts, and others. We define *data engineering* and *data engineer* as follows:

> *Data engineering is the development, implementation, and maintenance of systems and processes that take in raw data and produce high-quality, consistent information that supports downstream use cases, such as analysis and machine learning. Data engineering is the intersection of security, data management, DataOps, data architecture, orchestration, and software engineering. A data engineer manages the data engineering lifecycle, beginning with getting data from source systems and ending with serving data for use cases, such as analysis or machine learning.*

The Data Engineering Lifecycle

It is all too easy to fixate on technology and miss the bigger picture myopically. This book centers around a big idea called the *data engineering lifecycle* (Figure 1-1), which we believe gives data engineers the holistic context to view their role.

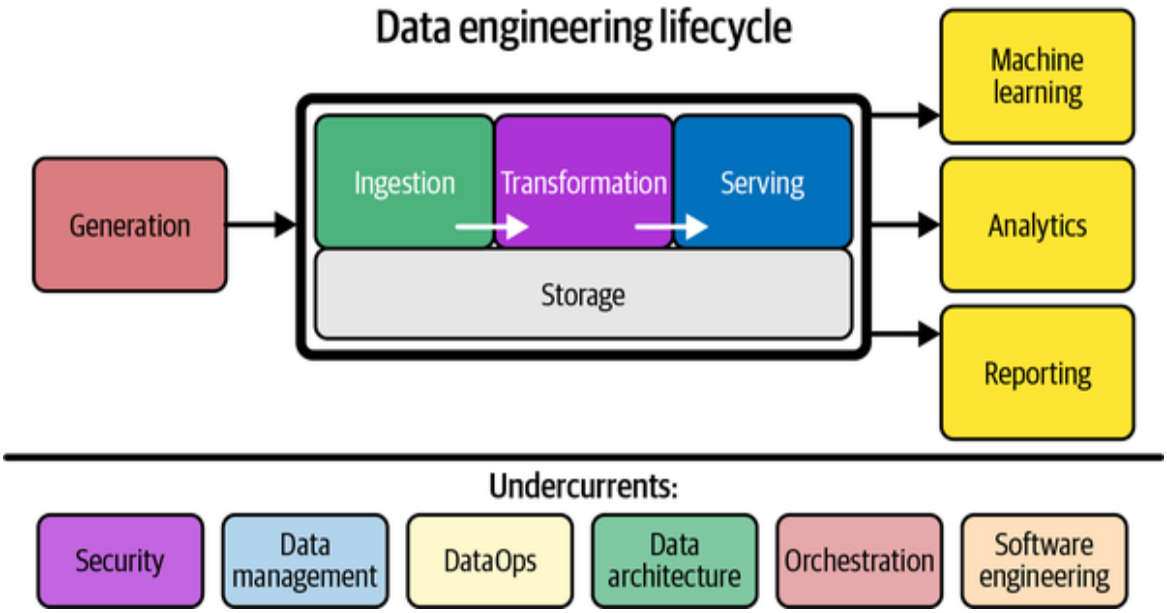


Figure 1-1. The data engineering lifecycle

The data engineering lifecycle shifts the conversation away from technology and toward the data itself and the end goals that it must serve. The stages of the data engineering lifecycle are as follows:

- Generation
- Storage
- Ingestion
- Transformation
- Serving

The data engineering lifecycle also has a notion of *undercurrents*—critical ideas across the entire lifecycle. These include security, data management, DataOps, data architecture, orchestration, and software engineering. We cover the data engineering lifecycle and its undercurrents more extensively in Chapter 2. Still, we introduce it here because it is essential to our definition of data engineering and the discussion that follows in this chapter.

Now that you have a working definition of data engineering and an introduction to its lifecycle, let's take a step back and look at a bit of history.

Evolution of the Data Engineer

> *History doesn't repeat itself, but it rhymes.*
>
> —A famous adage often attributed to Mark Twain

Understanding data engineering today and tomorrow requires a context of how the field evolved. This section is not a history lesson, but looking at the past is invaluable in understanding where we are today and where things are going. A common theme constantly reappears: what's old is new again.

The early days: 1980 to 2000, from data warehousing to the web

The birth of the data engineer arguably has its roots in data warehousing, dating as far back as the 1970s, with the *business data warehouse* taking shape in the 1980s and Bill Inmon officially coining the term *data warehouse* in 1990. After engineers at IBM developed the relational database and Structured Query Language (SQL), Oracle popularized the technology. As nascent data systems grew, businesses needed dedicated tools and data pipelines for reporting and business intelligence (BI). To help people correctly model their business logic in the data warehouse, Ralph Kimball and Inmon developed their respective eponymous data-modeling techniques and approaches, which are still widely used today.

Data warehousing ushered in the first age of scalable analytics, with new massively parallel processing (MPP) databases that use multiple processors to crunch large amounts of data coming on the market and supporting unprecedented volumes of data. Roles such as BI engineer, ETL developer, and data warehouse engineer addressed the various needs of the data warehouse. Data warehouse and BI engineering were a precursor to today's data engineering and still play a central role in the discipline.

The internet went mainstream around the mid-1990s, creating a whole new generation of web-first companies such as AOL, Yahoo, and Amazon. The dot-com boom spawned a ton of activity in web applications and the backend systems to support them—servers, databases, and storage. Much of the infrastructure was expensive, monolithic, and heavily licensed. The vendors selling these backend systems likely didn't foresee the sheer scale of the data that web applications would produce.

The early 2000s: The birth of contemporary data engineering

Fast-forward to the early 2000s, when the dot-com boom of the late '90s went bust, leaving behind a tiny cluster of survivors. Some of these companies, such as Yahoo, Google, and Amazon, would grow into powerhouse tech companies. Initially, these companies continued to rely on the traditional monolithic, relational databases and data warehouses of the 1990s, pushing these systems to the limit. As these systems buckled, updated approaches were needed to handle data growth. The new

generation of the systems must be cost-effective, scalable, available, and reliable.

Coinciding with the explosion of data, commodity hardware—such as servers, RAM, disks, and flash drives—also became cheap and ubiquitous. Several innovations allowed distributed computation and storage on massive computing clusters at a vast scale. These innovations started decentralizing and breaking apart traditionally monolithic services. The "big data" era had begun.

The *Oxford English Dictionary* defines big data as "extremely large data sets that may be analyzed computationally to reveal patterns, trends, and associations, especially relating to human behavior and interactions." Another famous and succinct description of big data is the three *V*'s of data: velocity, variety, and volume.

In 2003, Google published a paper on the Google File System, and shortly after that, in 2004, a paper on MapReduce, an ultra-scalable data-processing paradigm. In truth, big data has earlier antecedents in MPP data warehouses and data management for experimental physics projects, but Google's publications constituted a "big bang" for data technologies and the cultural roots of data engineering as we know it today. You'll learn more about MPP systems and MapReduce in Chapters 3 and 8, respectively.

The Google papers inspired engineers at Yahoo to develop and later open source Apache Hadoop in 2006.[6] It's hard to overstate the impact of Hadoop. Software engineers interested in large-scale data problems were drawn to the possibilities of this new open source technology ecosystem. As companies of all sizes and types saw their data grow into many terabytes and even petabytes, the era of the big data engineer was born.

Around the same time, Amazon had to keep up with its own exploding data needs and created elastic computing environments (Amazon Elastic Compute Cloud, or EC2), infinitely scalable storage systems (Amazon Simple Storage Service, or S3), highly scalable NoSQL databases (Amazon DynamoDB), and many other core data building blocks.[7] Amazon elected to offer these services for internal and external consumption through

Amazon Web Services (AWS), becoming the first popular public cloud. AWS created an ultra-flexible pay-as-you-go resource marketplace by virtualizing and reselling vast pools of commodity hardware. Instead of purchasing hardware for a data center, developers could simply rent compute and storage from AWS.

As AWS became a highly profitable growth engine for Amazon, other public clouds would soon follow, such as Google Cloud, Microsoft Azure, and DigitalOcean. The public cloud is arguably one of the most significant innovations of the 21st century and spawned a revolution in the way software and data applications are developed and deployed.

The early big data tools and public cloud laid the foundation for today's data ecosystem. The modern data landscape—and data engineering as we know it now—would not exist without these innovations.

The 2000s and 2010s: Big data engineering

Open source big data tools in the Hadoop ecosystem rapidly matured and spread from Silicon Valley to tech-savvy companies worldwide. For the first time, any business had access to the same bleeding-edge data tools used by the top tech companies. Another revolution occurred with the transition from batch computing to event streaming, ushering in a new era of big "real-time" data. You'll learn about batch and event streaming throughout this book.

Engineers could choose the latest and greatest—Hadoop, Apache Pig, Apache Hive, Dremel, Apache HBase, Apache Storm, Apache Cassandra, Apache Spark, Presto, and numerous other new technologies that came on the scene. Traditional enterprise-oriented and GUI-based data tools suddenly felt outmoded, and code-first engineering was in vogue with the ascendance of MapReduce. We (the authors) were around during this time, and it felt like old dogmas died a sudden death upon the altar of big data.

The explosion of data tools in the late 2000s and 2010s ushered in the *big data engineer*. To effectively use these tools and techniques—namely, the Hadoop ecosystem including Hadoop, YARN, Hadoop Distributed File

System (HDFS), and MapReduce—big data engineers had to be proficient in software development and low-level infrastructure hacking, but with a shifted emphasis. Big data engineers typically maintained massive clusters of commodity hardware to deliver data at scale. While they might occasionally submit pull requests to Hadoop core code, they shifted their focus from core technology development to data delivery.

Big data quickly became a victim of its own success. As a buzzword, *big data* gained popularity during the early 2000s through the mid-2010s. Big data captured the imagination of companies trying to make sense of the ever-growing volumes of data and the endless barrage of shameless marketing from companies selling big data tools and services. Because of the immense hype, it was common to see companies using big data tools for small data problems, sometimes standing up a Hadoop cluster to process just a few gigabytes. It seemed like everyone wanted in on the big data action. Dan Ariely tweeted, "Big data is like teenage sex: everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it."

Figure 1-2 shows a snapshot of Google Trends for the search term "big data" to get an idea of the rise and fall of big data.

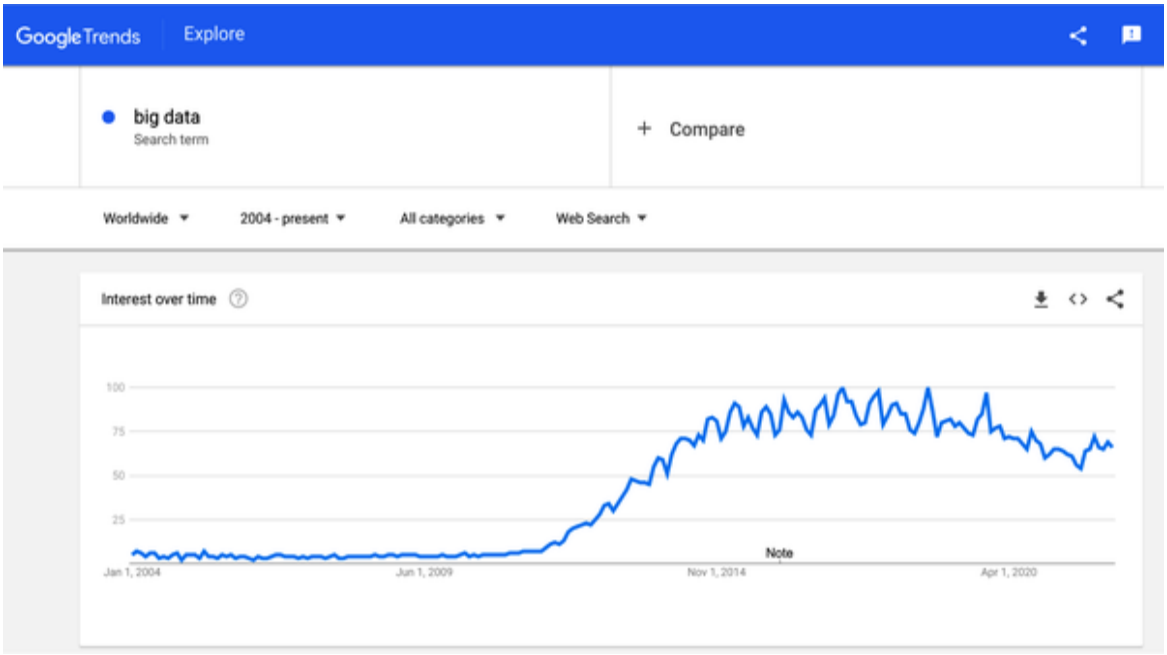


Figure 1-2. Google Trends for "big data" (March 2022)

Despite the term's popularity, big data has lost steam. What happened? One word: simplification. Despite the power and sophistication of open source big data tools, managing them was a lot of work and required constant attention. Often, companies employed entire teams of big data engineers, costing millions of dollars a year, to babysit these platforms. Big data engineers often spent excessive time maintaining complicated tooling and arguably not as much time delivering the business's insights and value.

Open source developers, clouds, and third parties started looking for ways to abstract, simplify, and make big data available without the high administrative overhead and cost of managing their clusters, and installing, configuring, and upgrading their open source code. The term *big data* is essentially a relic to describe a particular time and approach to handling large amounts of data.

Today, data is moving faster than ever and growing ever larger, but big data processing has become so accessible that it no longer merits a separate term; every company aims to solve its data problems, regardless of actual data size. Big data engineers are now simply *data engineers*.

The 2020s: Engineering for the data lifecycle

At the time of this writing, the data engineering role is evolving rapidly. We expect this evolution to continue at a rapid clip for the foreseeable future. Whereas data engineers historically tended to the low-level details of monolithic frameworks such as Hadoop, Spark, or Informatica, the trend is moving toward decentralized, modularized, managed, and highly abstracted tools.

Indeed, data tools have proliferated at an astonishing rate (see Figure 1-3). Popular trends in the early 2020s include the *modern data stack*, representing a collection of off-the-shelf open source and third-party products assembled to make analysts' lives easier. At the same time, data sources and data formats are growing both in variety and size. Data engineering is increasingly a discipline of interoperation, and connecting various technologies like LEGO bricks, to serve ultimate business goals.

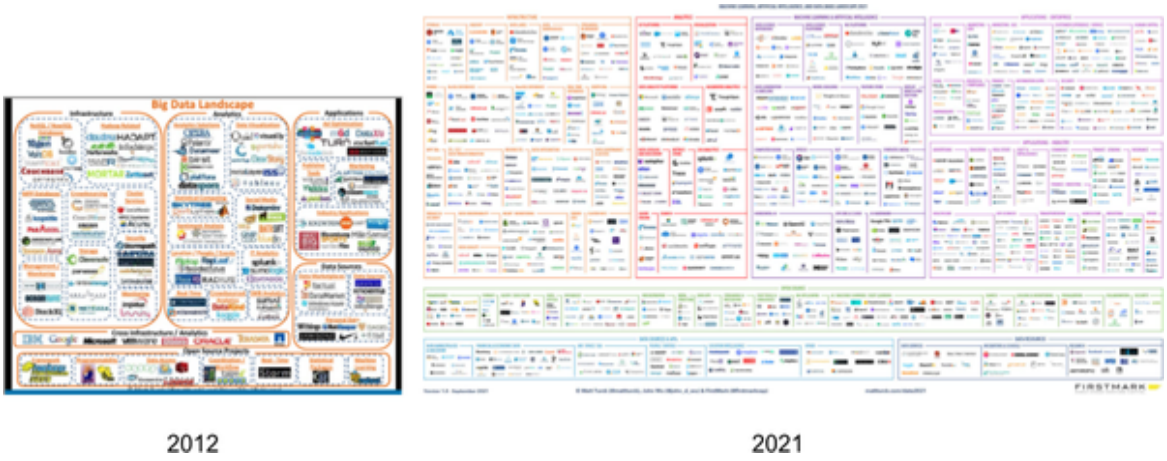


Figure 1-3. Matt Turck's Data Landscape in 2012 versus 2021

The data engineer we discuss in this book can be described more precisely as a *data lifecycle engineer*. With greater abstraction and simplification, a data lifecycle engineer is no longer encumbered by the gory details of yesterday's big data frameworks. While data engineers maintain skills in low-level data programming and use these as required, they increasingly find their role focused on things higher in the value chain: security, data management, DataOps, data architecture, orchestration, and general data lifecycle management.[8]

As tools and workflows simplify, we've seen a noticeable shift in the attitudes of data engineers. Instead of focusing on who has the "biggest data," open source projects and services are increasingly concerned with managing and governing data, making it easier to use and discover, and improving its quality. Data engineers are now conversant in acronyms such as *CCPA* and *GDPR*;[9] as they engineer pipelines, they concern themselves with privacy, anonymization, data garbage collection, and compliance with regulations.

What's old is new again. While "enterprisey" stuff like data management (including data quality and governance) was common for large enterprises in the pre-big-data era, it wasn't widely adopted in smaller companies. Now that many of the challenging problems of yesterday's data systems are solved, neatly productized, and packaged, technologists and entrepreneurs have shifted focus back to the "enterprisey" stuff, but with an emphasis on

decentralization and agility, that contrasts with the traditional enterprise command-and-control approach.

We view the present as a golden age of data lifecycle management. Data engineers managing the data engineering lifecycle have better tools and techniques than ever before. We discuss the data engineering lifecycle and its undercurrents in greater detail in the next chapter.

Data Engineering and Data Science

Where does data engineering fit in with data science? There's some debate, with some arguing data engineering is a subdiscipline of data science. We believe data engineering is *separate* from data science and analytics. They complement each other, but they are distinctly different. Data engineering sits upstream from data science (Figure 1-4), meaning data engineers provide the inputs used by data scientists (downstream from data engineering), who convert these inputs into something useful.

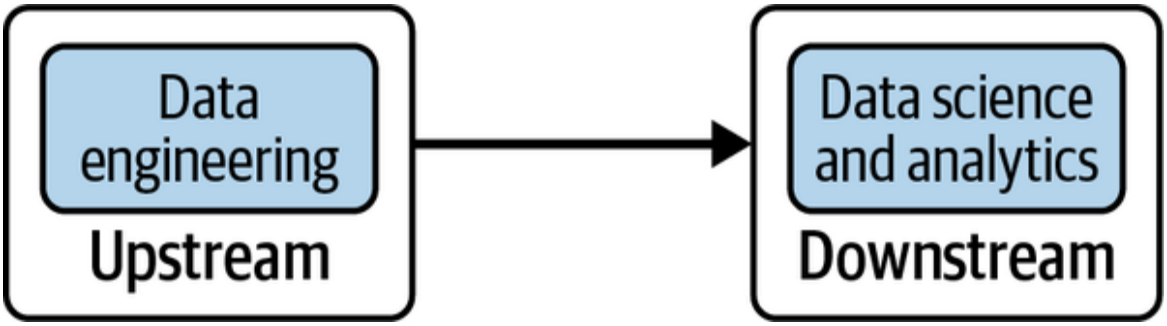


Figure 1-4. Data engineering sits upstream from data science

Consider the Data Science Hierarchy of Needs (Figure 1-5). In 2017, Monica Rogati published this hierarchy in an article that showed where AI and machine learning (ML) sat in proximity to more "mundane" areas such as data movement/storage, collection, and infrastructure.

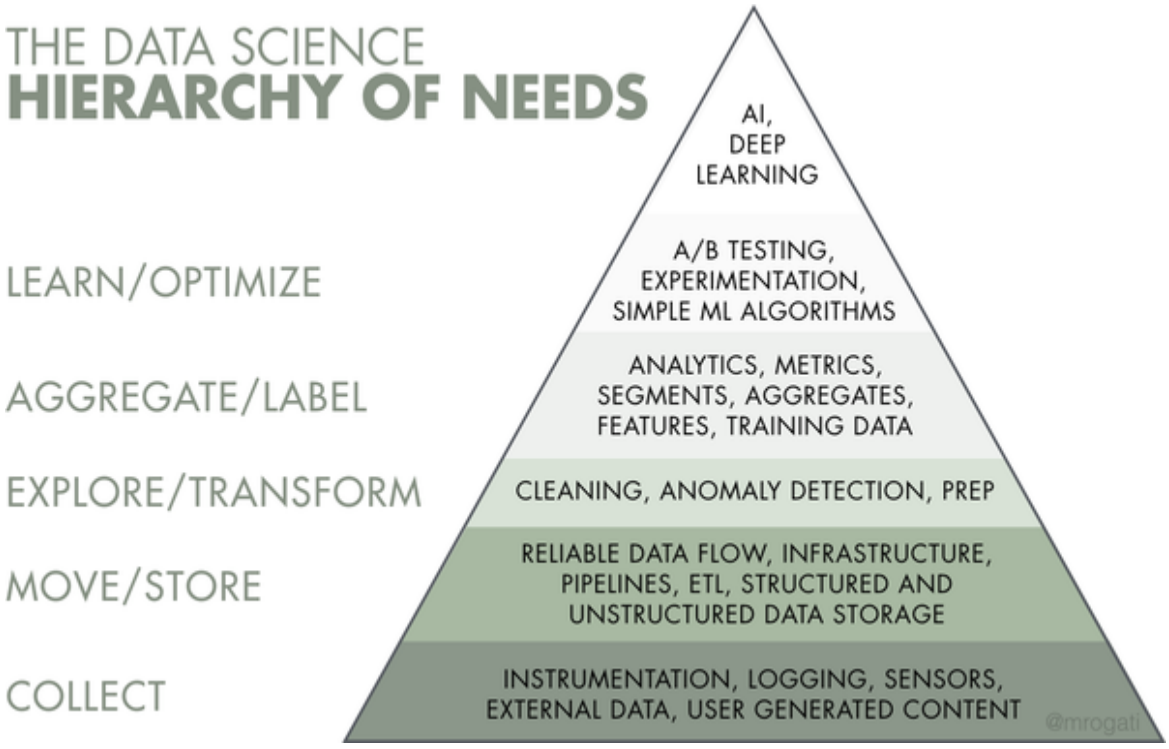


Figure 1-5. The Data Science Hierarchy of Needs

Although many data scientists are eager to build and tune ML models, the reality is an estimated 70% to 80% of their time is spent toiling in the bottom three parts of the hierarchy—gathering data, cleaning data, processing data—and only a tiny slice of their time on analysis and ML. Rogati argues that companies need to build a solid data foundation (the bottom three levels of the hierarchy) before tackling areas such as AI and ML.

Data scientists aren't typically trained to engineer production-grade data systems, and they end up doing this work haphazardly because they lack the support and resources of a data engineer. In an ideal world, data scientists should spend more than 90% of their time focused on the top layers of the pyramid: analytics, experimentation, and ML. When data engineers focus on these bottom parts of the hierarchy, they build a solid foundation for data scientists to succeed.

With data science driving advanced analytics and ML, data engineering straddles the divide between getting data and getting value from data (see Figure 1-6). We believe data engineering is of equal importance and

visibility to data science, with data engineers playing a vital role in making data science successful in production.

Figure 1-6. A data engineer gets data and provides value from the data

Data Engineering Skills and Activities

The skill set of a data engineer encompasses the "undercurrents" of data engineering: security, data management, DataOps, data architecture, and software engineering. This skill set requires an understanding of how to evaluate data tools and how they fit together across the data engineering lifecycle. It's also critical to know how data is produced in source systems and how analysts and data scientists will consume and create value after processing and curating data. Finally, a data engineer juggles a lot of complex moving parts and must constantly optimize along the axes of cost, agility, scalability, simplicity, reuse, and interoperability (Figure 1-7). We cover these topics in more detail in upcoming chapters.

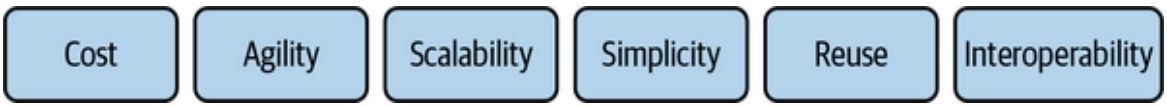


Figure 1-7. The balancing act of data engineering

As we discussed, in the recent past, a data engineer was expected to know and understand how to use a small handful of powerful and monolithic technologies (Hadoop, Spark, Teradata, Hive, and many others) to create a data solution. Utilizing these technologies often requires a sophisticated understanding of software engineering, networking, distributed computing, storage, or other low-level details. Their work would be devoted to cluster administration and maintenance, managing overhead, and writing pipeline and transformation jobs, among other tasks.

Nowadays, the data-tooling landscape is dramatically less complicated to manage and deploy. Modern data tools considerably abstract and simplify

workflows. As a result, data engineers are now focused on balancing the simplest and most cost-effective, best-of-breed services that deliver value to the business. The data engineer is also expected to create agile data architectures that evolve as new trends emerge.

What are some things a data engineer does *not* do? A data engineer typically does not directly build ML models, create reports or dashboards, perform data analysis, build key performance indicators (KPIs), or develop software applications. A data engineer should have a good functioning understanding of these areas to serve stakeholders best.

Data Maturity and the Data Engineer

The level of data engineering complexity within a company depends a great deal on the company's data maturity. This significantly impacts a data engineer's day-to-day job responsibilities and career progression. What is data maturity, exactly?

Data maturity is the progression toward higher data utilization, capabilities, and integration across the organization, but data maturity does not simply depend on the age or revenue of a company. An early-stage startup can have greater data maturity than a 100-year-old company with annual revenues in the billions. What matters is the way data is leveraged as a competitive advantage.

Data maturity models have many versions, such as Data Management Maturity (DMM) and others, and it's hard to pick one that is both simple and useful for data engineering. So, we'll create our own simplified data maturity model. Our data maturity model (Figure 1-8) has three stages: starting with data, scaling with data, and leading with data. Let's look at each of these stages and at what a data engineer typically does at each stage.

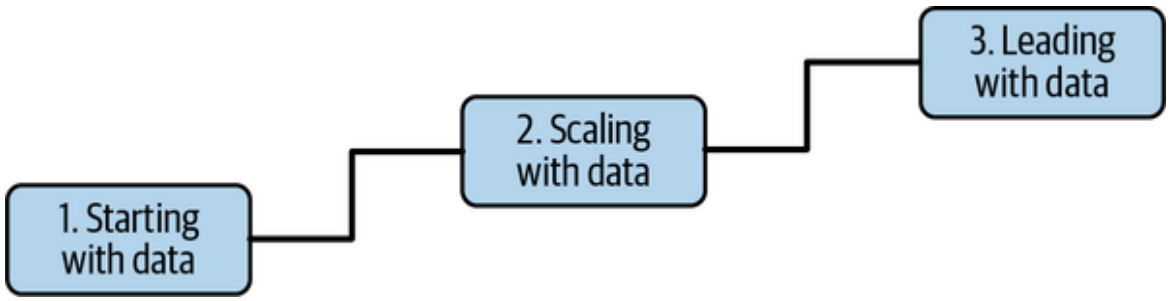


Figure 1-8. Our simplified data maturity model for a company

Stage 1: Starting with data

A company getting started with data is, by definition, in the very early stages of its data maturity. The company may have fuzzy, loosely defined goals or no goals. Data architecture and infrastructure are in the very early stages of planning and development. Adoption and utilization are likely low or nonexistent. The data team is small, often with a headcount in the single digits. At this stage, a data engineer is usually a generalist and will typically play several other roles, such as data scientist or software engineer. A data engineer's goal is to move fast, get traction, and add value.

The practicalities of getting value from data are typically poorly understood, but the desire exists. Reports or analyses lack formal structure, and most requests for data are ad hoc. While it's tempting to jump headfirst into ML at this stage, we don't recommend it. We've seen countless data teams get stuck and fall short when they try to jump to ML without building a solid data foundation.

That's not to say you can't get wins from ML at this stage—it is rare but possible. Without a solid data foundation, you likely won't have the data to train reliable ML models nor the means to deploy these models to production in a scalable and repeatable way. We half-jokingly call ourselves "recovering data scientists", mainly from personal experience with being involved in premature data science projects without adequate data maturity or data engineering support.

A data engineer should focus on the following in organizations getting started with data:

- Get buy-in from key stakeholders, including executive management. Ideally, the data engineer should have a sponsor for critical initiatives to design and build a data architecture to support the company's goals.
- Define the right data architecture (usually solo, since a data architect likely isn't available). This means determining business goals and the competitive advantage you're aiming to achieve with your data initiative. Work toward a data architecture that supports these goals. See Chapter 3 for our advice on "good" data architecture.
- Identify and audit data that will support key initiatives and operate within the data architecture you designed.
- Build a solid data foundation for future data analysts and data scientists to generate reports and models that provide competitive value. In the meantime, you may also have to generate these reports and models until this team is hired.

This is a delicate stage with lots of pitfalls. Here are some tips for this stage:

- Organizational willpower may wane if a lot of visible successes don't occur with data. Getting quick wins will establish the importance of data within the organization. Just keep in mind that quick wins will likely create technical debt. Have a plan to reduce this debt, as it will otherwise add friction for future delivery.
- Get out and talk to people, and avoid working in silos. We often see the data team working in a bubble, not communicating with people outside their departments and getting perspectives and feedback from business stakeholders. The danger is you'll spend a lot of time working on things of little use to people.
- Avoid undifferentiated heavy lifting. Don't box yourself in with unnecessary technical complexity. Use off-the-shelf, turnkey solutions wherever possible.

- Build custom solutions and code only where this creates a competitive advantage.

Stage 2: Scaling with data

At this point, a company has moved away from ad hoc data requests and has formal data practices. Now the challenge is creating scalable data architectures and planning for a future where the company is genuinely data-driven. Data engineering roles move from generalists to specialists, with people focusing on particular aspects of the data engineering lifecycle.

In organizations that are in stage 2 of data maturity, a data engineer's goals are to do the following:

- Establish formal data practices
- Create scalable and robust data architectures
- Adopt DevOps and DataOps practices
- Build systems that support ML
- Continue to avoid undifferentiated heavy lifting and customize only when a competitive advantage results

We return to each of these goals later in the book.

Issues to watch out for include the following:

- As we grow more sophisticated with data, there's a temptation to adopt bleeding-edge technologies based on social proof from Silicon Valley companies. This is rarely a good use of your time and energy. Any technology decisions should be driven by the value they'll deliver to your customers.
- The main bottleneck for scaling is not cluster nodes, storage, or technology but the data engineering team. Focus on solutions that are simple to deploy and manage to expand your team's throughput.

- You'll be tempted to frame yourself as a technologist, a data genius who can deliver magical products. Shift your focus instead to pragmatic leadership and begin transitioning to the next maturity stage; communicate with other teams about the practical utility of data. Teach the organization how to consume and leverage data.

Stage 3: Leading with data

At this stage, the company is data-driven. The automated pipelines and systems created by data engineers allow people within the company to do self-service analytics and ML. Introducing new data sources is seamless, and tangible value is derived. Data engineers implement proper controls and practices to ensure that data is always available to the people and systems. Data engineering roles continue to specialize more deeply than in stage 2.

In organizations in stage 3 of data maturity, a data engineer will continue building on prior stages, plus they will do the following:

- Create automation for the seamless introduction and usage of new data
- Focus on building custom tools and systems that leverage data as a competitive advantage
- Focus on the "enterprisey" aspects of data, such as data management (including data governance and quality) and DataOps
- Deploy tools that expose and disseminate data throughout the organization, including data catalogs, data lineage tools, and metadata management systems
- Collaborate efficiently with software engineers, ML engineers, analysts, and others
- Create a community and environment where people can collaborate and speak openly, no matter their role or position

Issues to watch out for include the following:

- At this stage, complacency is a significant danger. Once organizations reach stage 3, they must constantly focus on maintenance and improvement or risk falling back to a lower stage.
- Technology distractions are a more significant danger here than in the other stages. There's a temptation to pursue expensive hobby projects that don't deliver value to the business. Utilize custom-built technology only where it provides a competitive advantage.

The Background and Skills of a Data Engineer

Data engineering is a fast-growing field, and a lot of questions remain about how to become a data engineer. Because data engineering is a relatively new discipline, little formal training is available to enter the field. Universities don't have a standard data engineering path. Although a handful of data engineering boot camps and online tutorials cover random topics, a common curriculum for the subject doesn't yet exist.

People entering data engineering arrive with varying backgrounds in education, career, and skill set. Everyone entering the field should expect to invest a significant amount of time in self-study. Reading this book is a good starting point; one of the primary goals of this book is to give you a foundation for the knowledge and skills we think are necessary to succeed as a data engineer.

If you're pivoting your career into data engineering, we've found that the transition is easiest when moving from an adjacent field, such as software engineering, ETL development, database administration, data science, or data analysis. These disciplines tend to be "data aware" and provide good context for data roles in an organization. They also equip folks with the relevant technical skills and context to solve data engineering problems.

Despite the lack of a formalized path, a requisite body of knowledge exists that we believe a data engineer should know to be successful. By definition, a data engineer must understand both data and technology. With respect to data, this entails knowing about various best practices around data management. On the technology end, a data engineer must be aware of

various options for tools, their interplay, and their trade-offs. This requires a good understanding of software engineering, DataOps, and data architecture.

Zooming out, a data engineer must also understand the requirements of data consumers (data analysts and data scientists) and the broader implications of data across the organization. Data engineering is a holistic practice; the best data engineers view their responsibilities through business and technical lenses.

Business Responsibilities

The macro responsibilities we list in this section aren't exclusive to data engineers, but are crucial for anyone working in a data or technology field. Because a simple Google search will yield tons of resources to learn about these areas, we will simply list them for brevity:

Know how to communicate with nontechnical and technical people.

> Communication is key, and you need to be able to establish rapport and trust with people across the organization. We suggest paying close attention to organizational hierarchies, who reports to whom, how people interact, and which silos exist. These observations will be invaluable to your success.

Understand how to scope and gather business and product requirements.

> You need to know what to build and ensure that your stakeholders agree with your assessment. In addition, develop a sense of how data and technology decisions impact the business.

Understand the cultural foundations of Agile, DevOps, and DataOps.

> Many technologists mistakenly believe these practices are solved through technology. We feel this is dangerously wrong. Agile, DevOps,

and DataOps are fundamentally cultural, requiring buy-in across the organization.

Control costs.

You'll be successful when you can keep costs low while providing outsized value. Know how to optimize for time to value, the total cost of ownership, and opportunity cost. Learn to monitor costs to avoid surprises.

Learn continuously.

The data field feels like it's changing at light speed. People who succeed in it are great at picking up new things while sharpening their fundamental knowledge. They're also good at filtering, determining which new developments are most relevant to their work, which are still immature, and which are just fads. Stay abreast of the field and learn how to learn.

A successful data engineer always zooms out to understand the big picture and how to achieve outsized value for the business. Communication is vital, both for technical and nontechnical people. We often see data teams succeed based on their communication with other stakeholders; success or failure is rarely a technology issue. Knowing how to navigate an organization, scope and gather requirements, control costs, and continuously learn will set you apart from the data engineers who rely solely on their technical abilities to carry their career.

Technical Responsibilities

You must understand how to build architectures that optimize performance and cost at a high level, using prepackaged or homegrown components.

Ultimately, architectures and constituent technologies are building blocks to serve the data engineering lifecycle. Recall the stages of the data engineering lifecycle:

- Generation
- Storage
- Ingestion
- Transformation
- Serving

The undercurrents of the data engineering lifecycle are the following:

- Security
- Data management
- DataOps
- Data architecture
- Software engineering

Zooming in a bit, we discuss some of the tactical data and technology skills you'll need as a data engineer in this section; we discuss these in more detail in subsequent chapters.

People often ask, should a data engineer know how to code? Short answer: yes. A data engineer should have production-grade software engineering chops. We note that the nature of software development projects undertaken by data engineers has changed fundamentally in the last few years. Fully managed services now replace a great deal of low-level programming effort previously expected of engineers, who now use managed open source, and simple plug-and-play software-as-a-service (SaaS) offerings. For example, data engineers now focus on high-level abstractions or writing pipelines as code within an orchestration framework.

Even in a more abstract world, software engineering best practices provide a competitive advantage, and data engineers who can dive into the deep architectural details of a codebase give their companies an edge when specific technical needs arise. In short, a data engineer who can't write production-grade code will be severely hindered, and we don't see this changing anytime soon. Data engineers remain software engineers, in addition to their many other roles.

What languages should a data engineer know? We divide data engineering programming languages into primary and secondary categories. At the time of this writing, the primary languages of data engineering are SQL, Python, a Java Virtual Machine (JVM) language (usually Java or Scala), and bash:

SQL

: The most common interface for databases and data lakes. After briefly being sidelined by the need to write custom MapReduce code for big data processing, SQL (in various forms) has reemerged as the lingua franca of data.

Python

: The bridge language between data engineering and data science. A growing number of data engineering tools are written in Python or have Python APIs. It's known as "the second-best language at everything." Python underlies popular data tools such as pandas, NumPy, Airflow, sci-kit learn, TensorFlow, PyTorch, and PySpark. Python is the glue between underlying components and is frequently a first-class API language for interfacing with a framework.

JVM languages such as Java and Scala

: Prevalent for Apache open source projects such as Spark, Hive, and Druid. The JVM is generally more performant than Python and may

provide access to lower-level features than a Python API (for example, this is the case for Apache Spark and Beam). Understanding Java or Scala will be beneficial if you're using a popular open source data framework.

bash

The command-line interface for Linux operating systems. Knowing bash commands and being comfortable using CLIs will significantly improve your productivity and workflow when you need to script or perform OS operations. Even today, data engineers frequently use command-line tools like awk or sed to process files in a data pipeline or call bash commands from orchestration frameworks. If you're using Windows, feel free to substitute PowerShell for bash.

THE UNREASONABLE EFFECTIVENESS OF SQL

The advent of MapReduce and the big data era relegated SQL to passé status. Since then, various developments have dramatically enhanced the utility of SQL in the data engineering lifecycle. Spark SQL, Google BigQuery, Snowflake, Hive, and many other data tools can process massive amounts of data by using declarative, set-theoretic SQL semantics. SQL is also supported by many streaming frameworks, such as Apache Flink, Beam, and Kafka. We believe that competent data engineers should be highly proficient in SQL.

Are we saying that SQL is a be-all and end-all language? Not at all. SQL is a powerful tool that can quickly solve complex analytics and data transformation problems. Given that time is a primary constraint for data engineering team throughput, engineers should embrace tools that combine simplicity and high productivity. Data engineers also do well to develop expertise in composing SQL with other operations, either within frameworks such as Spark and Flink or by using orchestration to combine multiple tools. Data engineers should also learn modern SQL semantics for dealing with JavaScript Object Notation (JSON) parsing and nested data and consider leveraging a SQL management framework such as dbt (Data Build Tool).

A proficient data engineer also recognizes when SQL is not the right tool for the job and can choose and code in a suitable alternative. A SQL expert could likely write a query to stem and tokenize raw text in a natural language processing (NLP) pipeline but would also recognize that coding in native Spark is a far superior alternative to this masochistic exercise.

Data engineers may also need to develop proficiency in secondary programming languages, including R, JavaScript, Go, Rust, C/C++, C#, and Julia. Developing in these languages is often necessary when popular across the company or used with domain-specific data tools. For instance, JavaScript has proven popular as a language for user-defined functions in

cloud data warehouses. At the same time, C# and PowerShell are essential in companies that leverage Azure and the Microsoft ecosystem.

KEEPING PACE IN A FAST-MOVING FIELD

Once a new technology rolls over you, if you're not part of the steamroller, you're part of the road.

—Stewart Brand

How do you keep your skills sharp in a rapidly changing field like data engineering? Should you focus on the latest tools or deep dive into fundamentals? Here's our advice: focus on the fundamentals to understand what's not going to change; pay attention to ongoing developments to know where the field is going. New paradigms and practices are introduced all the time, and it's incumbent on you to stay current. Strive to understand how new technologies will be helpful in the lifecycle.

The Continuum of Data Engineering Roles, from A to B

Although job descriptions paint a data engineer as a "unicorn" who must possess every data skill imaginable, data engineers don't all do the same type of work or have the same skill set. Data maturity is a helpful guide to understanding the types of data challenges a company will face as it grows its data capability. It's beneficial to look at some critical distinctions in the kinds of work data engineers do. Though these distinctions are simplistic, they clarify what data scientists and data engineers do and avoid lumping either role into the unicorn bucket.

In data science, there's the notion of type A and type B data scientists.[10] *Type A data scientists*—where *A* stands for *analysis*—focus on understanding and deriving insight from data. *Type B data scientists*—where *B* stands for *building*—share similar backgrounds as type A data scientists and possess strong programming skills. The type B data scientist builds systems that make data science work in production. Borrowing from

this data scientist continuum, we'll create a similar distinction for two types of data engineers:

Type A data engineers

> *A* stands for *abstraction*. In this case, the data engineer avoids undifferentiated heavy lifting, keeping data architecture as abstract and straightforward as possible and not reinventing the wheel. Type A data engineers manage the data engineering lifecycle mainly by using entirely off-the-shelf products, managed services, and tools. Type A data engineers work at companies across industries and at all levels of data maturity.

Type B data engineers

> *B* stands for *build*. Type B data engineers build data tools and systems that scale and leverage a company's core competency and competitive advantage. In the data maturity range, a type B data engineer is more commonly found at companies in stage 2 and 3 (scaling and leading with data), or when an initial data use case is so unique and mission-critical that custom data tools are required to get started.

Type A and type B data engineers may work in the same company and may even be the same person! More commonly, a type A data engineer is first hired to set the foundation, with type B data engineer skill sets either learned or hired as the need arises within a company.

Data Engineers Inside an Organization

Data engineers don't work in a vacuum. Depending on what they're working on, they will interact with technical and nontechnical people and

face different directions (internal and external). Let's explore what data engineers do inside an organization and with whom they interact.

Internal-Facing Versus External-Facing Data Engineers

A data engineer serves several end users and faces many internal and external directions (Figure 1-9). Since not all data engineering workloads and responsibilities are the same, it's essential to understand whom the data engineer serves. Depending on the end-use cases, a data engineer's primary responsibilities are external facing, internal facing, or a blend of the two.

Figure 1-9. The directions a data engineer faces

An *external-facing* data engineer typically aligns with the users of external-facing applications, such as social media apps, Internet of Things (IoT) devices, and ecommerce platforms. This data engineer architects, builds, and manages the systems that collect, store, and process transactional and event data from these applications. The systems built by these data engineers have a feedback loop from the application to the data pipeline, and then back to the application (Figure 1-10).

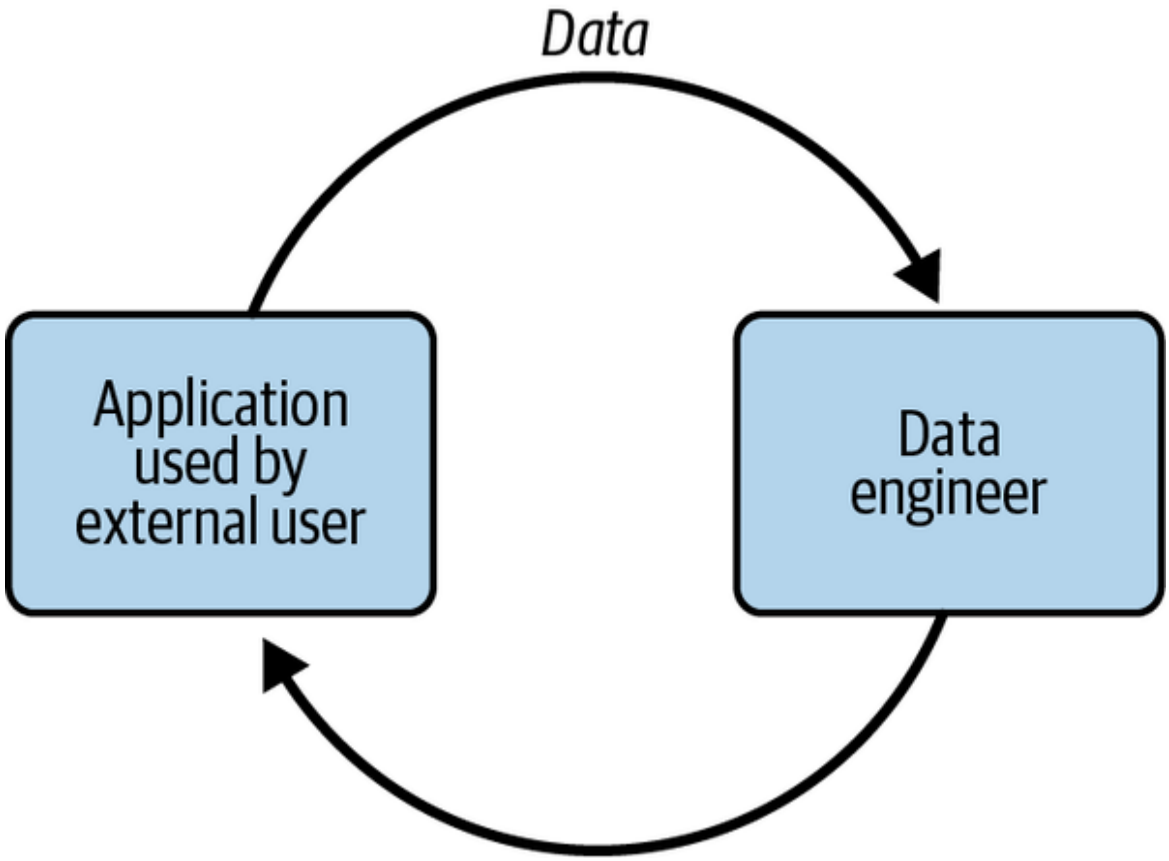


Figure 1-10. External-facing data engineer systems

External-facing data engineering comes with a unique set of problems. External-facing query engines often handle much larger concurrency loads than internal-facing systems. Engineers also need to consider putting tight limits on queries that users can run to limit the infrastructure impact of any single user. In addition, security is a much more complex and sensitive problem for external queries, especially if the data being queried is multitenant (data from many customers and housed in a single table).

An *internal-facing data engineer* typically focuses on activities crucial to the needs of the business and internal stakeholders (Figure 1-11). Examples include creating and maintaining data pipelines and data warehouses for BI dashboards, reports, business processes, data science, and ML models.

Figure 1-11. Internal-facing data engineer

External-facing and internal-facing responsibilities are often blended. In practice, internal-facing data is usually a prerequisite to external-facing data. The data engineer has two sets of users with very different requirements for query concurrency, security, and more.

Data Engineers and Other Technical Roles

In practice, the data engineering lifecycle cuts across many domains of responsibility. Data engineers sit at the nexus of various roles, directly or through managers, interacting with many organizational units.

Let's look at whom a data engineer may impact. In this section, we'll discuss technical roles connected to data engineering (Figure 1-12).

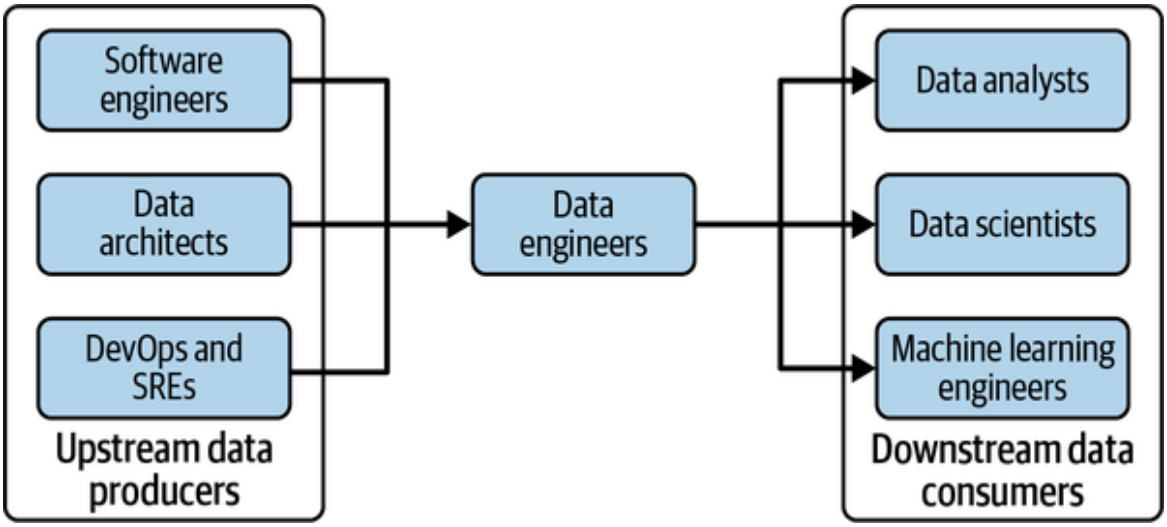


Figure 1-12. Key technical stakeholders of data engineering

The data engineer is a hub between *data producers*, such as software engineers, data architects, and DevOps or site-reliability engineers (SREs), and *data consumers*, such as data analysts, data scientists, and ML engineers. In addition, data engineers will interact with those in operational roles, such as DevOps engineers.

Given the pace at which new data roles come into vogue (analytics and ML engineers come to mind), this is by no means an exhaustive list.

Upstream stakeholders

To be successful as a data engineer, you need to understand the data architecture you're using or designing and the source systems producing the data you'll need. Next, we discuss a few familiar upstream stakeholders: data architects, software engineers, and DevOps engineers.

Data architects

Data architects function at a level of abstraction one step removed from data engineers. Data architects design the blueprint for organizational data management, mapping out processes and overall data architecture and systems.[11] They also serve as a bridge between an organization's technical and nontechnical sides. Successful data architects generally have "battle scars" from extensive engineering experience, allowing them to guide and assist engineers while successfully communicating engineering challenges to nontechnical business stakeholders.

Data architects implement policies for managing data across silos and business units, steer global strategies such as data management and data governance, and guide significant initiatives. Data architects often play a central role in cloud migrations and greenfield cloud design.

The advent of the cloud has shifted the boundary between data architecture and data engineering. Cloud data architectures are much more fluid than on-premises systems, so architecture decisions that traditionally involved extensive study, long lead times, purchase contracts, and hardware installation are now often made during the implementation process, just one step in a larger strategy. Nevertheless, data architects will remain influential visionaries in enterprises, working hand in hand with data engineers to determine the big picture of architecture practices and data strategies.

Depending on the company's data maturity and size, a data engineer may overlap with or assume the responsibilities of a data architect. Therefore, a data engineer should have a good understanding of architecture best practices and approaches.

Note that we have placed data architects in the *upstream stakeholders* section. Data architects often help design application data layers that are source systems for data engineers. Architects may also interact with data

engineers at various other stages of the data engineering lifecycle. We cover "good" data architecture in Chapter 3.

Software engineers

Software engineers build the software and systems that run a business; they are largely responsible for generating the *internal data* that data engineers will consume and process. The systems built by software engineers typically generate application event data and logs, which are significant assets in their own right. This internal data contrasts with *external data* pulled from SaaS platforms or partner businesses. In well-run technical organizations, software engineers and data engineers coordinate from the inception of a new project to design application data for consumption by analytics and ML applications.

A data engineer should work together with software engineers to understand the applications that generate data, the volume, frequency, and format of the generated data, and anything else that will impact the data engineering lifecycle, such as data security and regulatory compliance. For example, this might mean setting upstream expectations on what the data software engineers need to do their jobs. Data engineers must work closely with the software engineers.

DevOps engineers and site-reliability engineers

DevOps and SREs often produce data through operational monitoring. We classify them as upstream of data engineers, but they may also be downstream, consuming data through dashboards or interacting with data engineers directly in coordinating operations of data systems.

Downstream stakeholders

The modern data engineering profession exists to serve downstream data consumers and use cases. This section discusses how data engineers interact with various downstream roles. We'll also introduce a few service models, including centralized data engineering teams and cross-functional teams.

Data scientists

Data scientists build forward-looking models to make predictions and recommendations. These models are then evaluated on live data to provide value in various ways. For example, model scoring might determine automated actions in response to real-time conditions, recommend products to customers based on the browsing history in their current session, or make live economic predictions used by traders.

According to common industry folklore, data scientists spend 70% to 80% of their time collecting, cleaning, and preparing data.[12] In our experience, these numbers often reflect immature data science and data engineering practices. In particular, many popular data science frameworks can become bottlenecks if they are not scaled up appropriately. Data scientists who work exclusively on a single workstation force themselves to downsample data, making data preparation significantly more complicated and potentially compromising the quality of the models they produce. Furthermore, locally developed code and environments are often difficult to deploy in production, and a lack of automation significantly hampers data science workflows. If data engineers do their job and collaborate successfully, data scientists shouldn't spend their time collecting, cleaning, and preparing data after initial exploratory work. Data engineers should automate this work as much as possible.

The need for production-ready data science is a significant driver behind the emergence of the data engineering profession. Data engineers should help data scientists to enable a path to production. In fact, we (the authors) moved from data science to data engineering after recognizing this fundamental need. Data engineers work to provide the data automation and scale that make data science more efficient.

Data analysts

Data analysts (or business analysts) seek to understand business performance and trends. Whereas data scientists are forward-looking, a data analyst typically focuses on the past or present. Data analysts usually run SQL queries in a data warehouse or a data lake. They may also utilize spreadsheets for computation and analysis and various BI tools such as

Microsoft Power BI, Looker, or Tableau. Data analysts are domain experts in the data they work with frequently and become intimately familiar with data definitions, characteristics, and quality problems. A data analyst's typical downstream customers are business users, management, and executives.

Data engineers work with data analysts to build pipelines for new data sources required by the business. Data analysts' subject-matter expertise is invaluable in improving data quality, and they frequently collaborate with data engineers in this capacity.

Machine learning engineers and AI researchers

Machine learning engineers (ML engineers) overlap with data engineers and data scientists. ML engineers develop advanced ML techniques, train models, and design and maintain the infrastructure running ML processes in a scaled production environment. ML engineers often have advanced working knowledge of ML and deep learning techniques, and frameworks such as PyTorch or TensorFlow.

ML engineers also understand the hardware, services, and systems required to run these frameworks, both for model training and model deployment at a production scale. It's common for ML flows to run in a cloud environment where ML engineers can spin up and scale infrastructure resources on demand or rely on managed services.

As we've mentioned, the boundaries between ML engineering, data engineering, and data science are blurry. Data engineers may have some DevOps responsibilities over ML systems, and data scientists may work closely with ML engineering in designing advanced ML processes.

The world of ML engineering is snowballing and parallels a lot of the same developments occurring in data engineering. Whereas several years ago, the attention of ML was focused on how to build models, ML engineering now increasingly emphasizes incorporating best practices of machine learning operations (MLOps) and other mature practices previously adopted in software engineering and DevOps.

AI researchers work on new, advanced ML techniques. AI researchers may work inside large technology companies, specialized intellectual property startups (OpenAI, DeepMind), or academic institutions. Some practitioners are dedicated to part-time research in conjunction with ML engineering responsibilities inside a company. Those working inside specialized ML labs are often 100% dedicated to research. Research problems may target immediate practical applications or more abstract demonstrations of AI. AlphaGo and GPT-3/GPT-4 are great examples of ML research projects. We've provided some references in "Additional Resources".

AI researchers in well-funded organizations are highly specialized and operate with supporting teams of engineers to facilitate their work. ML engineers in academia usually have fewer resources but rely on teams of graduate students, postdocs, and university staff to provide engineering support. ML engineers who are partially dedicated to research often rely on the same support teams for research and production.

Data Engineers and Business Leadership

We've discussed technical roles with which a data engineer interacts. But data engineers also operate more broadly as organizational connectors, often in a nontechnical capacity. Businesses have come to rely increasingly on data as a core part of many products or a product in itself. Data engineers now participate in strategic planning and lead key initiatives that extend beyond the boundaries of IT. Data engineers often support data architects by acting as the glue between the business and data science/analytics.

Data in the C-suite

C-level executives are increasingly involved in data and analytics, as these are recognized as significant assets for modern businesses. CEOs now concern themselves with initiatives that were once the exclusive province of IT, such as cloud migrations or deployment of a new customer data platform.

Chief executive officer

Chief executive officers (CEOs) at nontech companies generally don't concern themselves with the nitty-gritty of data frameworks and software. Instead, they define a vision in collaboration with technical C-suite roles and company data leadership. Data engineers provide a window into what's possible with data. Data engineers and their managers maintain a map of what data is available to the organization—both internally and from third parties—in what time frame. They are also tasked to study primary data architectural changes in collaboration with other engineering roles. For example, data engineers are often heavily involved in cloud migrations, migrations to new data systems, or deployment of streaming technologies.

Chief information officer

A chief information officer (CIO) is the senior C-suite executive responsible for information technology within an organization; it is an internal-facing role. A CIO must possess deep knowledge of information technology and business processes—either alone is insufficient. CIOs direct the information technology organization, setting ongoing policies while also defining and executing significant initiatives under the direction of the CEO.

CIOs often collaborate with data engineering leadership in organizations with a well-developed data culture. If an organization is not very high in its data maturity, a CIO will typically help shape its data culture. CIOs will work with engineers and architects to map out major initiatives and make strategic decisions on adopting major architectural elements, such as enterprise resource planning (ERP) and customer relationship management (CRM) systems, cloud migrations, data systems, and internal-facing IT.

Chief technology officer

A chief technology officer (CTO) is similar to a CIO but faces outward. A CTO owns the key technological strategy and architectures for external-facing applications, such as mobile, web apps, and IoT—all critical data sources for data engineers. The CTO is likely a skilled technologist and has

a good sense of software engineering fundamentals and system architecture. In some organizations without a CIO, the CTO or sometimes the chief operating officer (COO) plays the role of CIO. Data engineers often report directly or indirectly through a CTO.

Chief data officer

The chief data officer (CDO) was created in 2002 at Capital One to recognize the growing importance of data as a business asset. The CDO is responsible for a company's data assets and strategy. CDOs are focused on data's business utility but should have a strong technical grounding. CDOs oversee data products, strategy, initiatives, and core functions such as master data management and privacy. Occasionally, CDOs manage business analytics and data engineering.

Chief analytics officer

The chief analytics officer (CAO) is a variant of the CDO role. Where both roles exist, the CDO focuses on the technology and organization required to deliver data. The CAO is responsible for analytics, strategy, and decision making for the business. A CAO may oversee data science and ML, though this largely depends on whether the company has a CDO or CTO role.

Chief algorithms officer

A chief algorithms officer (CAO-2) is a recent innovation in the C-suite, a highly technical role focused specifically on data science and ML. CAO-2s typically have experience as individual contributors and team leads in data science or ML projects. Frequently, they have a background in ML research and a related advanced degree.

CAO-2s are expected to be conversant in current ML research and have deep technical knowledge of their company's ML initiatives. In addition to creating business initiatives, they provide technical leadership, set research and development agendas, and build research teams.

Data engineers and project managers

Data engineers often work on significant initiatives, potentially spanning many years. As we write this book, many data engineers are working on cloud migrations, migrating pipelines and warehouses to the next generation of data tools. Other data engineers are starting greenfield projects, assembling new data architectures from scratch by selecting from an astonishing number of best-of-breed architecture and tooling options.

These large initiatives often benefit from *project management* (in contrast to product management, discussed next). Whereas data engineers function in an infrastructure and service delivery capacity, project managers direct traffic and serve as gatekeepers. Most project managers operate according to some variation of Agile and Scrum, with Waterfall still appearing occasionally. Business never sleeps, and business stakeholders often have a significant backlog of things they want to address and new initiatives they want to launch. Project managers must filter a long list of requests and prioritize critical deliverables to keep projects on track and better serve the company.

Data engineers interact with project managers, often planning sprints for projects and ensuing standups related to the sprint. Feedback goes both ways, with data engineers informing project managers and other stakeholders about progress and blockers, and project managers balancing the cadence of technology teams against the ever-changing needs of the business.

Data engineers and product managers

Product managers oversee product development, often owning product lines. In the context of data engineers, these products are called *data products*. Data products are either built from the ground up or are incremental improvements upon existing products. Data engineers interact more frequently with *product managers* as the corporate world has adopted a data-centric focus. Like project managers, product managers balance the activity of technology teams against the needs of the customer and business.

Data engineers and other management roles

Data engineers interact with various managers beyond project and product managers. However, these interactions usually follow either the services or cross-functional models. Data engineers either serve a variety of incoming requests as a centralized team or work as a resource assigned to a particular manager, project, or product.

For more information on data teams and how to structure them, we recommend John Thompson's *Building Analytics Teams* (Packt) and Jesse Anderson's *Data Teams* (Apress). Both books provide strong frameworks and perspectives on the roles of executives with data, who to hire, and how to construct the most effective data team for your company.

> **NOTE**
>
> Companies don't hire engineers simply to hack on code in isolation. To be worthy of their title, engineers should develop a deep understanding of the problems they're tasked with solving, the technology tools at their disposal, and the people they work with and serve.

Conclusion

This chapter provided you with a brief overview of the data engineering landscape, including the following:

- Defining data engineering and describing what data engineers do
- Describing the types of data maturity in a company
- Type A and type B data engineers
- Whom data engineers work with

We hope that this first chapter has whetted your appetite, whether you are a software development practitioner, data scientist, ML engineer, business stakeholder, entrepreneur, or venture capitalist. Of course, a great deal still remains to elucidate in subsequent chapters. Chapter 2 covers the data engineering lifecycle, followed by architecture in Chapter 3. The following

chapters get into the nitty-gritty of technology decisions for each part of the lifecycle. The entire data field is in flux, and as much as possible, each chapter focuses on the *immutables*—perspectives that will be valid for many years amid relentless change.

Additional Resources

- "On Complexity in Big Data" by Jesse Anderson (O'Reilly)
- "Which Profession Is More Complex to Become, a Data Engineer or a Data Scientist?" thread on Quora
- "The Future of Data Engineering Is the Convergence of Disciplines" by Liam Hausmann
- The Information Management Body of Knowledge website
- "Doing Data Science at Twitter" by Robert Chang
- "A Short History of Big Data" by Mark van Rijmenam
- "Data Engineering: A Quick and Simple Definition" by James Furbush (O'Reilly)
- "Big Data Will Be Dead in Five Years" by Lewis Gavin
- "The AI Hierarchy of Needs" by Monica Rogati
- Chapter 1 of*What Is Data Engineering?* by Lewis Gavin (O'Reilly)
- "The Three Levels of Data Analysis: A Framework for Assessing Data Organization Maturity" by Emilie Schario
- "Data as a Product vs. Data as a Service" by Justin Gage
- "The Downfall of the Data Engineer" by Maxime Beauchemin
- "The Rise of the Data Engineer" by Maxime Beauchemin
- "Skills of the Data Architect" by Bob Lambert

- "What Is a Data Architect? IT's Data Framework Visionary" by Thor Olavsrud
- "OpenAI's New Language Generator GPT-3 Is Shockingly Good—and Completely Mindless" by Will Douglas Heaven
- The AlphaGo research web page
- "How CEOs Can Lead a Data-Driven Culture" by Thomas H. Davenport and Nitin Mittal
- "Why CEOs Must Lead Big Data Initiatives" by John Weathington
- "How Creating a Data-Driven Culture Can Drive Success" by Frederik Bussler
- *Building Analytics Teams* by John K. Thompson (Packt)
- *Data Teams* by Jesse Anderson (Apress)
- "Information Management Body of Knowledge" Wikipedia page
- "Information management" Wikipedia page

---

1 "Data Engineering and Its Main Concepts," AlexSoft, last updated August 26, 2021, https://oreil.ly/e94py.

2 *ETL* stands for *extract, transform, load*, a common pattern we cover in the book.

3 Jesse Anderson, "The Two Types of Data Engineering," June 27, 2018, *https://oreil.ly/dxDt6*.

4 Maxime Beauchemin, "The Rise of the Data Engineer," January 20, 2017, *https://oreil.ly/kNDmd*.

5 Lewis Gavin, *What Is Data Engineering?* (Sebastapol, CA: O'Reilly, 2020), *https://oreil.ly/ELxLi*.

6 Cade Metz, "How Yahoo Spawned Hadoop, the Future of Big Data," *Wired*, October 18, 2011, *https://oreil.ly/iaD9G*.

7 Ron Miller, "How AWS Came to Be," *TechCrunch*, July 2, 2016, *https://oreil.ly/VJehv*.

8 *DataOps* is an abbreviation for *data operations*. We cover this topic in Chapter 2. For more information, read the DataOps Manifesto.

9 These acronyms stand for *California Consumer Privacy Act* and *General Data Protection Regulation*, respectively.

10 Robert Chang, "Doing Data Science at Twitter," *Medium*, June 20, 2015, *https://oreil.ly/xqjAx*.

11 Paramita (Guha) Ghosh, "Data Architect vs. Data Engineer," Dataversity, November 12, 2021, *https://oreil.ly/TlyZY*.

12 A variety of references exist for this notion. Although this cliche is widely known, a healthy debate has arisen around its validity in different practical settings. For more details, see Leigh Dodds, "Do Data Scientists Spend 80% of Their Time Cleaning Data? Turns Out, No?" Lost Boy blog, January 31, 2020, *https://oreil.ly/szFww*; and Alex Woodie, "Data Prep Still Dominates Data Scientists' Time, Survey Finds," *Datanami*, July 6, 2020, *https://oreil.ly/jDVWF*.

Chapter 2. The Data Engineering Lifecycle

The major goal of this book is to encourage you to move beyond viewing data engineering as a specific collection of data technologies. The data landscape is undergoing an explosion of new data technologies and practices, with ever-increasing levels of abstraction and ease of use. Because of increased technical abstraction, data engineers will increasingly become *data lifecycle engineers*, thinking and operating in terms of the *principles* of data lifecycle management.

In this chapter, you'll learn about the *data engineering lifecycle*, which is the central theme of this book. The data engineering lifecycle is our framework describing "cradle to grave" data engineering. You will also learn about the undercurrents of the data engineering lifecycle, which are key foundations that support all data engineering efforts.

What Is the Data Engineering Lifecycle?

The data engineering lifecycle comprises stages that turn raw data ingredients into a useful end product, ready for consumption by analysts, data scientists, ML engineers, and others. This chapter introduces the major stages of the data engineering lifecycle, focusing on each stage's core concepts and saving details for later chapters.

We divide the data engineering lifecycle into the following five stages (Figure 2-1, top):

- Generation
- Storage
- Ingestion

- Transformation
- Serving data

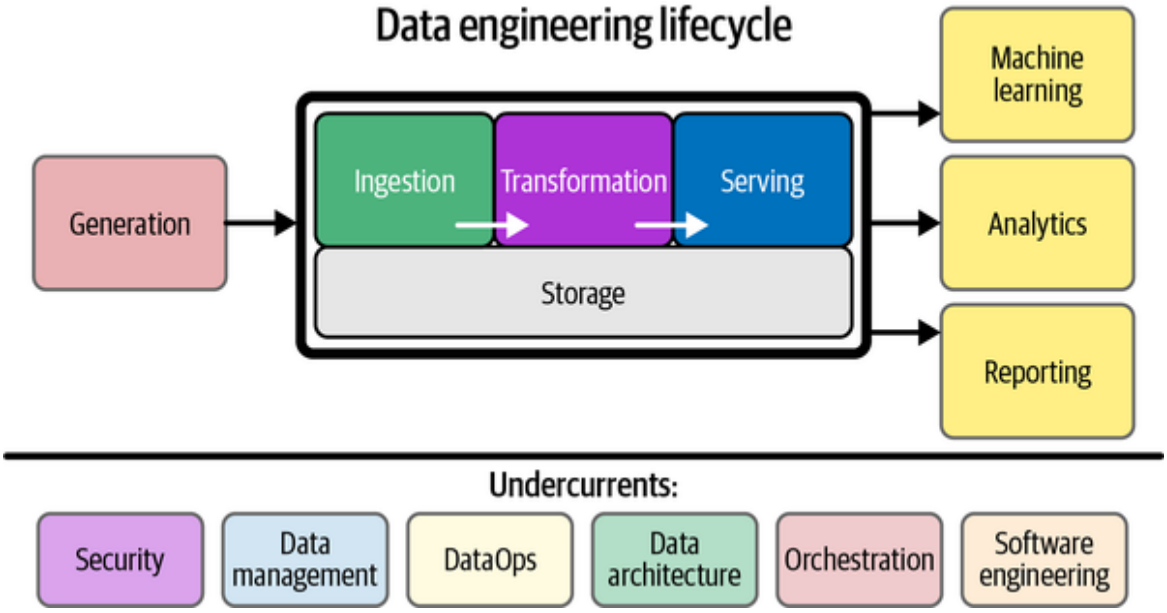


Figure 2-1. Components and undercurrents of the data engineering lifecycle

We begin the data engineering lifecycle by getting data from source systems and storing it. Next, we transform the data and then proceed to our central goal, serving data to analysts, data scientists, ML engineers, and others. In reality, storage occurs throughout the lifecycle as data flows from beginning to end—hence, the diagram shows the storage "stage" as a foundation that underpins other stages.

In general, the middle stages—storage, ingestion, transformation—can get a bit jumbled. And that's OK. Although we split out the distinct parts of the data engineering lifecycle, it's not always a neat, continuous flow. Various stages of the lifecycle may repeat themselves, occur out of order, overlap, or weave together in interesting and unexpected ways.

Acting as a bedrock are *undercurrents* (Figure 2-1, bottom) that cut across multiple stages of the data engineering lifecycle: security, data management, DataOps, data architecture, orchestration, and software engineering. No part of the data engineering lifecycle can adequately function without these undercurrents.

The Data Lifecycle Versus the Data Engineering Lifecycle

You may be wondering about the difference between the overall data lifecycle and the data engineering lifecycle. There's a subtle distinction between the two. The data engineering lifecycle is a subset of the whole data lifecycle (Figure 2-2). Whereas the full data lifecycle encompasses data across its entire lifespan, the data engineering lifecycle focuses on the stages a data engineer controls.

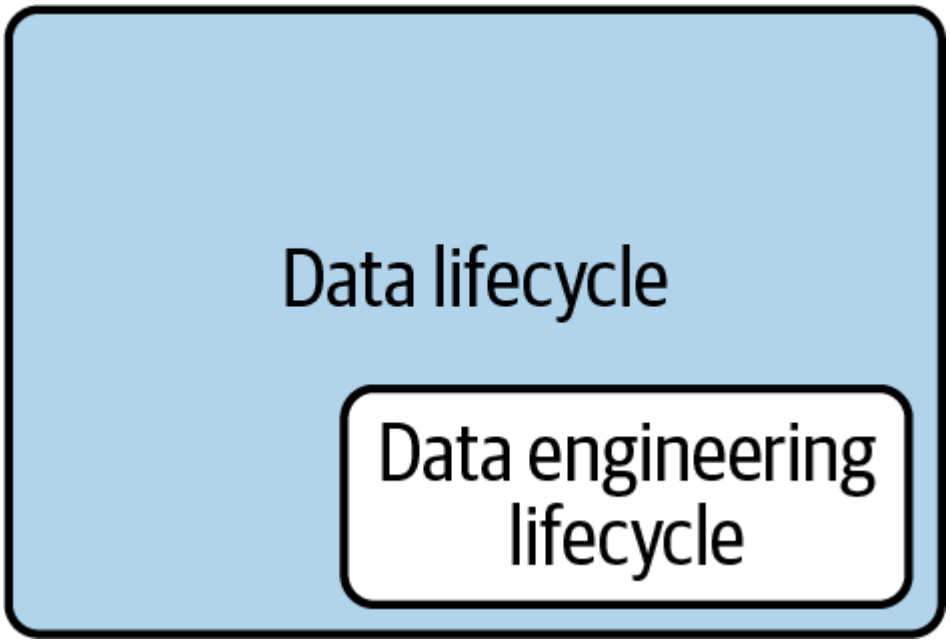


Figure 2-2. The data engineering lifecycle is a subset of the full data lifecycle

Generation: Source Systems

A *source system* is the origin of the data used in the data engineering lifecycle. For example, a source system could be an IoT device, an application message queue, or a transactional database. A data engineer consumes data from a source system, but doesn't typically own or control the source system itself. The data engineer needs to have a working understanding of the way source systems work, the way they generate data, the frequency and velocity of the data, and the variety of data they generate.

Engineers also need to keep an open line of communication with source system owners on changes that could break pipelines and analytics. Application code might change the structure of data in a field, or the application team might even choose to migrate the backend to an entirely new database technology.

A major challenge in data engineering is the dizzying array of data source systems engineers must work with and understand. As an illustration, let's look at two common source systems, one very traditional (an application database) and the other a more recent example (IoT swarms).

Figure 2-3 illustrates a traditional source system with several application servers supported by a database. This source system pattern became popular in the 1980s with the explosive success of relational database management systems (RDBMSs). The application + database pattern remains popular today with various modern evolutions of software development practices. For example, applications often consist of many small service/database pairs with microservices rather than a single monolith.

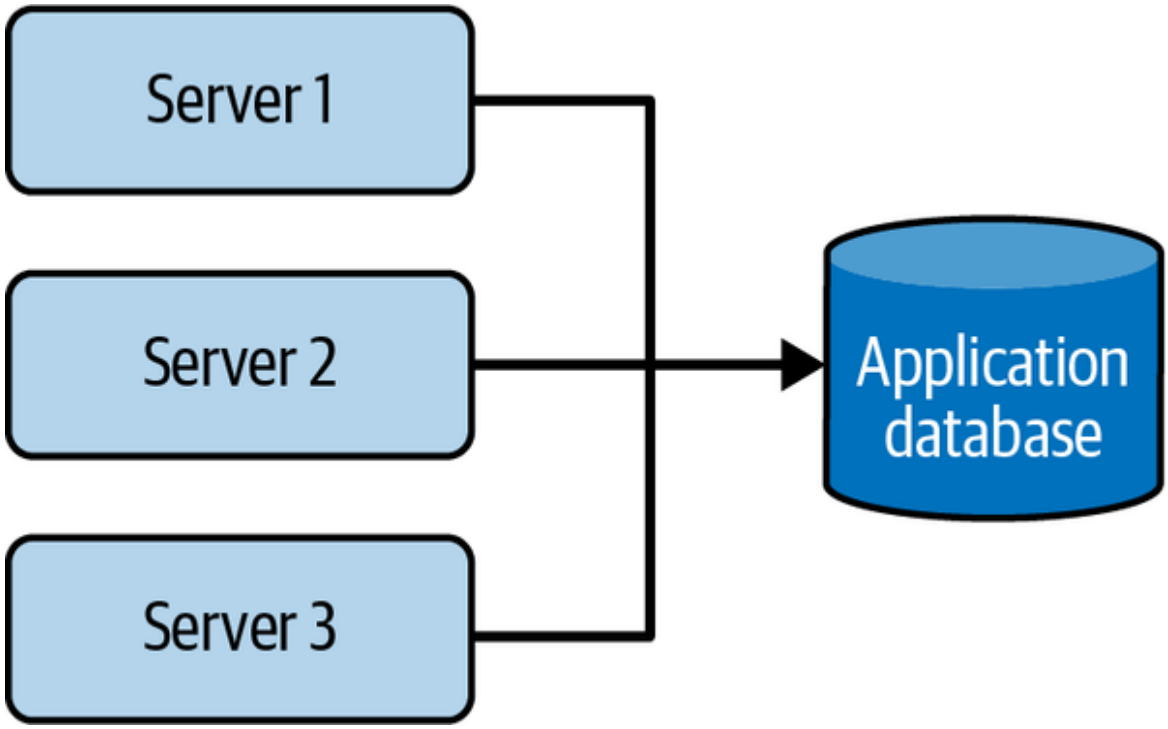


Figure 2-3. Source system example: an application database

Let's look at another example of a source system. Figure 2-4 illustrates an IoT swarm: a fleet of devices (circles) sends data messages (rectangles) to a central collection system. This IoT source system is increasingly common as IoT devices such as sensors, smart devices, and much more increase in the wild.

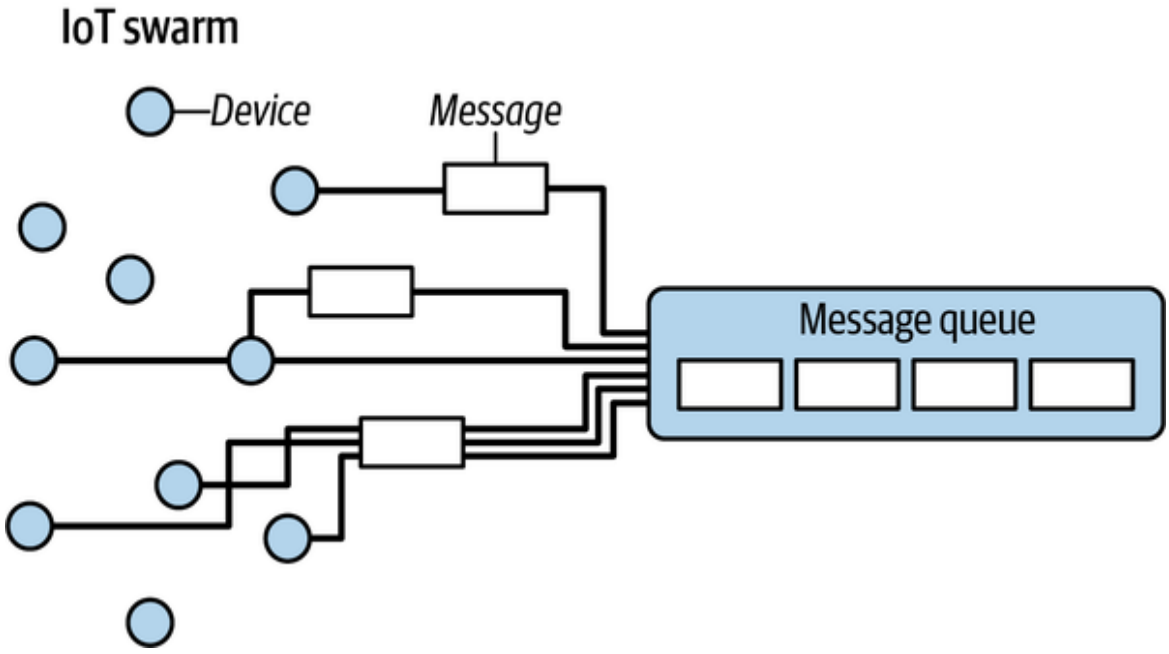


Figure 2-4. Source system example: an IoT swarm and messaging queue

Evaluating source systems: Key engineering considerations

There are many things to consider when assessing source systems, including how the system handles ingestion, state, and data generation. The following is a starting set of evaluation questions of source systems that data engineers must consider:

- What are the essential characteristics of the data source? Is it an application? A swarm of IoT devices?
- How is data persisted in the source system? Is data persisted long term, or is it temporary and quickly deleted?
- At what rate is data generated? How many events per second? How many gigabytes per hour?

- What level of consistency can data engineers expect from the output data? If you're running data-quality checks against the output data, how often do data inconsistencies occur—nulls where they aren't expected, lousy formatting, etc.?
- How often do errors occur?
- Will the data contain duplicates?
- Will some data values arrive late, possibly much later than other messages produced simultaneously?
- What is the schema of the ingested data? Will data engineers need to join across several tables or even several systems to get a complete picture of the data?
- If schema changes (say, a new column is added), how is this dealt with and communicated to downstream stakeholders?
- How frequently should data be pulled from the source system?
- For stateful systems (e.g., a database tracking customer account information), is data provided as periodic snapshots or update events from change data capture (CDC)? What's the logic for how changes are performed, and how are these tracked in the source database?
- Who/what is the data provider that will transmit the data for downstream consumption?
- Will reading from a data source impact its performance?
- Does the source system have upstream data dependencies? What are the characteristics of these upstream systems?
- Are data-quality checks in place to check for late or missing data?

Sources produce data consumed by downstream systems, including human-generated spreadsheets, IoT sensors, and web and mobile applications. Each source has its unique volume and cadence of data generation. A data engineer should know how the source generates data, including relevant

quirks or nuances. Data engineers also need to understand the limits of the source systems they interact with. For example, will analytical queries against a source application database cause resource contention and performance issues?

One of the most challenging nuances of source data is the schema. The *schema* defines the hierarchical organization of data. Logically, we can think of data at the level of a whole source system, drilling down into individual tables, all the way to the structure of respective fields. The schema of data shipped from source systems is handled in various ways. Two popular options are schemaless and fixed schema.

Schemaless doesn't mean the absence of schema. Rather, it means that the application defines the schema as data is written, whether to a messaging queue, a flat file, a blob, or a document database such as MongoDB. A more traditional model built on relational database storage uses a *fixed schema* enforced in the database, to which application writes must conform.

Either of these models presents challenges for data engineers. Schemas change over time; in fact, schema evolution is encouraged in the Agile approach to software development. A key part of the data engineer's job is taking raw data input in the source system schema and transforming this into valuable output for analytics. This job becomes more challenging as the source schema evolves.

We dive into source systems in greater detail in Chapter 5; we also cover schemas and data modeling in Chapters 6 and 8, respectively.

Storage

After ingesting data, you need a place to store it. Choosing a storage solution is key to success in the rest of the data lifecycle, and it's also one of the most complicated stages of the data lifecycle for a variety of reasons. First, data architectures in the cloud often leverage *several* storage solutions. Second, few data storage solutions function purely as storage, with many supporting complex transformation queries; even object storage solutions may support powerful query capabilities—e.g., Amazon S3

Select. Third, while storage is a stage of the data engineering lifecycle, it frequently touches on other stages, such as ingestion, transformation, and serving.

Storage runs across the entire data engineering lifecycle, often occurring in multiple places in a data pipeline, with storage systems crossing over with source systems, ingestion, transformation, and serving. In many ways, the way data is stored impacts how it is used in all of the stages of the data engineering lifecycle. For example, cloud data warehouses can store data, process data in pipelines, and serve it to analysts. Streaming frameworks such as Apache Kafka and Pulsar can function simultaneously as ingestion, storage, and query systems for messages, with object storage being a standard layer for data transmission.

Evaluating storage systems: Key engineering considerations

Here are a few key engineering questions to ask when choosing a storage system for a data warehouse, data lakehouse, database, or object storage:

- Is this storage solution compatible with the architecture's required write and read speeds?
- Will storage create a bottleneck for downstream processes?
- Do you understand how this storage technology works? Are you utilizing the storage system optimally or committing unnatural acts? For instance, are you applying a high rate of random access updates in an object storage system? (This is an antipattern with significant performance overhead.)
- Will this storage system handle anticipated future scale? You should consider all capacity limits on the storage system: total available storage, read operation rate, write volume, etc.
- Will downstream users and processes be able to retrieve data in the required service-level agreement (SLA)?

- Are you capturing metadata about schema evolution, data flows, data lineage, and so forth? Metadata has a significant impact on the utility of data. Metadata represents an investment in the future, dramatically enhancing discoverability and institutional knowledge to streamline future projects and architecture changes.
- Is this a pure storage solution (object storage), or does it support complex query patterns (i.e., a cloud data warehouse)?
- Is the storage system schema-agnostic (object storage)? Flexible schema (Cassandra)? Enforced schema (a cloud data warehouse)?
- How are you tracking master data, golden records data quality, and data lineage for data governance? (We have more to say on these in "Data Management".)
- How are you handling regulatory compliance and data sovereignty? For example, can you store your data in certain geographical locations but not others?

Understanding data access frequency

Not all data is accessed in the same way. Retrieval patterns will greatly vary based on the data being stored and queried. This brings up the notion of the "temperatures" of data. Data access frequency will determine the temperature of your data.

Data that is most frequently accessed is called *hot data*. Hot data is commonly retrieved many times per day, perhaps even several times per second, in systems that serve user requests. This data should be stored for fast retrieval, where "fast" is relative to the use case. *Lukewarm data* might be accessed every so often—say, every week or month.

Cold data is seldom queried and is appropriate for storing in an archival system. Cold data is often retained for compliance purposes or in case of a catastrophic failure in another system. In the "old days," cold data would be stored on tapes and shipped to remote archival facilities. In cloud

environments, vendors offer specialized storage tiers with very cheap monthly storage costs but high prices for data retrieval.

Selecting a storage system

What type of storage solution should you use? This depends on your use cases, data volumes, frequency of ingestion, format, and size of the data being ingested—essentially, the key considerations listed in the preceding bulleted questions. There is no one-size-fits-all universal storage recommendation. Every storage technology has its trade-offs. Countless varieties of storage technologies exist, and it's easy to be overwhelmed when deciding the best option for your data architecture.

Chapter 6 covers storage best practices and approaches in greater detail, as well as the crossover between storage and other lifecycle stages.

Ingestion

After you understand the data source and the characteristics of the source system you're using, you need to gather the data. The second stage of the data engineering lifecycle is data ingestion from source systems.

In our experience, source systems and ingestion represent the most significant bottlenecks of the data engineering lifecycle. The source systems are normally outside your direct control and might randomly become unresponsive or provide data of poor quality. Or, your data ingestion service might mysteriously stop working for many reasons. As a result, data flow stops or delivers insufficient data for storage, processing, and serving.

Unreliable source and ingestion systems have a ripple effect across the data engineering lifecycle. But you're in good shape, assuming you've answered the big questions about source systems.

Key engineering considerations for the ingestion phase

When preparing to architect or build a system, here are some primary questions about the ingestion stage:

- What are the use cases for the data I'm ingesting? Can I reuse this data rather than create multiple versions of the same dataset?
- Are the systems generating and ingesting this data reliably, and is the data available when I need it?
- What is the data destination after ingestion?
- How frequently will I need to access the data?
- In what volume will the data typically arrive?
- What format is the data in? Can my downstream storage and transformation systems handle this format?
- Is the source data in good shape for immediate downstream use? If so, for how long, and what may cause it to be unusable?
- If the data is from a streaming source, does it need to be transformed before reaching its destination? Would an in-flight transformation be appropriate, where the data is transformed within the stream itself?

These are just a sample of the factors you'll need to think about with ingestion, and we cover those questions and more in Chapter 7. Before we leave, let's briefly turn our attention to two major data ingestion concepts: batch versus streaming and push versus pull.

Batch versus streaming

Virtually all data we deal with is inherently *streaming*. Data is nearly always produced and updated continually at its source. *Batch ingestion* is simply a specialized and convenient way of processing this stream in large chunks—for example, handling a full day's worth of data in a single batch.

Streaming ingestion allows us to provide data to downstream systems—whether other applications, databases, or analytics systems—in a continuous, real-time fashion. Here, *real-time* (or *near real-time*) means that the data is available to a downstream system a short time after it is

produced (e.g., less than one second later). The latency required to qualify as real-time varies by domain and requirements.

Batch data is ingested either on a predetermined time interval or as data reaches a preset size threshold. Batch ingestion is a one-way door: once data is broken into batches, the latency for downstream consumers is inherently constrained. Because of limitations of legacy systems, batch was for a long time the default way to ingest data. Batch processing remains an extremely popular way to ingest data for downstream consumption, particularly in analytics and ML.

However, the separation of storage and compute in many systems and the ubiquity of event-streaming and processing platforms make the continuous processing of data streams much more accessible and increasingly popular. The choice largely depends on the use case and expectations for data timeliness.

Key considerations for batch versus stream ingestion

Should you go streaming-first? Despite the attractiveness of a streaming-first approach, there are many trade-offs to understand and think about. The following are some questions to ask yourself when determining whether streaming ingestion is an appropriate choice over batch ingestion:

- If I ingest the data in real time, can downstream storage systems handle the rate of data flow?
- Do I need millisecond real-time data ingestion? Or would a micro-batch approach work, accumulating and ingesting data, say, every minute?
- What are my use cases for streaming ingestion? What specific benefits do I realize by implementing streaming? If I get data in real time, what actions can I take on that data that would be an improvement upon batch?
- Will my streaming-first approach cost more in terms of time, money, maintenance, downtime, and opportunity cost than simply doing

batch?

- Are my streaming pipeline and system reliable and redundant if infrastructure fails?
- What tools are most appropriate for the use case? Should I use a managed service (Amazon Kinesis, Google Cloud Pub/Sub, Google Cloud Dataflow) or stand up my own instances of Kafka, Flink, Spark, Pulsar, etc.? If I do the latter, who will manage it? What are the costs and trade-offs?
- If I'm deploying an ML model, what benefits do I have with online predictions and possibly continuous training?
- Am I getting data from a live production instance? If so, what's the impact of my ingestion process on this source system?

As you can see, streaming-first might seem like a good idea, but it's not always straightforward; extra costs and complexities inherently occur. Many great ingestion frameworks do handle both batch and micro-batch ingestion styles. We think batch is an excellent approach for many common use cases, such as model training and weekly reporting. Adopt true real-time streaming only after identifying a business use case that justifies the trade-offs against using batch.

Push versus pull

In the *push* model of data ingestion, a source system writes data out to a target, whether a database, object store, or filesystem. In the *pull* model, data is retrieved from the source system. The line between the push and pull paradigms can be quite blurry; data is often pushed and pulled as it works its way through the various stages of a data pipeline.

Consider, for example, the extract, transform, load (ETL) process, commonly used in batch-oriented ingestion workflows. ETL's *extract* (*E*) part clarifies that we're dealing with a pull ingestion model. In traditional ETL, the ingestion system queries a current source table snapshot on a fixed

schedule. You'll learn more about ETL and extract, load, transform (ELT) throughout this book.

In another example, consider continuous CDC, which is achieved in a few ways. One common method triggers a message every time a row is changed in the source database. This message is *pushed* to a queue, where the ingestion system picks it up. Another common CDC method uses binary logs, which record every commit to the database. The database *pushes* to its logs. The ingestion system reads the logs but doesn't directly interact with the database otherwise. This adds little to no additional load to the source database. Some versions of batch CDC use the *pull* pattern. For example, in timestamp-based CDC, an ingestion system queries the source database and pulls the rows that have changed since the previous update.

With streaming ingestion, data bypasses a backend database and is pushed directly to an endpoint, typically with data buffered by an event-streaming platform. This pattern is useful with fleets of IoT sensors emitting sensor data. Rather than relying on a database to maintain the current state, we simply think of each recorded reading as an event. This pattern is also growing in popularity in software applications as it simplifies real-time processing, allows app developers to tailor their messages for downstream analytics, and greatly simplifies the lives of data engineers.

We discuss ingestion best practices and techniques in depth in Chapter 7. Next, let's turn to the transformation stage of the data engineering lifecycle.

Transformation

After you've ingested and stored data, you need to do something with it. The next stage of the data engineering lifecycle is *transformation*, meaning data needs to be changed from its original form into something useful for downstream use cases. Without proper transformations, data will sit inert, and not be in a useful form for reports, analysis, or ML. Typically, the transformation stage is where data begins to create value for downstream user consumption.

Immediately after ingestion, basic transformations map data into correct types (changing ingested string data into numeric and date types, for example), putting records into standard formats, and removing bad ones. Later stages of transformation may transform the data schema and apply normalization. Downstream, we can apply large-scale aggregation for reporting or featurize data for ML processes.

Key considerations for the transformation phase

When considering data transformations within the data engineering lifecycle, it helps to consider the following:

- What's the cost and return on investment (ROI) of the transformation? What is the associated business value?
- Is the transformation as simple and self-isolated as possible?
- What business rules do the transformations support?
- Am I minimizing data movement between the transformation and the storage system during transformation?

You can transform data in batch or while streaming in flight. As mentioned in "Ingestion", virtually all data starts life as a continuous stream; batch is just a specialized way of processing a data stream. Batch transformations are overwhelmingly popular, but given the growing popularity of stream-processing solutions and the general increase in the amount of streaming data, we expect the popularity of streaming transformations to continue growing, perhaps entirely replacing batch processing in certain domains soon.

Logically, we treat transformation as a standalone area of the data engineering lifecycle, but the realities of the lifecycle can be much more complicated in practice. Transformation is often entangled in other phases of the lifecycle. Typically, data is transformed in source systems or in flight during ingestion. For example, a source system may add an event timestamp to a record before forwarding it to an ingestion process. Or a record within a streaming pipeline may be "enriched" with additional fields

and calculations before it's sent to a data warehouse. Transformations are ubiquitous in various parts of the lifecycle. Data preparation, data wrangling, and cleaning—these transformative tasks add value for end consumers of data.

Business logic is a major driver of data transformation, often in data modeling. Data translates business logic into reusable elements (e.g., a sale means "somebody bought 12 picture frames from me for $30 each, or $360 in total"). In this case, somebody bought 12 picture frames for $30 each. Data modeling is critical for obtaining a clear and current picture of business processes. A simple view of raw retail transactions might not be useful without adding the logic of accounting rules so that the CFO has a clear picture of financial health. Ensure a standard approach for implementing business logic across your transformations.

Data featurization for ML is another data transformation process. Featurization intends to extract and enhance data features useful for training ML models. Featurization can be a dark art, combining domain expertise (to identify which features might be important for prediction) with extensive experience in data science. For this book, the main point is that once data scientists determine how to featurize data, featurization processes can be automated by data engineers in the transformation stage of a data pipeline.

Transformation is a profound subject, and we cannot do it justice in this brief introduction. Chapter 8 delves into queries, data modeling, and various transformation practices and nuances.

Serving Data

You've reached the last stage of the data engineering lifecycle. Now that the data has been ingested, stored, and transformed into coherent and useful structures, it's time to get value from your data. "Getting value" from data means different things to different users.

Data has *value* when it's used for practical purposes. Data that is not consumed or queried is simply inert. Data vanity projects are a major risk for companies. Many companies pursued vanity projects in the big data era,

gathering massive datasets in data lakes that were never consumed in any useful way. The cloud era is triggering a new wave of vanity projects built on the latest data warehouses, object storage systems, and streaming technologies. Data projects must be intentional across the lifecycle. What is the ultimate business purpose of the data so carefully collected, cleaned, and stored?

Data serving is perhaps the most exciting part of the data engineering lifecycle. This is where the magic happens. This is where ML engineers can apply the most advanced techniques. Let's look at some of the popular uses of data: analytics, ML, and reverse ETL.

Analytics

Analytics is the core of most data endeavors. Once your data is stored and transformed, you're ready to generate reports or dashboards, and do ad hoc analysis on the data. Whereas the bulk of analytics used to encompass BI, it now includes other facets such as operational analytics and customer-facing analytics (Figure 2-5). Let's briefly touch on these variations of analytics.

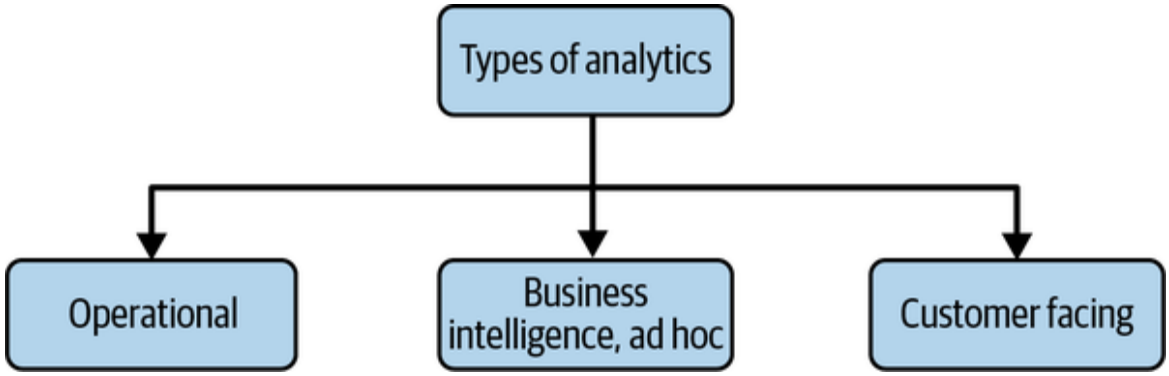


Figure 2-5. Types of analytics

Business intelligence

BI marshals collected data to describe a business's past and current state. BI requires using business logic to process raw data. Note that data serving for analytics is yet another area where the stages of the data engineering lifecycle can get tangled. As we mentioned earlier, business logic is often applied to data in the transformation stage of the data engineering lifecycle, but a logic-on-read approach has become increasingly popular. Data is

stored in a clean but fairly raw form, with minimal postprocessing business logic. A BI system maintains a repository of business logic and definitions. This business logic is used to query the data warehouse so that reports and dashboards align with business definitions.

As a company grows its data maturity, it will move from ad hoc data analysis to self-service analytics, allowing democratized data access to business users without needing IT to intervene. The capability to do self-service analytics assumes that data is good enough that people across the organization can simply access it themselves, slice and dice it however they choose, and get immediate insights. Although self-service analytics is simple in theory, it's tough to pull off in practice. The main reason is that poor data quality, organizational silos, and a lack of adequate data skills get in the way of allowing widespread use of analytics.

Operational analytics

Operational analytics focuses on the fine-grained details of operations, promoting actions that a user of the reports can act upon immediately. Operational analytics could be a live view of inventory or real-time dashboarding of website health. In this case, data is consumed in real time, either directly from a source system or from a streaming data pipeline. The types of insights in operational analytics differ from traditional BI since operational analytics is focused on the present and doesn't necessarily concern historical trends.

Embedded analytics

You may wonder why we've broken out embedded analytics (customer-facing analytics) separately from BI. In practice, analytics provided to customers on a SaaS platform come with a separate set of requirements and complications. Internal BI faces a limited audience and generally presents a limited number of unified views. Access controls are critical but not particularly complicated. Access is managed using a handful of roles and access tiers.

With customer-facing analytics, the request rate for reports, and the corresponding burden on analytics systems, go up dramatically; access control is significantly more complicated and critical. Businesses may be serving separate analytics and data to thousands or more customers. Each customer must see their data and only their data. An internal data-access error at a company would likely lead to a procedural review. A data leak between customers would be considered a massive breach of trust, leading to media attention and a significant loss of customers. Minimize your blast radius related to data leaks and security vulnerabilities. Apply tenant- or data-level security within your storage, and anywhere there's a possibility of data leakage.

MULTITENANCY

Many current storage and analytics systems support multitenancy in various ways. Data engineers may choose to house data for many customers in common tables to allow a unified view for internal analytics and ML. This data is presented externally to individual customers through logical views with appropriately defined controls and filters. It is incumbent on data engineers to understand the minutiae of multitenancy in the systems they deploy to ensure absolute data security and isolation.

Machine learning

The emergence and success of ML is one of the most exciting technology revolutions. Once organizations reach a high level of data maturity, they can begin to identify problems amenable to ML and start organizing a practice around it.

The responsibilities of data engineers overlap significantly in analytics and ML, and the boundaries between data engineering, ML engineering, and analytics engineering can be fuzzy. For example, a data engineer may need to support Spark clusters that facilitate analytics pipelines and ML model training. They may also need to provide a system that orchestrates tasks

across teams and support metadata and cataloging systems that track data history and lineage. Setting these domains of responsibility and the relevant reporting structures is a critical organizational decision.

The feature store is a recently developed tool that combines data engineering and ML engineering. Feature stores are designed to reduce the operational burden for ML engineers by maintaining feature history and versions, supporting feature sharing among teams, and providing basic operational and orchestration capabilities, such as backfilling. In practice, data engineers are part of the core support team for feature stores to support ML engineering.

Should a data engineer be familiar with ML? It certainly helps. Regardless of the operational boundary between data engineering, ML engineering, business analytics, and so forth, data engineers should maintain operational knowledge about their teams. A good data engineer is conversant in the fundamental ML techniques and related data-processing requirements, the use cases for models within their company, and the responsibilities of the organization's various analytics teams. This helps maintain efficient communication and facilitate collaboration. Ideally, data engineers will build tools in partnership with other teams that neither team can make independently.

This book cannot possibly cover ML in depth. A growing ecosystem of books, videos, articles, and communities is available if you're interested in learning more; we include a few suggestions in "Additional Resources".

The following are some considerations for the serving data phase specific to ML:

- Is the data of sufficient quality to perform reliable feature engineering? Quality requirements and assessments are developed in close collaboration with teams consuming the data.
- Is the data discoverable? Can data scientists and ML engineers easily find valuable data?

- Where are the technical and organizational boundaries between data engineering and ML engineering? This organizational question has significant architectural implications.
- Does the dataset properly represent ground truth? Is it unfairly biased?

While ML is exciting, our experience is that companies often prematurely dive into it. Before investing a ton of resources into ML, take the time to build a solid data foundation. This means setting up the best systems and architecture across the data engineering and ML lifecycle. It's generally best to develop competence in analytics before moving to ML. Many companies have dashed their ML dreams because they undertook initiatives without appropriate foundations.

Reverse ETL

Reverse ETL has long been a practical reality in data, viewed as an antipattern that we didn't like to talk about or dignify with a name. *Reverse ETL* takes processed data from the output side of the data engineering lifecycle and feeds it back into source systems, as shown in Figure 2-6. In reality, this flow is beneficial and often necessary; reverse ETL allows us to take analytics, scored models, etc., and feed these back into production systems or SaaS platforms.

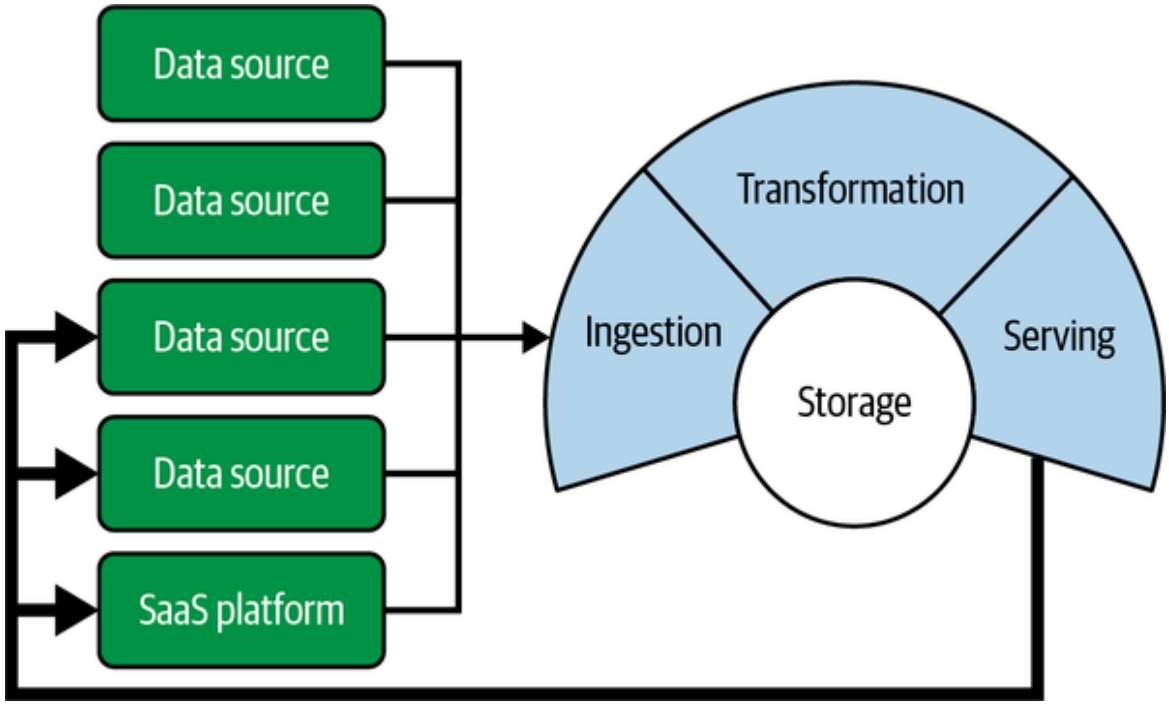


Figure 2-6. Reverse ETL

Marketing analysts might calculate bids in Microsoft Excel by using the data in their data warehouse, and then upload these bids to Google Ads. This process was often entirely manual and primitive.

As we've written this book, several vendors have embraced the concept of reverse ETL and built products around it, such as Hightouch and Census. Reverse ETL remains nascent as a field, but we suspect that it is here to stay.

Reverse ETL has become especially important as businesses rely increasingly on SaaS and external platforms. For example, companies may want to push specific metrics from their data warehouse to a customer data platform or CRM system. Advertising platforms are another everyday use case, as in the Google Ads example. Expect to see more activity in reverse ETL, with an overlap in both data engineering and ML engineering.

The jury is out on whether the term *reverse ETL* will stick. And the practice may evolve. Some engineers claim that we can eliminate reverse ETL by handling data transformations in an event stream and sending those events back to source systems as needed. Realizing widespread adoption of this

pattern across businesses is another matter. The gist is that transformed data will need to be returned to source systems in some manner, ideally with the correct lineage and business process associated with the source system.

Major Undercurrents Across the Data Engineering Lifecycle

Data engineering is rapidly maturing. Whereas prior cycles of data engineering simply focused on the technology layer, the continued abstraction and simplification of tools and practices have shifted this focus. Data engineering now encompasses far more than tools and technology. The field is now moving up the value chain, incorporating traditional enterprise practices such as data management and cost optimization, and newer practices like DataOps.

We've termed these practices *undercurrents*—security, data management, DataOps, data architecture, orchestration, and software engineering—that support every aspect of the data engineering lifecycle (Figure 2-7). In this section, we give a brief overview of these undercurrents and their major components, which you'll see in more detail throughout the book.

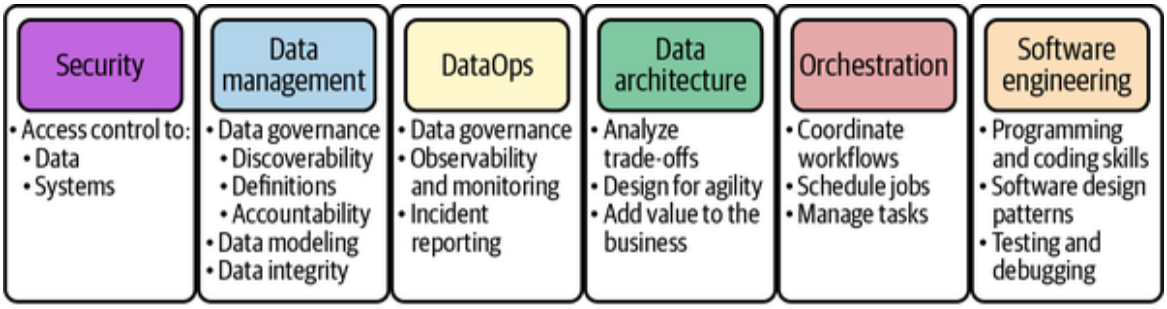


Figure 2-7. The major undercurrents of data engineering

Security

Security must be top of mind for data engineers, and those who ignore it do so at their peril. That's why security is the first undercurrent. Data engineers must understand both data and access security, exercising the principle of

least privilege. The principle of least privilege means giving a user or system access to only the essential data and resources to perform an intended function. A common antipattern we see with data engineers with little security experience is to give admin access to all users. This is a catastrophe waiting to happen!

Give users only the access they need to do their jobs today, nothing more. Don't operate from a root shell when you're just looking for visible files with standard user access. When querying tables with a lesser role, don't use the superuser role in a database. Imposing the principle of least privilege on ourselves can prevent accidental damage and keep you in a security-first mindset.

People and organizational structure are always the biggest security vulnerabilities in any company. When we hear about major security breaches in the media, it often turns out that someone in the company ignored basic precautions, fell victim to a phishing attack, or otherwise acted irresponsibly. The first line of defense for data security is to create a culture of security that permeates the organization. All individuals who have access to data must understand their responsibility in protecting the company's sensitive data and its customers.

Data security is also about timing—providing data access to exactly the people and systems that need to access it and *only for the duration necessary to perform their work*. Data should be protected from unwanted visibility, both in flight and at rest, by using encryption, tokenization, data masking, obfuscation, and simple, robust access controls.

Data engineers must be competent security administrators, as security falls in their domain. A data engineer should understand security best practices for the cloud and on prem. Knowledge of user and identity access management (IAM) roles, policies, groups, network security, password policies, and encryption are good places to start.

Throughout the book, we highlight areas where security should be top of mind in the data engineering lifecycle. You can also gain more detailed insights into security in Chapter 10.

Data Management

You probably think that data management sounds very…corporate. "Old school" data management practices make their way into data and ML engineering. What's old is new again. Data management has been around for decades but didn't get a lot of traction in data engineering until recently. Data tools are becoming simpler, and there is less complexity for data engineers to manage. As a result, the data engineer moves up the value chain toward the next rung of best practices. Data best practices once reserved for huge companies—data governance, master data management, data-quality management, metadata management—are now filtering down to companies of all sizes and maturity levels. As we like to say, data engineering is becoming "enterprisey." This is ultimately a great thing!

The Data Management Association International (DAMA) *Data Management Body of Knowledge* (*DMBOK*), which we consider to be the definitive book for enterprise data management, offers this definition:

> *Data management is the development, execution, and supervision of plans, policies, programs, and practices that deliver, control, protect, and enhance the value of data and information assets throughout their lifecycle.*

That's a bit lengthy, so let's look at how it ties to data engineering. Data engineers manage the data lifecycle, and data management encompasses the set of best practices that data engineers will use to accomplish this task, both technically and strategically. Without a framework for managing data, data engineers are simply technicians operating in a vacuum. Data engineers need a broader perspective of data's utility across the organization, from the source systems to the C-suite, and everywhere in between.

Why is data management important? Data management demonstrates that data is vital to daily operations, just as businesses view financial resources, finished goods, or real estate as assets. Data management practices form a cohesive framework that everyone can adopt to ensure that the organization gets value from data and handles it appropriately.

Data management has quite a few facets, including the following:

- Data governance, including discoverability and accountability
- Data modeling and design
- Data lineage
- Storage and operations
- Data integration and interoperability
- Data lifecycle management
- Data systems for advanced analytics and ML
- Ethics and privacy

While this book is in no way an exhaustive resource on data management, let's briefly cover some salient points from each area as they relate to data engineering.

Data governance

According to *Data Governance: The Definitive Guide,* "Data governance is, first and foremost, a data management function to ensure the quality, integrity, security, and usability of the data collected by an organization."[1]

We can expand on that definition and say that data governance engages people, processes, and technologies to maximize data value across an organization while protecting data with appropriate security controls. Effective data governance is developed with intention and supported by the organization. When data governance is accidental and haphazard, the side effects can range from untrusted data to security breaches and everything in between. Being intentional about data governance will maximize the organization's data capabilities and the value generated from data. It will also (hopefully) keep a company out of the headlines for questionable or downright reckless data practices.

Think of the typical example of data governance being done poorly. A business analyst gets a request for a report but doesn't know what data to use to answer the question. They may spend hours digging through dozens of tables in a transactional database, wildly guessing at which fields might be useful. The analyst compiles a "directionally correct" report but isn't entirely sure that the report's underlying data is accurate or sound. The recipient of the report also questions the validity of the data. The integrity of the analyst—and of all data in the company's systems—is called into question. The company is confused about its performance, making business planning impossible.

Data governance is a foundation for data-driven business practices and a mission-critical part of the data engineering lifecycle. When data governance is practiced well, people, processes, and technologies align to treat data as a key business driver; if data issues occur, they are promptly handled.

The core categories of data governance are discoverability, security, and accountability.[2] Within these core categories are subcategories, such as data quality, metadata, and privacy. Let's look at each core category in turn.

Discoverability

In a data-driven company, data must be available and discoverable. End users should have quick and reliable access to the data they need to do their jobs. They should know where the data comes from, how it relates to other data, and what the data means.

Some key areas of data discoverability include metadata management and master data management. Let's briefly describe these areas.

Metadata

Metadata is "data about data," and it underpins every section of the data engineering lifecycle. Metadata is exactly the data needed to make data discoverable and governable.

We divide metadata into two major categories: autogenerated and human generated. Modern data engineering revolves around automation, but

metadata collection is often manual and error prone.

Technology can assist with this process, removing much of the error-prone work of manual metadata collection. We're seeing a proliferation of data catalogs, data-lineage tracking systems, and metadata management tools. Tools can crawl databases to look for relationships and monitor data pipelines to track where data comes from and where it goes. A low-fidelity manual approach uses an internally led effort where various stakeholders crowdsource metadata collection within the organization. These data management tools are covered in depth throughout the book, as they undercut much of the data engineering lifecycle.

Metadata becomes a byproduct of data and data processes. However, key challenges remain. In particular, interoperability and standards are still lacking. Metadata tools are only as good as their connectors to data systems and their ability to share metadata. In addition, automated metadata tools should not entirely take humans out of the loop.

Data has a social element; each organization accumulates social capital and knowledge around processes, datasets, and pipelines. Human-oriented metadata systems focus on the social aspect of metadata. This is something that Airbnb has emphasized in its various blog posts on data tools, particularly its original Dataportal concept.[3] Such tools should provide a place to disclose data owners, data consumers, and domain experts. Documentation and internal wiki tools provide a key foundation for metadata management, but these tools should also integrate with automated data cataloging. For example, data-scanning tools can generate wiki pages with links to relevant data objects.

Once metadata systems and processes exist, data engineers can consume metadata in useful ways. Metadata becomes a foundation for designing pipelines and managing data throughout the lifecycle.

DMBOK identifies four main categories of metadata that are useful to data engineers:

- Business metadata

- Technical metadata
- Operational metadata
- Reference metadata

Let's briefly describe each category of metadata.

Business metadata relates to the way data is used in the business, including business and data definitions, data rules and logic, how and where data is used, and the data owner(s).

A data engineer uses business metadata to answer nontechnical questions about who, what, where, and how. For example, a data engineer may be tasked with creating a data pipeline for customer sales analysis. But what is a customer? Is it someone who's purchased in the last 90 days? Or someone who's purchased at any time the business has been open? A data engineer would use the correct data to refer to business metadata (data dictionary or data catalog) to look up how a "customer" is defined. Business metadata provides a data engineer with the right context and definitions to properly use data.

Technical metadata describes the data created and used by systems across the data engineering lifecycle. It includes the data model and schema, data lineage, field mappings, and pipeline workflows. A data engineer uses technical metadata to create, connect, and monitor various systems across the data engineering lifecycle.

Here are some common types of technical metadata that a data engineer will use:

- Pipeline metadata (often produced in orchestration systems)
- Data lineage
- Schema

Orchestration is a central hub that coordinates workflow across various systems. *Pipeline metadata* captured in orchestration systems provides

details of the workflow schedule, system and data dependencies, configurations, connection details, and much more.

Data-lineage metadata tracks the origin and changes to data, and its dependencies, over time. As data flows through the data engineering lifecycle, it evolves through transformations and combinations with other data. Data lineage provides an audit trail of data's evolution as it moves through various systems and workflows.

Schema metadata describes the structure of data stored in a system such as a database, a data warehouse, a data lake, or a filesystem; it is one of the key differentiators across different storage systems. Object stores, for example, don't manage schema metadata; instead, this must be managed in a *metastore*. On the other hand, cloud data warehouses manage schema metadata internally.

These are just a few examples of technical metadata that a data engineer should know about. This is not a complete list, and we cover additional aspects of technical metadata throughout the book.

Operational metadata describes the operational results of various systems and includes statistics about processes, job IDs, application runtime logs, data used in a process, and error logs. A data engineer uses operational metadata to determine whether a process succeeded or failed and the data involved in the process.

Orchestration systems can provide a limited picture of operational metadata, but the latter still tends to be scattered across many systems. A need for better-quality operational metadata, and better metadata management, is a major motivation for next-generation orchestration and metadata management systems.

Reference metadata is data used to classify other data. This is also referred to as *lookup data*. Standard examples of reference data are internal codes, geographic codes, units of measurement, and internal calendar standards. Note that much of reference data is fully managed internally, but items such as geographic codes might come from standard external references.

Reference data is essentially a standard for interpreting other data, so if it changes, this change happens slowly over time.

Data accountability

Data accountability means assigning an individual to govern a portion of data. The responsible person then coordinates the governance activities of other stakeholders. Managing data quality is tough if no one is accountable for the data in question.

Note that people accountable for data need not be data engineers. The accountable person might be a software engineer or product manager, or serve in another role. In addition, the responsible person generally doesn't have all the resources necessary to maintain data quality. Instead, they coordinate with all people who touch the data, including data engineers.

Data accountability can happen at various levels; accountability can happen at the level of a table or a log stream but could be as fine-grained as a single field entity that occurs across many tables. An individual may be accountable for managing a customer ID across many systems. For enterprise data management, a data domain is the set of all possible values that can occur for a given field type, such as in this ID example. This may seem excessively bureaucratic and meticulous, but it can significantly affect data quality.

Data quality

> *Can I trust this data?*
>
> —Everyone in the business

Data quality is the optimization of data toward the desired state and orbits the question, "What do you get compared with what you expect?" Data should conform to the expectations in the business metadata. Does the data match the definition agreed upon by the business?

A data engineer ensures data quality across the entire data engineering lifecycle. This involves performing data-quality tests, and ensuring data conformance to schema expectations, data completeness, and precision.

According to *Data Governance: The Definitive Guide*, data quality is defined by three main characteristics:[4]

Accuracy

> Is the collected data factually correct? Are there duplicate values? Are the numeric values accurate?

Completeness

> Are the records complete? Do all required fields contain valid values?

Timeliness

> Are records available in a timely fashion?

Each of these characteristics is quite nuanced. For example, how do we think about bots and web scrapers when dealing with web event data? If we intend to analyze the customer journey, we must have a process that lets us separate humans from machine-generated traffic. Any bot-generated events misclassified as *human* present data accuracy issues, and vice versa.

A variety of interesting problems arise concerning completeness and timeliness. In the Google paper introducing the Dataflow model, the authors give the example of an offline video platform that displays ads.[5] The platform downloads video and ads while a connection is present, allows the user to watch these while offline, and then uploads ad view data once a connection is present again. This data may arrive late, well after the ads are watched. How does the platform handle billing for the ads?

Fundamentally, this problem can't be solved by purely technical means. Rather, engineers will need to determine their standards for late-arriving data and enforce these uniformly, possibly with the help of various technology tools.

Data quality sits across the boundary of human and technology problems. Data engineers need robust processes to collect actionable human feedback on data quality and use technology tools to detect quality issues

preemptively before downstream users ever see them. We cover these collection processes in the appropriate chapters throughout this book.

MASTER DATA MANAGEMENT

Master data is data about business entities such as employees, customers, products, and locations. As organizations grow larger and more complex through organic growth and acquisitions, and collaborate with other businesses, maintaining a consistent picture of entities and identities becomes more and more challenging.

Master data management (MDM) is the practice of building consistent entity definitions known as *golden records*. Golden records harmonize entity data across an organization and with its partners. MDM is a business operations process facilitated by building and deploying technology tools. For example, an MDM team might determine a standard format for addresses, and then work with data engineers to build an API to return consistent addresses and a system that uses address data to match customer records across company divisions.

MDM reaches across the full data cycle into operational databases. It may fall directly under the purview of data engineering, but is often the assigned responsibility of a dedicated team that works across the organization. Even if they don't own MDM, data engineers must always be aware of it, as they will collaborate on MDM initiatives.

Data modeling and design

To derive business insights from data, through business analytics and data science, the data must be in a usable form. The process for converting data into a usable form is known as *data modeling and design*. Whereas we traditionally think of data modeling as a problem for database administrators (DBAs) and ETL developers, data modeling can happen almost anywhere in an organization. Firmware engineers develop the data format of a record for an IoT device, or web application developers design

the JSON response to an API call or a MySQL table schema—these are all instances of data modeling and design.

Data modeling has become more challenging because of the variety of new data sources and use cases. For instance, strict normalization doesn't work well with event data. Fortunately, a new generation of data tools increases the flexibility of data models, while retaining logical separations of measures, dimensions, attributes, and hierarchies. Cloud data warehouses support the ingestion of enormous quantities of denormalized and semistructured data, while still supporting common data modeling patterns, such as Kimball, Inmon, and data vault. Data processing frameworks such as Spark can ingest a whole spectrum of data, from flat structured relational records to raw unstructured text. We discuss these data modeling and transformation patterns in greater detail in Chapter 8.

With the wide variety of data that engineers must cope with, there is a temptation to throw up our hands and give up on data modeling. This is a terrible idea with harrowing consequences, made evident when people murmur of the write once, read never (WORN) access pattern or refer to a *data swamp*. Data engineers need to understand modeling best practices as well as develop the flexibility to apply the appropriate level and type of modeling to the data source and use case.

Data lineage

As data moves through its lifecycle, how do you know what system affected the data or what the data is composed of as it gets passed around and transformed? *Data lineage* describes the recording of an audit trail of data through its lifecycle, tracking both the systems that process the data and the upstream data it depends on.

Data lineage helps with error tracking, accountability, and debugging of data and the systems that process it. It has the obvious benefit of giving an audit trail for the data lifecycle and helps with compliance. For example, if a user would like their data deleted from your systems, having lineage for that data lets you know where that data is stored and its dependencies.

Data lineage has been around for a long time in larger companies with strict compliance standards. However, it's now being more widely adopted in smaller companies as data management becomes mainstream. We also note that Andy Petrella's concept of Data Observability Driven Development (DODD) is closely related to data lineage. DODD observes data all along its lineage. This process is applied during development, testing, and finally production to deliver quality and conformity to expectations.

Data integration and interoperability

Data integration and interoperability is the process of integrating data across tools and processes. As we move away from a single-stack approach to analytics and toward a heterogeneous cloud environment in which various tools process data on demand, integration and interoperability occupy an ever-widening swath of the data engineer's job.

Increasingly, integration happens through general-purpose APIs rather than custom database connections. For example, a data pipeline might pull data from the Salesforce API, store it to Amazon S3, call the Snowflake API to load it into a table, call the API again to run a query, and then export the results to S3 where Spark can consume them.

All of this activity can be managed with relatively simple Python code that talks to data systems rather than handling data directly. While the complexity of interacting with data systems has decreased, the number of systems and the complexity of pipelines has dramatically increased. Engineers starting from scratch quickly outgrow the capabilities of bespoke scripting and stumble into the need for *orchestration.* Orchestration is one of our undercurrents, and we discuss it in detail in "Orchestration".

Data lifecycle management

The advent of data lakes encouraged organizations to ignore data archival and destruction. Why discard data when you can simply add more storage ad infinitum? Two changes have encouraged engineers to pay more attention to what happens at the end of the data engineering lifecycle.

First, data is increasingly stored in the cloud. This means we have pay-as-you-go storage costs instead of large up-front capital expenditures for an on-premises data lake. When every byte shows up on a monthly AWS statement, CFOs see opportunities for savings. Cloud environments make data archival a relatively straightforward process. Major cloud vendors offer archival-specific object storage classes that allow long-term data retention at an extremely low cost, assuming very infrequent access (it should be noted that data retrieval isn't so cheap, but that's for another conversation). These storage classes also support extra policy controls to prevent accidental or deliberate deletion of critical archives.

Second, privacy and data retention laws such as the GDPR and the CCPA require data engineers to actively manage data destruction to respect users' "right to be forgotten." Data engineers must know what consumer data they retain and must have procedures to destroy data in response to requests and compliance requirements.

Data destruction is straightforward in a cloud data warehouse. SQL semantics allow deletion of rows conforming to a `where` clause. Data destruction was more challenging in data lakes, where write-once, read-many was the default storage pattern. Tools such as Hive ACID and Delta Lake allow easy management of deletion transactions at scale. New generations of metadata management, data lineage, and cataloging tools will also streamline the end of the data engineering lifecycle.

Ethics and privacy

> *Ethical behavior is doing the right thing when no one else is watching.*
>
> —Aldo Leopold

The last several years of data breaches, misinformation, and mishandling of data make one thing clear: data impacts people. Data used to live in the Wild West, freely collected and traded like baseball cards. Those days are long gone. Whereas data's ethical and privacy implications were once considered nice to have, like security, they're now central to the general data lifecycle. Data engineers need to do the right thing when no one else is

watching, because everyone will be watching someday. We hope that more organizations will encourage a culture of good data ethics and privacy.

How do ethics and privacy impact the data engineering lifecycle? Data engineers need to ensure that datasets mask personally identifiable information (PII) and other sensitive information; bias can be identified and tracked in datasets as they are transformed. Regulatory requirements and compliance penalties are only growing. Ensure that your data assets are compliant with a growing number of data regulations, such as GDPR and CCPA. Please take this seriously. We offer tips throughout the book to ensure that you're baking ethics and privacy into the data engineering lifecycle.

Orchestration

> *We think that orchestration matters because we view it as really the center of gravity of both the data platform as well as the data lifecycle, the software development lifecycle as it comes to data.*
>
> —Nick Schrock, founder of Elementl

Orchestration is not only a central DataOps process, but also a critical part of the engineering and deployment flow for data jobs. So, what is orchestration?

Orchestration is the process of coordinating many jobs to run as quickly and efficiently as possible on a scheduled cadence. For instance, people often refer to orchestration tools like Apache Airflow as *schedulers*. This isn't quite accurate. A pure scheduler, such as cron, is aware only of time; an orchestration engine builds in metadata on job dependencies, generally in the form of a directed acyclic graph (DAG). The DAG can be run once or scheduled to run at a fixed interval of daily, weekly, every hour, every five minutes, etc.

As we discuss orchestration throughout this book, we assume that an orchestration system stays online with high availability. This allows the orchestration system to sense and monitor constantly without human intervention and run new jobs anytime they are deployed. An orchestration

system monitors jobs that it manages and kicks off new tasks as internal DAG dependencies are completed. It can also monitor external systems and tools to watch for data to arrive and criteria to be met. When certain conditions go out of bounds, the system also sets error conditions and sends alerts through email or other channels. You might set an expected completion time of 10 a.m. for overnight daily data pipelines. If jobs are not done by this time, alerts go out to data engineers and consumers.

Orchestration systems also build job history capabilities, visualization, and alerting. Advanced orchestration engines can backfill new DAGs or individual tasks as they are added to a DAG. They also support dependencies over a time range. For example, a monthly reporting job might check that an ETL job has been completed for the full month before starting.

Orchestration has long been a key capability for data processing but was not often top of mind nor accessible to anyone except the largest companies. Enterprises used various tools to manage job flows, but these were expensive, out of reach of small startups, and generally not extensible. Apache Oozie was extremely popular in the 2010s, but it was designed to work within a Hadoop cluster and was difficult to use in a more heterogeneous environment. Facebook developed Dataswarm for internal use in the late 2000s; this inspired popular tools such as Airflow, introduced by Airbnb in 2014.

Airflow was open source from its inception, and was widely adopted. It was written in Python, making it highly extensible to almost any use case imaginable. While many other interesting open source orchestration projects exist, such as Luigi and Conductor, Airflow is arguably the mindshare leader for the time being. Airflow arrived just as data processing was becoming more abstract and accessible, and engineers were increasingly interested in coordinating complex flows across multiple processors and storage systems, especially in cloud environments.

At this writing, several nascent open source projects aim to mimic the best elements of Airflow's core design while improving on it in key areas. Some

of the most interesting examples are Prefect and Dagster, which aim to improve the portability and testability of DAGs to allow engineers to move from local development to production more easily. Argo is an orchestration engine built around Kubernetes primitives; Metaflow is an open source project out of Netflix that aims to improve data science orchestration.

We must point out that orchestration is strictly a batch concept. The streaming alternative to orchestrated task DAGs is the streaming DAG. Streaming DAGs remain challenging to build and maintain, but next-generation streaming platforms such as Pulsar aim to dramatically reduce the engineering and operational burden. We talk more about these developments Chapter 8.

DataOps

DataOps maps the best practices of Agile methodology, DevOps, and statistical process control (SPC) to data. Whereas DevOps aims to improve the release and quality of software products, DataOps does the same thing for data products.

Data products differ from software products because of the way data is used. A software product provides specific functionality and technical features for end users. By contrast, a data product is built around sound business logic and metrics, whose users make decisions or build models that perform automated actions. A data engineer must understand both the technical aspects of building software products, and the business logic, quality, and metrics that will create excellent data products.

Like DevOps, DataOps borrows much from lean manufacturing and supply chain management, mixing people, processes, and technology to reduce time to value. As Data Kitchen (experts in DataOps) describes it:[6]

DataOps is a collection of technical practices, workflows, cultural norms, and architectural patterns that enable:

- *Rapid innovation and experimentation delivering new insights to customers with increasing velocity*
- *Extremely high data quality and very low error rates*
- *Collaboration across complex arrays of people, technology, and environments*
- *Clear measurement, monitoring, and transparency of results*

Lean practices (such as lead time reduction and minimizing defects) and the resulting improvements to quality and productivity are things we are glad to see gaining momentum both in software and data operations.

First and foremost, DataOps is a set of cultural habits; the data engineering team needs to adopt a cycle of communicating and collaborating with the business, breaking down silos, continuously learning from successes and mistakes, and rapid iteration. Only when these cultural habits are set in place can the team get the best results from technology and tools.

Depending on a company's data maturity, a data engineer has some options to build DataOps into the fabric of the overall data engineering lifecycle. If the company has no preexisting data infrastructure or practices, DataOps is very much a greenfield opportunity that can be baked in from day one. With an existing project or infrastructure that lacks DataOps, a data engineer can begin adding DataOps into workflows. We suggest first starting with observability and monitoring to get a window into the performance of a system, then adding in automation and incident response. A data engineer may work alongside an existing DataOps team to improve the data engineering lifecycle in a data-mature company. In all cases, a data engineer must be aware of the philosophy and technical aspects of DataOps.

DataOps has three core technical elements: automation, monitoring and observability, and incident response (Figure 2-8). Let's look at each of these pieces and how they relate to the data engineering lifecycle.

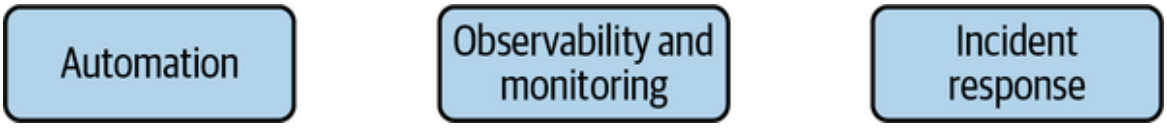


Figure 2-8. The three pillars of DataOps

Automation

Automation enables reliability and consistency in the DataOps process and allows data engineers to quickly deploy new product features, and improvements to existing workflows. DataOps automation has a similar framework and workflow to DevOps, consisting of change management (environment, code, and data version control), continuous integration/continuous deployment (CI/CD), and configuration as code. Like DevOps, DataOps practices monitor and maintain the reliability of technology and systems (data pipelines, orchestration, etc.), with the added dimension of checking for data quality, data/model drift, metadata integrity, and more.

Let's briefly discuss the evolution of DataOps automation within a hypothetical organization. An organization with a low level of DataOps maturity often attempts to schedule multiple stages of data transformation processes using cron jobs. This works well for a while. As data pipelines become more complicated, several things are likely to happen. If the cron jobs are hosted on a cloud instance, the instance may have an operational problem, causing the jobs to stop running unexpectedly. As the spacing between jobs becomes tighter, a job will eventually run long, causing a subsequent job to fail or produce stale data. Engineers may not be aware of job failures until they hear from analysts that their reports are out-of-date.

As the organization's data maturity grows, data engineers will typically adopt an orchestration framework, perhaps Airflow or Dagster. Data engineers are aware that Airflow presents an operational burden, but the benefits of orchestration eventually outweigh the complexity. Engineers will gradually migrate their cron jobs to Airflow jobs. Now, dependencies are checked before jobs run. More transformation jobs can be packed into a given time because each job can start as soon as upstream data is ready rather than at a fixed, predetermined time.

The data engineering team still has room for operational improvements. A data scientist eventually deploys a broken DAG, bringing down the Airflow web server and leaving the data team operationally blind. After enough such headaches, the data engineering team members realize that they need to stop allowing manual DAG deployments. In their next phase of operational maturity, they adopt automated DAG deployment. DAGs are tested before deployment, and monitoring processes ensure that the new DAGs start running properly. In addition, data engineers block the deployment of new Python dependencies until installation is validated. After automation is adopted, the data team is much happier and experiences far fewer headaches.

One of the tenets of the DataOps Manifesto is "Embrace change." This does not mean change for the sake of change, but goal-oriented change. At each stage of our automation journey, opportunities exist for operational improvement. Even at the high level of maturity that we've described here, further room for improvement remains. Engineers might embrace a next-generation orchestration framework that builds in better metadata capabilities. Or they might try to develop a framework that builds DAGs automatically based on data-lineage specifications. The main point is that engineers constantly seek to implement improvements in automation that will reduce their workload and increase the value that they deliver to the business.

Observability and monitoring

As we tell our clients, "Data is a silent killer." We've seen countless examples of bad data lingering in reports for months or years. Executives may make key decisions from this bad data, discovering the error only much later. The outcomes are usually bad and sometimes catastrophic for the business. Initiatives are undermined and destroyed, years of work wasted. In some of the worst cases, bad data may lead companies to financial ruin.

Another horror story occurs when the systems that create the data for reports randomly stop working, resulting in reports being delayed by