Data Engineering Teams

Creating Successful Big Data Teams and Products

A practical guide to managing and creating Big Data teams

Jesse Anderson

Data Engineering Teams

Creating Successful Big Data Teams and Products

Jesse Anderson

Version 2.0.764ea52

ISBN 978-1-7326496-0-6

Contents

Second Edition Preface 5

1 Introduction **7**
About This Book 7
Warnings and Success Stories 8
Who Should Read This Book 9
Navigating the Book's Chapters 9
Conventions Used in This Book 10

2 The Need for Data Engineering **11**
Big Data 11
Why Is Big Data So Much More Complicated? 11
Distributed Systems Are Hard 15
What Does It All Mean? 16
What Does It Mean for Software Engineering Teams? 16
What Does It Mean for Data Warehousing Teams? 17
It's Time to Change How We Manage Data Teams 17
What Is a Data Engineering Team? 19
Skills Needed in a Team 20
Skills Gap Analysis 24
Skill Gap Analysis Results 26
What I Look for in Data Engineering Teams 28
Operations 30
Quality Assurance 31

3 Data Engineers **33**
What It Looks Like When a Team Is Missing 33
What Is a Data Engineer? 37
What I Look for in Data Engineers 38
Qualified Data Engineers 39
Not Just Data Warehousing and DBAs 39
Ability Gap 40

4 Productive Data Engineering Teams **42**
Advice for Small Teams and Startups on Data Engineering 42

Themes and Thoughts of a Data Engineering Team 45
Hub of the Wheel 47
How to Work with a Data Science Team 49
Why Some Data Science Teams Don't Think They Need Data Engineering 53
What Happens When Data Science Teams Add A Data Engineer 54
How to Work with a Data Warehousing Team 57
How to Work with an Analytics or Business Intelligence Team 58
How I Evaluate Teams 58
Equipment and Resources 60

5 Creating Data Pipelines 62
Thought Frameworks 62
Building Data Pipelines 63
Knowledge of Use Case 64
Right Tool for the Job 65

6 Running Successful Data Engineering Projects 68
Crawl, Walk, Run 68
Technologies 70
KPIs Every Business Should Have For Its Data Teams 70
Why Do Teams Fail? 73
Why Do Teams Succeed? 75
Paying the Piper 77
Some Technologies Are Just Dead Ends 78
What if You Have Skills Gaps and Still Have to Transition to Big Data? . 79

7 Steps to Creating Big Data Solutions 82
Pre-project Steps 82
Use Case 83
Evaluate the Team 85
Choose the Technologies 86
Write the Code 86
Evaluate 87
Repeat 87
Probability of Success 88

8 Conclusion 89
Best Efforts 89
About the Author 90

A Appendix A: Criteo's Data Engineering Journey 91
Introduction . . . 91
About This Interview . . . 91
Background . . . 91
Organizational Structure . . . 92
Prologue to Change . . . 92
A Big Data Approach . . . 93
Roadblocks With DBAs . . . 94
Creating New Systems . . . 95
Executive Sponsorship . . . 96
Better Tools Now . . . 96
Governance and Engagement Models . . . 97
The Anarchists of Data Versus Pirate Infrastructure . . . 98
Organizational Stress Points . . . 99
Passing the Baton To François . . . 100
Stupidly Simple . . . 101
Friction and Getting Into Production . . . 102
Toiling . . . 103
Cultural Differences . . . 103
Changes Due To COVID-19 . . . 104
Effective Post Mortems . . . 105
Advice . . . 105

Second Edition Preface

I first started writing the first edition of Data Engineering Teams in 2017. I realized we were making some fundamental mistakes in the industry, and I made it my mission to educate an entire nascent industry on the best practices. I spent a significant amount of time since 2012 talking about the teams behind data and how they should be organized rather than just focusing on the technologies of the time.

I've been proud to see so many of my ideas and hypotheses becoming a reality. As you read this book, you might think the concepts are commonplace or well-known. Trust me; these weren't commonplace or well-known when I wrote the book's first edition. Some ideas in the book were downright controversial and met with intense criticism. Much of what you see as the current industry best practices came from this book.

In this second edition, I'm going deeper and updating. There are some new technologies in play, concepts, and technologies. I want the second edition to be the most complete and authoritative work on data engineering teams.

CHAPTER 1

Introduction

About This Book

You're a manager at a company with growing data needs, and your engineers are starting to say "can't" a lot–or have been saying "can't" for some time. The *can'ts* always revolve around processing vast quantities of data. Sometimes, you're asking for an analysis that spans years of company activity. You're asking for processing that spans billions of rows or looks at every customer you have. Other times, you're just asking for the day-to-day reports and data products that are the lifeblood of your business. Every time the data gets unwieldy, the engineers encounter a technical limitation that keeps you from accomplishing your goal. You have to compromise every single time due to the technology and your engineers' knowledge limitations.

All these familiar problems rest on one fact: You're sitting on a big pile of data. A really big pile. Buried somewhere in there are a bunch of really, really valuable insights–the kind of insights that could fundamentally and profoundly change your business. Maybe you could more deeply understand your customers. Perhaps it has to do with fraud or customer retention.

You, or your boss, have heard about *big data*, this magic bullet that everyone is talking about. It's allowing companies to free themselves of the technical limitations listed here. It's also allowing your competition to move past you.

Here's the good news: Managers who understand the challenges of making the big data transition can adopt policies at any point in their journey to create successful outcomes. For more than ten years, I've mentored companies through these processes, and this book shares the principles behind the techniques that have worked. I've made it my mission to get data engineering to be recognized as the wonderful and challenging discipline it is.

Looking back on the first edition of this book, I can truly say that my efforts and the field of data engineering have both made dramatic headway. In this second edition, I expand on what I already shared in the first edition.

The process isn't magic, and big data seems like a magic bullet only to the uninformed observer. In fact, a lot of hard work goes into making a successful big data project. It takes a team with a lot of technical and managerial discipline. But if you follow the strategies and plans I outline in this book, you'll be way ahead of the 85% of your competitors who started down the big data road and failed.

When competitors or other companies talk about big data technologies failing or their project failing due to the technologies, they do not recognize a key and fundamental part of the project. They didn't follow the strategies and plans I outline in this book. They unfairly lay the blame on the technology. The reality is that their missteps lay several months or a year earlier when they started the project off on the wrong foot. I'll show you how to start it off right and increase your probability of success.

Warnings and Success Stories

This book grew from years of carefully observing teams and entire companies. I've also spent years mentoring and teaching hundreds of companies and thousands of students. These companies span the world and participate in all kinds of industries. My organizational analysis's scope starts with my first interaction with teams and continues as I follow up with those teams to see how their projects went. From there, I analyze what went right, what went wrong, and why.

Throughout this book, I will share some of these interactions and stories. They'll look something like this:

A cautionary story

⚠ Learn from this company's mistake(s)

These will be cautionary tales from the past. Learn from and avoid those mistakes.

Follow this example

✓ Learn from this company's success

These will be success stories from the past. Learn from and imitate their success.

My story

A personal experience

These are passing anecdotes from my time in the field. They will give you more background or color on a topic.

Who Should Read This Book

This book is primarily written for managers, VPs, and CxOs—people who are managing teams that are currently creating or about to start creating a big data solution. The book will help you understand why some teams succeed while so many others fail.

This book can also be useful to Enterprise and Data Architects, the bridge between the technical and business sides of the company. They have to make sure that the technology meets the needs of the business. They're the tip of the spear in advocating for big data technologies. This book will help you understand how to set the technical team up for success and alert you to some pitfalls to avoid.

Team leads, technical leads, and individual contributors have appreciated this book, too. They're often on the receiving end of why big data teams fail. This book will help these individuals understand the problems behind these failures to work with their management team to fix the issues.

Navigating the Book's Chapters

I highly suggest you read the entire book from start to finish to understand every concept and point. Without the whole background and all of the concepts, you may not understand fully why I recommend a technique or make a specific suggestion.

Here are the chapters and what we'll be covering in each chapter:

- Chapter 2 shows you why you need to rethink data engineering and launch a modern data engineering team. The key lies in the complexity of big data. A manager or team that doesn't internalize this understanding is doomed to failure.
- Chapter 3 shows you what a data engineering team is and what it consists of. I'll talk about what makes the team successful and the common patterns behind the unsuccessful teams.
- Chapter 4 talks about the data engineers themselves. I'll show you how and why data engineers are different from software engineers.
- Chapter 5 shows you how to make your data engineering team productive after you've created it. I'll illustrate how data engineering teams function within an organization and how they interact with the rest of the company.
- Chapter 6 covers how to create data pipelines—the basic deliverable of data engineering teams. I'll share how I mentor data engineers to make the most performant, reliable, and time-efficient pipelines.
- Chapter 7 shares the secrets to creating successful big data projects. I'll talk about breaking up the project into manageable pieces, and I'll share the reasons why teams succeed or fail.
- Chapter 8 shows you the exact steps to creating big data solutions. I've taught these steps to companies all over the world. They've used them to increase their probability of success. But don't just skip to this chapter and think you're going to be successful!

Conventions Used in This Book

A *DBA* (Database Administrator) is someone who maintains a database and often writes SQL queries. For the sake of simplicity, in this book, I group together several commonly used job titles for people who write SQL queries. In addition to DBA, these titles include Data Warehouse Engineer, SQL Developer, and ETL Developer.

I also use the terms *software engineer* and *programmer* interchangeably. Both refer to someone who has programming or software engineering skills. They are the team members responsible for writing the project's code.

With all that negative stuff and housekeeping out of the way, let's start mining for gold.

CHAPTER 2

The Need for Data Engineering

Big Data

The heading is a buzzword but a useful one. At its core, the emergence of big data is why we need data engineering teams. We have a proliferation of data embodying a massive amount of potential business value. We have a high demand for data products that unlock business value but a low supply of people who can create them.

Out of this demand comes the creation of teams dedicated to processing data. At least, they *should* be dedicated. It takes concerted effort and specialization to coalesce as a data engineering team.

Without this focus, IT departments enter a vicious cycle happening at companies and enterprises around the world. No one recognizes when the quantity of data grows past some threshold where a totally new approach is needed. The assumption among programmers and managers alike is that big data is just like small data. The short answer is that it isn't. The companies that never internalize this fact are doomed to fail repeatedly. Worse yet, they never understand why they failed.

Why Is Big Data So Much More Complicated?

Let's start off with a diagram that shows a sample architecture for a mobile product, with a database back end. Figure 2.1 illustrates a run-of-the-mill mobile application that uses a web service to store data in a database.

Now, suppose that the mobile app got considerably more complicated, and look at the resulting architecture diagram (Figure 2.2).

My point is that I could just reuse the architecture from Figure 2.1. A more complex mobile solution usually requires more code or more web service calls but no additional technologies or architectural elements.

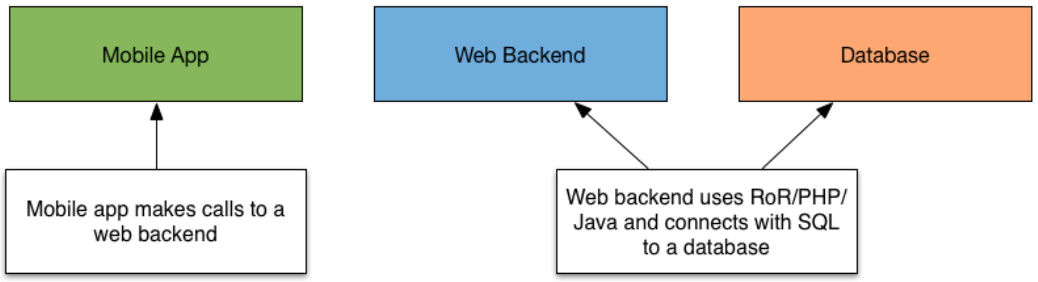

Mobile App
Web Backend
Database
Mobile app makes calls to a web backend
Web backend uses RoR/PHP/ Java and connects with SQL to a database

Figure 2.1: Simple architecture diagram for a mobile application, with a web backend.

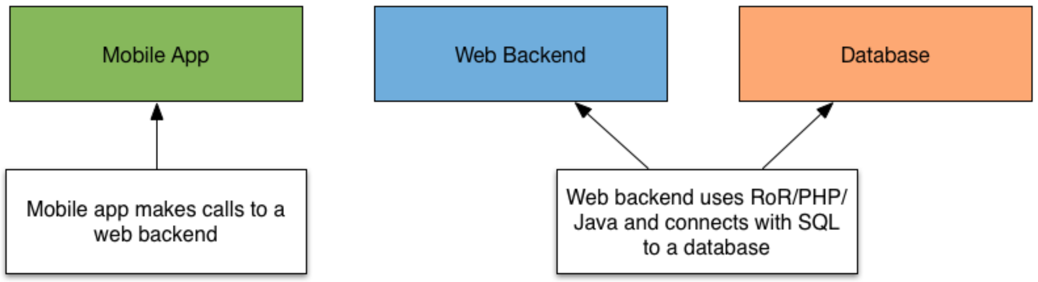

Mobile App
Web Backend
Database
Mobile app makes calls to a web backend
Web backend uses RoR/PHP/ Java and connects with SQL to a database

Figure 2.2: Complex architecture diagram for a mobile application with a web backend (yes, it’s the same as the simple one).

Now let's turn to a starter Spark solution, illustrated in Figure 2.3.

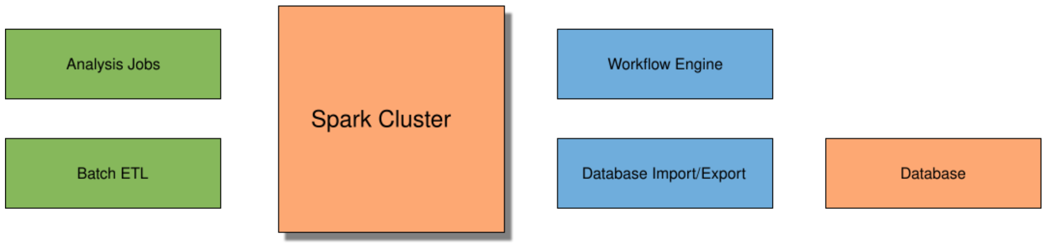


Figure 2.3: Simple architecture diagram for a Spark project.

As you can see, Spark weighs in at double the number of boxes shown in Figures 2.1 and 2.2. Why? What's the big deal? A "simple" Spark solution actually isn't very simple at all. You might think of this more as a starting point or, to put it another way, as "crawling" with Spark. A Spark solution's "Hello World!" is more complicated than other solutions' intermediate to advanced setups. Look at the source code for a "Hello World!" application in big data to see it's not so simple.

To open your eyes to the scope of the big data challenge, think about the growth of the big data project and look at the architecture of the more complex big data solution in Figure 2.4.

Yes, that's a lot of boxes representing various types of components you may need when dealing with a complex big data solution. This l is what I call the "running" phase of a big data project.

You might think I'm exaggerating the number of technologies to make a point; I'm not. I've taught at companies where this is their basic architectural stack. I've also taught at companies with twice as much complexity.

Instead of just looking at boxes, let's compare the amount of training required for a complex mobile application and a big data application, assuming that the programmer already knows Java and SQL. Based on my experience, a complex mobile course would take four to five days, whereas a complex big data course would take 18 to 20 days. And the big data estimate is rather optimistic because it assumes you can quickly grok the distributed systems side. In my experience teaching courses, a data engineer can learn mobile, but a mobile engineer has a tough time learning data engineering.

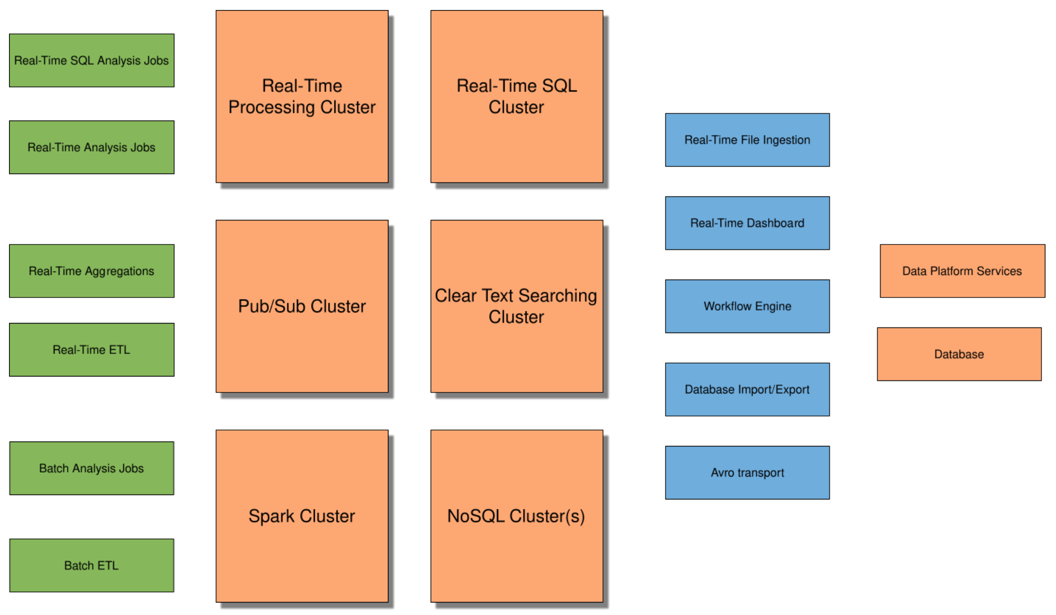


Figure 2.4: More complex architecture diagram for a big data project with real-time components.

Flavor of the month

While teaching at a company, I overheard one of the students say, "This is what we'll be doing this month." That company expected their developers to become experts during that week's training, develop the software, and then move on to another unrelated technology after that month was done. Bouncing around on technologies is how you set your developers up for failure.

It struck me as odd that something like big data takes individuals six months to a year to learn and get good at. This team, in contrast, was given only a month to be successful. I've seen uniquely gifted individuals understand big data in two to three months. On the other hand, your average enterprise developer takes six months to a year to really understand big data and become productive.

I often point out that data engineers need to know 10 to 30 different technologies to choose the right tool for the job. Data engineering is hard because we're taking ten complex systems, for example, and making them work together at a large scale. There are about ten technologies in Figure 2.4. To make the right decision in choosing, for example, a NoSQL cluster, you'll need to have learned the pros and cons of five to ten different NoSQL technologies. From that list, you can then narrow the choices down to two to three for a more in-depth look.

During this period, you might compare, for example, HBase and Cassandra. There are a plethora of questions you'd need to ask before settling on a choice. Which is the right data store? That comes down to you knowing what you're doing with the data—for instance, do you need ACID guarantees?

And don't get me started on choosing a real-time processing system, which requires knowledge of and comparison among Kafka, Spark, Flink, Storm, Heron, Flume—the list goes on.

Distributed Systems Are Hard

Distributed systems frameworks like Spark make it easy to scale out applications, right? Well, yes and no.

Yes, distributed systems frameworks make it easy to focus on the use case at hand. You're not thinking about how and where to spin up a task or threading. You're not worried about how to make a network connection, serialize some data, and then deserialize it again. Distributed systems frameworks allow you to focus on what you want to do rather than coordinating all of the pieces to do it.

But no, distributed systems frameworks don't make everything easy. They make it even more important to know the weaknesses and strengths of the underlying system. They assume that you know and understand the unwritten rules and design decisions made when the frameworks were created. One of those assumptions is that you already know distributed systems or can learn the fundamentals quickly.

I think Kafka is an excellent example of how designers added ten times as much complexity in making a system distributed. Think of your favorite messaging system, such as RabbitMQ. Now, imagine you added ten times the complexity to it. This complexity isn't added through some kind of over-engineering. It's simply the result of making it feasible to distribute processing across several different machines. Creating a system that is distributed, fault-tolerant, and scalable adds complexity.

What Does It All Mean?

We can make our APIs as simple as we want, but doing so only masks the greater complexity of creating the system. As newer distributed systems frameworks come out, they're changing how data is processed; they're not fundamentally altering the complexity of these big data systems.

Distributed systems are hard. They'll be largely similar at a conceptual level but will have subtle differences that bite you hard enough to lose five months of development time if one of them makes a use case impossible. When you debug an issue or figure out why something isn't performing, you won't be looking at a single process running on a single computer; you'll be looking at hundreds of processes running on tens of computers.

What Does It Mean for Software Engineering Teams?

The members of software engineering teams are intelligent people. Yet they often fail at a project because they underestimate the level of complexity required for a big data solution.

Trust me. big data is a very different and complex animal.

Underestimating Big Data Complexity

I've heard software engineers say they can teach my big data technology classes. They're under the impression that big data is easy. I've listened to software engineers say they'll be done with a four-day course in half a day. I've heard DBAs tell me that big data processing is no different from a data warehouse. They think I don't know what I'm talking about or that I'm overstating the difficulty.

When this happens, I ask them to sit tight. You may be one of the very, very, very few outliers because they do exist. But more than likely, you're right there in the middle of the bell curve, and the transition will be complex for your team.

What Does It Mean for Data Warehousing Teams?

A company often assigns data engineering to its existing Data Warehouse team because they already have data experts there who believe they can handle the new responsibilities. Existing data teams may also be wary and envious of the prospect of hiring a new team just for big data. On the surface, it seems like a logical assumption that the Data Warehouse team is the right one for all things data-related. This sort of thinking is one of the most common ways a big data project fails. The data warehouse team isn't ready for the massive jump in complexity.

Think back to Figures 2.1 through 2.4, which showed the fundamental difference in complexity between small and big data. The same conclusions apply when creating teams to manage data. The data warehouse team deals with a single database technology, or one box. More advanced teams add maybe one more box for ETL. The point is that the data warehousing team is not ready for the rigors and complications of big data.

It's Time to Change How We Manage Data Teams

As a distributed systems person, I'm used to figuring out how to spread a problem out to the largest possible number of computers. Spreading out a problem lets me capitalize on resources far better and faster. Most companies fail to apply this optimization to their real-world processes for management and innovation.

In business and technology decision-making, we're centralizing innovation and strategizing onto just a relatively small part of the company–the management team. We aren't spreading out the problem to individual contributors to get more ideas, perspectives, and solutions to problems. We're leaving the best and brightest–and most importantly, the people closest to the data and business problems–out of the conversation.

Much of organizational thinking and management philosophy today still comes from industrial-age methods. With Agile, we've realized that we need to involve individual contributors more in planning. We'll ask them how long something will take (scoping or story points). They can try to add technical tasks (technical debt) to their task list, but the process doesn't allow for new features or ideation. Instead, ideation is left to a product manager or some other management layer.

Data teams replicate this industrial-age top-down innovation. Their managers think they are typical knowledge workers who need to be told what to do, ignoring the crucial data aspects that make data teams unique. So we're missing out on the insights that our individual contributors might have. When will we start listening to the people closest to the data? What insights could the data teams closest to the business problem have? We're missing out on considerable brainpower if we aren't listening to them or asking these questions.

While writing my book Data Teams, I started to find companies who exploited the latent brainpower in data teams. Their management got out of the way, and they started listening to the people closest to the data. I've started calling this innovation *emergence*.

What does emergence look like in the real world? I set out to find that answer during my interview with Stitch Fix's Brad Klingenberg (Chief Analytics Office) and Eric Colson (Chief Analytics Office Emeritus). Brad said, "The approach we've taken is much more like being a gardener. You just want to create circumstances where people can do good work and, occasionally you need to trim a branch back or make room for a new sapling, but generally, you're just trying to get the conditions right to then get out of the way." You can read the rest of the interview in my Data Teams book. Stitch Fix created a site talking about how they cultivate algorithms.

I read David Marquet's Turn the Ship Around: A True Story of Turning Followers

Into Leaders. David increased the success of a low-performing submarine by taking their followers (enlisted) and making them leaders. I realize that the book is about the U.S. Navy and submarines, not about data teams. However, I argue that the mindset shift of turning individual contributors (followers) into leaders is the exact process that data teams need to make for emergence to happen. David also shows that this process can work even when the stakes are as high as life and death on a submarine. If you can work with high stakes, we know it can work for other, less risky situations.

This idea comes out in Aaron Dignan's Brave New Work. A big theme in the book is that teams need a clear purpose. This purpose comes after better integrating the data teams and communication with the business side. This reorientation allows the data teams to directly see the value and results of what they're doing. A direct view into results creates a virtuous cycle that makes more effective teams. As we come out of the COVID-19 and other economic issues era, we have to draw on our teams' talents to achieve their maximum potential. We aren't utilizing the entire team by restricting innovative ideas to a select few. We have to start using the whole team for our innovation and strategy. Only then can we use analytics and insights to their highest potential.

What Is a Data Engineering Team?

If the data warehouse and software engineering team are up to managing big data, what is the right team? The answer to that is a brand-new type of team. It's made up of members of other teams and has a wider variety of specialties and skills. This new kind of team is called a *data engineering team.*

People often confuse data engineering teams with Data Engineers. They aren't the same. A data engineering team isn't made up of a single type of person or title. When a data engineering team isn't multidisciplinary, I deduct points from their probability of success.

This multidisciplinary approach is required because the team doesn't just handle data. They aren't limited to programming behind the scenes. The team interacts with virtually every part of the company, and their products become its lifeblood. This data product isn't your average backend software; this will be how your company improves its bottom line and starts making money from its data.

That breadth and centrality of data products require people who aren't just programmers. They're programmers who have cross-trained in other fields and skills. The team also has some members or skillsets that aren't typically found in an orthodox software engineering team.

The team is almost entirely focused on big data. Everyone on the data engineering team needs to understand and know how big data systems work and are created. Despite the variety of skills on the team, this aspect of specialization is consistent.

That isn't to say that a data engineering team can't create small data solutions. Some small data technologies will be part of a big data solution. Using the data engineering team for small data is entirely possible but is generally a waste of their specialty.

For an overview of the philosophy behind the effective use of data engineers, please view the video or transcript of my presentation, Creating a Data Engineering Culture.

Skills Needed in a Team

As I said earlier, a data engineering team is a group of complementary skills and people. The team itself is usually dominated by software engineers but has other titles that aren't usually part of a software engineering team.

The skills needed on a data engineering team are:

- Distributed systems
- Programming
- Analysis
- Visual communication
- Verbal communication
- Project veteran
- Schema
- Domain knowledge

We'll take each of these and see why the team needs the skill.

Distributed Systems

Big data is a subspecialty of distributed systems. And distributed systems are hard. The systems have to take many different computers and make them work together. This distribution requires systems to be designed differently. You have to plan how data moves around those computers.

Having taught distributed systems for many years, I know this is something that takes people time to understand. It takes time and effort to get it right.

Common titles with this skill: Software Architect, Software Engineer

Programming

This is the skill for someone who writes the code. They are tasked with taking the use case and writing the code that executes it on the big data framework.

The actual code for big data frameworks isn't difficult. Usually, the most significant difficulty is keeping all of the different APIs straight; programmers will need to know 10 to 30 different technologies.

Programmers also bring engineering fundamentals to the team. The programmers demand continuous integration, unit tests, and engineering processes. Sometimes data engineering teams forget that they are still doing engineering and operate as if they've forgotten their fundamentals; the programmers have to uphold them.

Common titles with this skill: Software Engineer

Can't everyone program?

A company had me teach their DBAs how to program. It was an uphill battle for the team and one they didn't win. Out of the 20 students, only one person was capable of accomplishing her goals. Programming isn't the memorization of APIs; it's getting a computer to do what you want and creating a system. That company and team didn't understand that concept and thought they just needed to memorize some APIs. From there, they thought they could just look everything up on StackOverflow. That just isn't the case.

Analysis

A data engineering team releases data analyses as a product. These analyses can range from simple counts and sums to more complex products. The actual bar for skills can vary dramatically on data engineering teams; it depends entirely on the use case and organization. The quickest guide to the skill level needed for data analysis is the complexity of the data products. Are they relatively straightforward, or do they consist of equations that most programmers wouldn't understand?

At other times, a data product is a simple report that's given to another business unit. This report could be done with SQL queries.

Very advanced analysis is often the purview of a data scientist. I'll talk about data scientists and data engineering teams in Chapter 5.

Common titles with this skill: Software Engineer, Data Analyst, Business Intelligence Analyst, DBA

Visual Communication

A data engineering team needs to communicate its data products visually. Graphs and animations are often the best ways to show what's happening with data, especially vast amounts of it, so that others can readily use it. You'll often have to show data over time and with animation. This function combines programming and visualization.

A team member with visual communication skills will help you tell a graphic story with your data. They can show the data logically and with the proper aesthetics.

Common titles with this skill: Software Engineer, Business Intelligence Analyst, UX Engineer, UI Engineer, Graphic Artist

Verbal Communication

The data engineering team is the hub in the wheel where many spokes of the organization connect. We'll talk more about that concept later in Chapter 5, but the effect on staffing is essential here. You need people on the team who can communicate verbally with the other parts of your organization.

Your verbal communicator is responsible for helping other teams successfully use the big data platform or data products. They'll also need to speak to these teams about what data is available. Other data engineering teams will operate like internal solutions consultants.

This skill can mean the difference between increasing the use of the cluster and the work going to waste.

Common titles with this skill: Software Architect, Software Engineer, Technical Manager

Project Veteran

A project veteran has worked with big data and has had at least one solution in production for a while. This person is ideally someone who has extensive experience in distributed systems or, at the very least, extensive multithreading experience. This person brings the benefit of their substantial experience to the team.

The project veteran holds the team back from bad ideas. They have the experience to know when something is technically feasible but a bad idea in the real world. They will give the team some footing or long-term viewpoint on distributed systems. This experience translates into better designs that save the team money and time once things are in production.

Common titles with this skill: Senior Software Architect, Senior Software Engineer, Senior Technical Manager

Schema

This may be a surprising inclusion because you might assume that nobody on the data engineering team knows how to work with schemas or needs to. However, schemas often come into play when processing big data at some points. Members with this skill help teams layout data. They're responsible for creating the data definitions and designing its representation when it is stored, retrieved, transmitted, or received.

The importance of this skill emerges as data pipelines mature. I tell my classes that this skill makes or breaks you as your data pipelines become more complex. When you have 1 PB of data saved in some key-value store or NoSQL data store, you can't rewrite it every time a new field is added. This skill helps you look at the data you have and the data you need to define what your data looks like.

JSON (and even YAML) actually add a lot of overhead because every item of data includes the name of the field along with the value (for instance, "address" : "40"). XML adds even more because it consists of both opening and closing tags. Some 25% to 50% of the information is just the overhead of tags. Furthermore, if data in any of these formats is stored in plain text, storage costs can be huge. These hierarchical formats also make you serialize and deserialize the data every time it needs to be used.

The schema skill goes beyond simple data modeling. Practitioners understand the trade-offs between saving data as a string and a binary integer. They also implement binary formats such as Avro or Protobuf. They know to do this because data use grows as other groups in a company hear about its existence and capabilities. A format like Avro keeps the data engineering team from having to type-check everyone's code for correctness.

Common titles with this skill: DBA, Software Architect, Software Engineer

Domain Knowledge

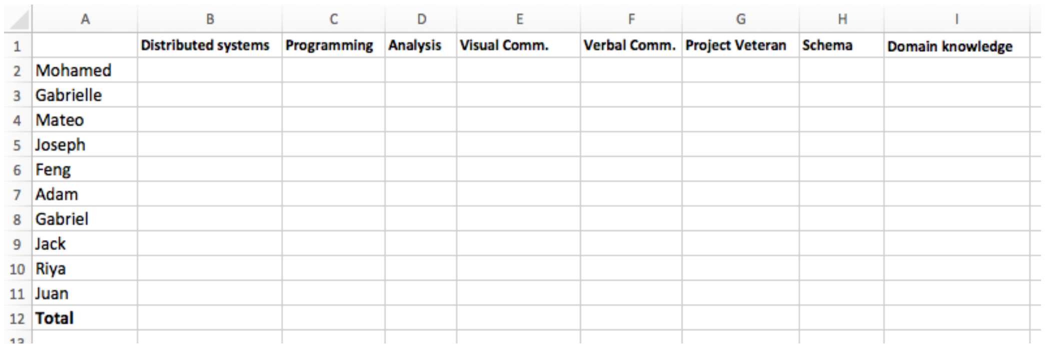
Some companies, and technical positions within companies, focus less on technology than on domain expertise. These jobs focus 80% of their effort on understanding and implementing the domain. They focus the other 20% on getting the technology right. Domain-focused jobs are especially prevalent in finance, health care, consumer products, and similar companies.

Domain knowledge needs to be part of the data engineering team. This person will need to know how the whole system works throughout the entire company. They'll need to deeply understand the domain for which you're creating data products because they will need to reflect this domain and be used within it.

Common titles with this skill: DBA, Software Architect, Software Engineer, Technical Manager, The Graybeard, Project Manager

Skills Gap Analysis

Now that you've seen each skill, you need to do what's called a *gap analysis* (based on a technique that Data Scientist Paco Nathan teaches in courses). Take a piece of paper and turn it width-wise, or open a spreadsheet and create eight columns–one for each skill listed earlier in this section. Then create rows for every person on the team, and list the names of everyone on the team in the far-left column. See Figure 3.1 for an example of how to do this.



	A	B	C	D	E	F	G	H	I
1		Distributed systems	Programming	Analysis	Visual Comm.	Verbal Comm.	Project Veteran	Schema	Domain knowledge
2	Mohamed								
3	Gabrielle								
4	Mateo								
5	Joseph								
6	Feng								
7	Adam								
8	Gabriel								
9	Jack								
10	Riya								
11	Juan								
12	**Total**								

Figure 3.1: Doing gap analysis using a spreadsheet

Now you're going to fill out the paper or computer spreadsheet. Go through everyone on the team and put an "X" whenever that team member has that particular skill. This assessment requires a very honest look—doing this analysis as honestly as possible could be the difference between success and failure for the team.

Now that you've placed the Xes on everyone's skills, total them up. This total will tell you a story about the makeup of your team. It will tell you where you're strong, weak, or maybe even nonexistent.

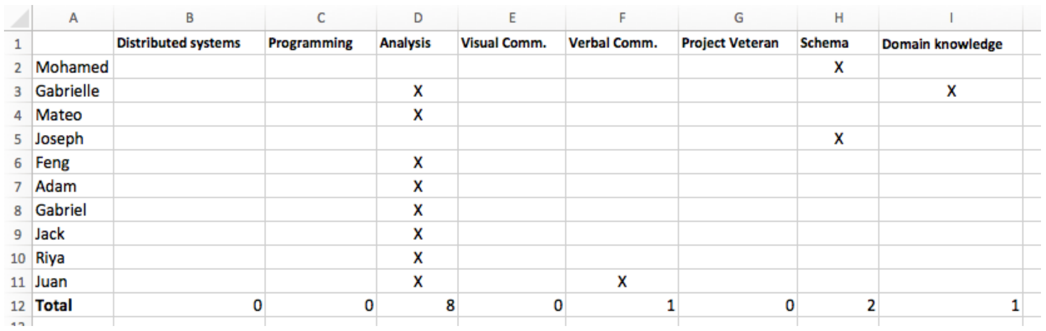
Do Two Halves Make a Whole?

Sometimes, people will do their gap analysis and put percentages or numbers with decimal points instead of an "X." They'll put a "0.1" or "0.5" instead of "X." Based on my experience, ten people with a "0.1" don't make a "1" in total. They're really just several beginners with a "0.1," and all have the same limitations on the skill. This exercise should be a very binary check. Either the person has those skills or doesn't, but several beginners don't add up to an expert.

Skill Gap Analysis Results

You've seen me make some bold statements about how data warehouse teams are not data engineering teams. Now you're going to see why. If you have a team of DBAs, you will have several critical columns with nobody listed or one person if you're lucky (Figure 3.2).

More than likely, those columns concern distributed systems and programming. Without these skills, data pipelines don't get created. This gap is how big data projects fail.



	A	B	C	D	E	F	G	H	I
1		Distributed systems	Programming	Analysis	Visual Comm.	Verbal Comm.	Project Veteran	Schema	Domain knowledge
2	Mohamed							X	
3	Gabrielle			X					X
4	Mateo			X					
5	Joseph							X	
6	Feng			X					
7	Adam			X					
8	Gabriel			X					
9	Jack			X					
10	Riya			X					
11	Juan			X		X			
12	**Total**	0	0	8	0	1	0	2	1

Figure 3.2: A gap analysis for an average DBA or data warehouse team with a simple analysis use case

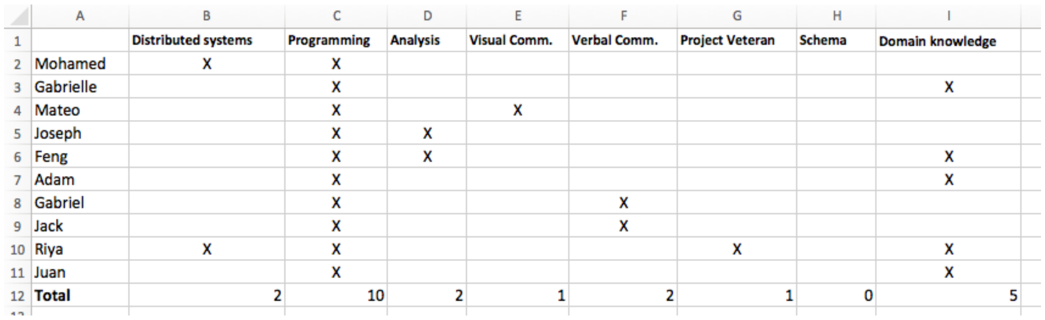
Retail Woes

A very large retailer started their big data journey and assigned the project to their data warehouse team. The team didn't have anyone with programming or distributed systems skills. They decided to overcome this shortcoming by purchasing an off-the-shelf big data solution that said it didn't require programming skills.

The off-the-shelf solution didn't solve their need for skill in distributed systems. The team didn't understand how to create the system and how the data needed to flow through it to create business value.

The project eventually exhausted its budget and, worse yet, had little to nothing of value to show for the million dollars and months of work.

Let's take another example. Say we just took our software engineering team and didn't make it multidisciplinary. You may have several different low or absent skills—in this case, distributed systems, analysis, visual communication, verbal communication, project veteran, and schema (Figure 3.3). This scenario is another way that big data projects fail; they just fail later on when you're creating an enterprise-ready solution.



	A	B	C	D	E	F	G	H	I
1		Distributed systems	Programming	Analysis	Visual Comm.	Verbal Comm.	Project Veteran	Schema	Domain knowledge
2	Mohamed	X	X						
3	Gabrielle		X						X
4	Mateo		X		X				
5	Joseph		X	X					
6	Feng		X	X					X
7	Adam		X						X
8	Gabriel		X			X			
9	Jack		X			X			
10	Riya	X	X				X		X
11	Juan		X						X
12	**Total**	2	10	2	1	2	1	0	5

Figure 3.3: A gap analysis for an average enterprise software engineering team

Other times you won't have any zeros. Congratulations! Or maybe not. I've found that teams overestimate their abilities and skills. Take another honest look and make sure you aren't deluding yourself. If you have a gap, you need to address it right away. Hiding weak spots or lying to yourself won't make the problem disappear.

You want to assess risk and any changes to the team that might be necessary. At this point, I ask the following questions:

- What is your project or team mostly doing?
- Are you primarily creating data pipelines consumed by other teams?
- Are you mostly creating reports or analyzing data?
- How complex is your use case?
- Are you doing real-time or batch processing of data?
- Is the team serving as the hub of data and system information for the rest of the company?

Answering these questions gives you insight into where your team should be strongest. If your team is mostly analyzing data or creating reports, you'll need a few programmers, but primarily people with analysis skills. If you have a complex use case, you'll want more programmers and more veterans. If the team serves as the hub, you'll want more practitioners of visual communication, verbal communication, schemas, and domain knowledge.

Some of these skills are impossible or difficult to hire. For example, you'll have difficulty hiring someone with domain knowledge of your company's systems. You'll prioritize knowledge transfer from your team members with domain knowledge to your veterans or other team members in these situations.

What I Look for in Data Engineering Teams

I want to share some of the things I've seen successful data engineering teams do and how they act. These are some of the critical elements that make the team successful.

I look for generally intelligent people who know how to work in a team. The skills needed for the team probably won't exist in a single person. A data engineering team is a multidisciplinary group. All the skills are spread among different people and in different quantities. The individuals will need to know how to work together to create a data pipeline.

Most teams don't need as much interaction as a data engineering team. You need people on the team who can communicate verbally and visually.

As part of working together, each team member needs to have a very accurate view of themselves and the rest of the team. They need to know how their skills complement one another and are used in the team. The multidisciplinary nature of the team means that members will need to use one another's skills. This interdependence requires a great deal of honesty as the individual looks over their skills and the skills of other team members. Such an internal assessment tells them how to use others' complementary skills.

I look for teams where each member understands what the company and team are doing. You'd be surprised how often the entire team isn't thinking or doing the same thing. This misunderstanding comes from a general lack of understanding of the company's overall goals and mission. These misunderstandings cost you a few hours or a day with small data.

These misunderstandings cost days, weeks, or months when you're doing big data. The stakes and complexity are so high that the team can suffer far more wasted time.

Diversity in Teams

Having taught extensively, I can say a diverse team performs much better than a uniform team. Monocultures lead to groupthink, which leads to a less innovative team because everyone has similar ideas and thoughts. So the data engineering team must be diverse in many different ways, from educational discipline to gender, ethnicity, and socioeconomic backgrounds. This diversity is how teams become more innovative and perform at their best.

When I'm teaching, I've noticed that members of a uniform team often have the same misunderstanding or inability to understand a subject. They have the same or similar ideas about execution. A diverse team has some people who understand and others who don't. The people who understand help explain the subject to those who don't. A diverse team also has more ideas about execution and can choose the best one.

Diversity while teaching

I love working with diverse teams and watching how they perform differently. I was teaching a difficult, advanced course to a diverse team that included students at different levels of understanding. During the course lecture, exercises, and even breaks, the students would help each other and explain the concepts to the other members. I was happy to see this because I knew this behavior would continue long after leaving.

A team that lacks diversity rarely has this ability. Typically, they have the same misunderstandings and can't help each other.

Operations

I'm often asked whether a data engineering team also should be engaged in operations.

I'll start with an indirect answer. I think that managed services pay off, and we shouldn't be in the operations game or should use managed services as much as possible. We should be asking ourselves why we're still in the operations business at all with the cloud.

That said, the cloud isn't a possibility for everyone. You might have to do operations on an on-premises cluster.

I've seen all different models for handling operations, and I haven't seen one work better overall than another. Possibilities include:

- Data engineering in one team and operations on a different team
- DevOps models where one team does both the engineering and operations
- Data engineering and final-tier support in one team and lower-tier operational support in another team

You don't want to adopt a throw-over-the-fence model, where the data engineers create a prototype-level system and hand it over to operators to implement in production. Historically, such a model leaves the data engineering team so isolated from production issues that they don't know how their code is acting—or acting up—in production.

Some big data technologies aren't built with operations in mind. Sometimes, the resulting pipelines are operationally fragile or require massive knowledge. Some technologies require an enormous understanding of the technology and the code that you need DevOps from the beginning.

When I teach classes on the more complex technologies, I give operational requirements and programming requirements equal weight. The team is encouraged to use the DevOps model.

Choosing an operations model

Some technologies require a DevOps model for deployment. One of those technologies is Apache HBase. When I wrote Cloudera's HBase class, I interviewed everyone I could talk to about HBase. One thing stood out: You need to have a DevOps model. To write the code, you needed an understanding of the operations side. To support the operations, you needed a deep understanding of the code. I wrote the class expecting teams to run DevOps and taught them how it should be set up.

For other technologies, the answer is not as clear-cut. My general thought in such situations is that most developers lack the Linux, networking, and hardware skills needed for DevOps. I've watched too many developers struggle to make basic BIOS changes, for instance.

Quality Assurance

Another common concern is how a quality assurance (QA) team works with a data engineering team.

There is a lack of automated tools for doing QA on big data pipelines. When teaching QA folks, I'm often asked: How do I test this thing? The answer is complicated. It involves manual scripting and writing unit tests.

Most QA teams lack the programming skills to create these scripts and tests. So the data engineering team will have to write many of these tests to allow the QA team to automate their testing. That means that your QA team needs Quality Assurance Engineers who can write this test code.

My recommendation is that data engineers should be writing unit tests no matter what. Usually, the tests and bindings the QA team needs are different. Therefore, time needs to be allocated for the data engineers to work with the QA team.

CHAPTER 3

Data Engineers

What It Looks Like When a Team Is Missing

Data teams require all of their parts to be complete to succeed. When one of the subteams of a data team is missing, the other teams suffer.

Often, organizations or team members don't understand what's happening when a team is missing. They blame themselves or the technology for their issues. In these situations, though, the problems are much more profound and require more fundamental changes to fix. I wrote Data Teams to help management understand and uncover how successful data teams work.

A common misconception with data teams is that you need only the data science team to do everything. The reality is that you need three different teams to succeed with big data. A healthy, effective data team consists of three subteams: Data Science, Data Engineering, and Operations (Figure x.1). Each team compliments and fills in the gaps of the other teams.

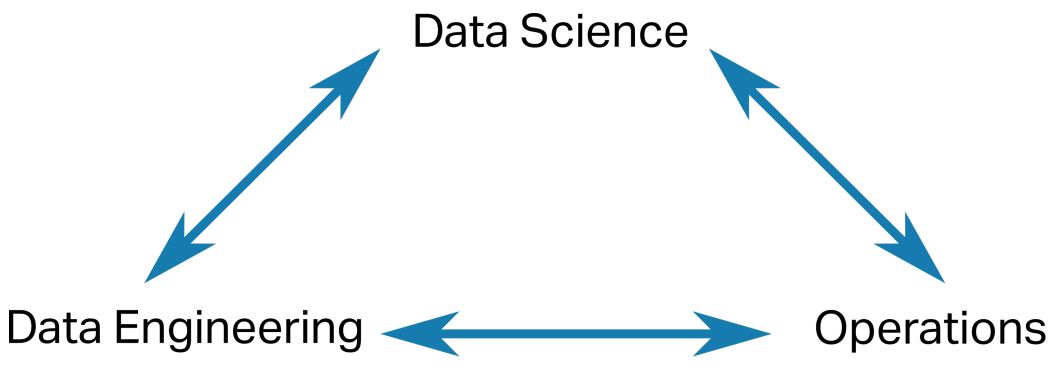


Figure x.1: A complete data team

- The data science team is responsible for doing advanced analytics. This includes creating machine learning models.

- The data engineering team is responsible for the software engineering and architecture required to run analytics at scale. Data engineers are the creators of the data products that the rest of the organization uses.
- The operations team is responsible for keeping everything running smoothly. This day-to-day operational excellence allows business and end customers to rely on the system.

Given the need for all three teams, what does it look like when one is missing? Will you see your organization in this mirror?

Missing Data Engineering

When you lack the data engineering team (Figure x.2), you lack the software engineering rigor that needs to be part of big data projects.

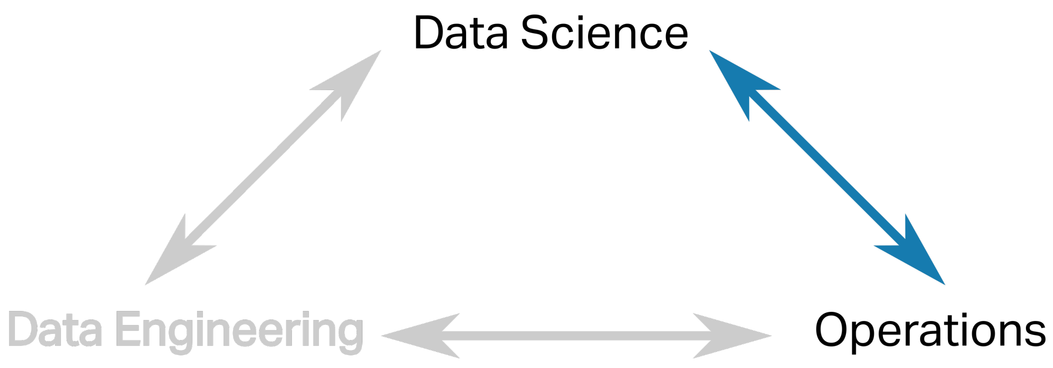


Figure x.2: A data team without data engineering

A lack of data engineers generally means that the data scientists act as data engineers. But data scientists lack the software engineering skills needed for large-scale projects. The manifestation of their lack of software engineering skills leads to code and models that are highly inefficient or not robust. Data scientists also make their own technology and architectural choices in such environments. These choices lack a sound engineering basis and often use the wrong technology for the job.

For the operations team, a lack of data engineers leaves them with code that isn't operationally sound. The data scientists' code has trouble at scale, takes far too long to run, or doesn't run at all. The operations team spends its time plugging the leaks caused by poor coding instead of improving operational efficiency.

Missing the data engineering team is the most common mistake organizations make. They make this mistake because industries put an undue focus on data scientists and don't fully understand their abilities and limitations. Even after a data engineering team is put in place, it will have to spend a significant amount of time dealing with the technical debt created by the data scientists.

Missing Operations

Data teams without an operations team (Figure x.3) tend to create systems that customers–whether internal or external–can't rely on. Whenever these customers use the system, they wonder whether the system will work this time or how long it will be down. The adoption of the system is severely inhibited because of constant operational issues.

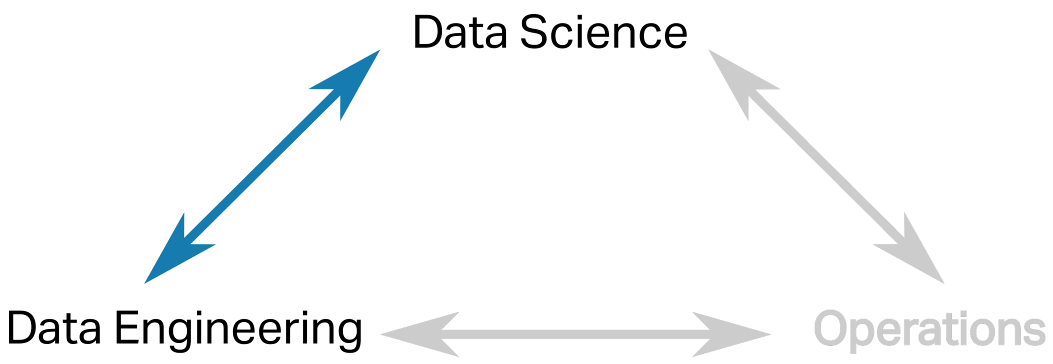


Figure x.3: A data team without operations

A lack of operations means, in practice, that the data engineers act as the operations engineers. Often, data engineers aren't well-suited to the job. They will take longer and create a less stable system than an operations engineer would. A lack of operations forces the data engineer to put their time into constant operations issues instead of writing code.

The notable exception, where an organization can get along without a formal operations team, are organizations that practice DevOps. In this case, the data engineering team is doing both software engineering and operations. Organizations looking to do DevOps with big data systems must understand the operational rigors placed on a team that does both the coding and operations.

Poor operational excellence inhibits the data science team too. A model with poor uptime or performance means the model isn't performing at the level expected. For example, the model may fail so often that the end customer rarely sees the model's results, and the fallback option is always used.

A missing operations team is relatively common. The problem usually comes from misunderstandings about the continued need–intensified by the greater scale of the Bit Data projects–for operational excellence when dealing with big data.

Missing Data Science

Data teams without a data science team are limited in the value and sheer complexity of analytics that the team can create (Figure x.4). The missing data science team means that the data engineering team or a data analytics team is trying to handle advanced analytics.

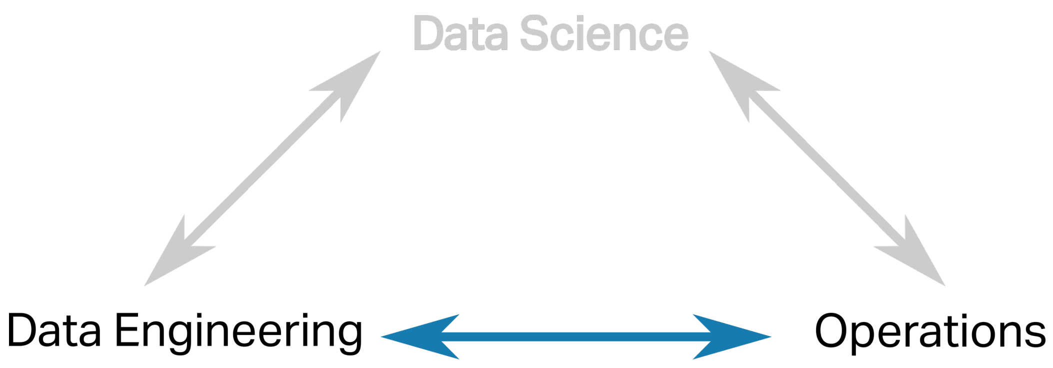


Figure x.4: A data team without data science

Although your data analytics team are not members of the data team, they may have the mathematical background to create advanced analytics. However, they will lack the programming and technical skills to carry out the project. Many data analysts have just rudimentary SQL skills or GUI analytics programs; the next level of analytics requires intermediate-level programming skills. The lack of programming skills keeps data analysts from doing data science.

Some data engineers have a mathematical background, but most have a computer science background. Their lack of a hard-core mathematical and statistical experience limits the complexity of their analytical skills.

Although it's comparatively rare to lack a data science team, I question the value of creating data teams–with all of their related expenses–without ensuring that the last and highest layer of value creation is present: the data science team.

What To Do?

You might have realized that your team or organization is missing one of the teams I've talked about. You might be experiencing the issues first-hand as the team manager or as a business customer who doesn't have the correct data products. I want you to know these problems don't go away on their own, and it will take a concerted effort to fix both the pre-existing and the resulting issues.

I invite you to read Chapter 10 of Data Teams, "Starting a Team." This chapter goes through the step-by-step process of establishing one or all of the data teams. I also encourage you to read Chapters 3-5, where I go in-depth into what data science, data engineering, and operations teams do and what skills they require.

What Is a Data Engineer?

A data engineer is a programmer specializing in creating software solutions around data. Their skills are predominantly based on Spark and the open source big data ecosystem projects. They usually program in Java, Scala, or Python. Finally, they have an in-depth knowledge of creating data pipelines.

Other languages

I'm often asked about other languages. Let's say a team doesn't program in Java, Scala, or Python. What should they do? The answer is very team-specific and company-specific. Do you already have a large codebase in another language? How difficult is it to use the Java bindings for that language? How similar is that language to Java?

The data pipelines that data engineers create are usually reports, analytics, and dashboards. They create the data endpoints and APIs for others to access data. Deliverables could range from REST endpoints for accessing smaller amounts of data to helping other teams consume large amounts of data with a big data framework. More advanced examples of data pipelines include fraud analytics or predictive analytics.

The field of big data is constantly changing. A data engineer needs to have in-depth knowledge of at least ten technologies and know at least 20 more. You might have read that and thought I was exaggerating or didn't believe me. Based on my interactions with companies, that is precisely what's needed.

I'm often asked how it's possible for a general-purpose software engineer to know so many different technologies. It isn't. It's possible for generalist software engineers to know all of these technologies only once they start specializing. However, you can expect this from a software engineer specializing in big data technologies.

What I Look for in Data Engineers

Once again, I want to share some of the traits I've found in excellent data engineers. Not every data engineer possesses every one of these traits, but good ones will have several.

I can't stress enough how important it is for a data engineer to have a strong programming background. data engineers are commonly more midway to senior in their careers. Those fresh out of school usually have a Master's degree or above in Computer Science focusing on distributed systems or data. I have seen some exceptionally bright junior engineers make outstanding contributions to the team.

This observation will sound odd, given how much I talked about the importance of programming, but the best data engineers are bored with just programming. That means that they've mastered or nearly mastered programming as a discipline. Writing another enterprise system or small data project doesn't hold much interest. As a result, they've started to cross-train into other fields. These could be related to programming, like data science, or unrelated, like marketing or analysis.

Data engineers are bored with creating small data systems because they aren't as big or complex as the systems they'd like to design. The engineers' main driver is their desire to create data products that everyone can use.

This desire comes out of their shared love of data. You might know a software engineer who loves coding or maybe even a language. They are happiest when coding. Data engineers love both coding *and* data—if there isn't a love for data, there's at least an interest in it. I've found that this interest distinguishes the great data engineers from the good data engineers.

They use data because they are inherently curious about what is happening and why. They use their data to either prove or disprove their hypotheses.

I don't focus on what technologies a data engineer knows. I focus on their understanding of systems, mainly distributed systems. They obviously need to know some big data technologies and APIs. However, learning APIs or other technology is much easier once they know big data systems' basic architectural and design patterns. A data engineer who has shown they can learn some big data technologies is likely to be able to learn other technologies.

I see this adaptability all the time when I train a team that is already working with big data technologies. The team learns more from the training because they're not starting from scratch. They catch on to the concepts quicker because there are similarities to their other big data technologies.

Qualified Data Engineers

I often talk about qualified data engineers. These engineers have shown their abilities in at least one real-world big data deployment. Their code is running in production, and they've learned from that experience. They also know 10 to 30 different technologies, as I've mentioned.

A qualified data engineer's value is to know the right tool for the job. They understand the subtle differences in use cases and between technologies. They're the ones you rely on to make the difficult technology decisions.

Not Just Data Warehousing and DBAs

Sometimes companies take their data warehousing team or DBAs and call them data engineers. People with data warehousing and DBAs skills do have a place on the data engineering team, but they aren't data engineers.

A data engineer needs extensive skills in programming, distributed systems, and big data. Most data warehouse teams and DBAs lack the programming skills necessary. For those skillsets, everything has to be in SQL. SQL is good, but it can't do everything. If a job can't be done with SQL, DBAs can't do the job.

When a team has the wrong kind of engineers, it starts to stagnate. It will never be able to move up to more complicated use cases. I call this "getting stuck at crawl." The team lacks the skills to move beyond the crawl phase (discussed in Chapter 7, "Running Successful Data Engineering Projects") and into more complex phases

Teaching a data warehouse team

I was teaching a data warehouse team who kept on asking very advanced questions. We covered the basics, but a particular student asked me how to do everything in real-time (which would be very difficult and advanced). I'd give them the answer, and I'd get a blank look each time. Having taught extensively, I know that a blank look means they didn't understand the answer. I couldn't make the answer any simpler for this team because they were asking advanced questions.

I overheard the student asking advanced questions while talking to a coworker, and the coworker was asking the same questions the student had asked me. He replied that those use cases were impossible and implicated the technology as to why it couldn't be done. The technology was perfectly capable of doing it; the student just wouldn't admit that they personally couldn't do it or understand how to do it.

Ability Gap

As a trainer, I've lost count of the number of students and companies I've taught. One thing is common throughout my teaching: an ability gap. Some people simply won't understand big data concepts on their best day.

Most industry analysts usually talk about skills gaps when referring to a new technology. They believe it's a matter of an individual simply learning and eventually mastering that technology. I think that too, except when it comes to big data.

Big data isn't your average industry shift. Just like all of the changes before it, it's revolutionary. Unlike the previous shifts, the level of complexity is vastly greater.

This gap manifests itself in the individuals and students trying to learn big data. I'll talk to people whom I've figured out that they have no chance of understanding the big data concept I'm discussing. They'll keep asking the same question repeatedly in a vain attempt to understand. They've simply hit their ability gap.

Big data is hard. Not everyone can be a data engineer. They may be part of the data engineering team but not be a data engineer.

Yes, this is harsh

I know when a person has an ability gap. It's a person struggling valiantly to understand big data but who, in the end, won't ever be able to understand it well enough to create a system. They'll be able to understand the concepts but not understand them deeply enough to create a data pipeline. I've tested this theory out for more extended periods with students. No matter how much time or effort, they won't be able to get it. Becoming a data engineer isn't a good use of their time.

As a manager or team lead, you need to make these difficult judgment calls. I've found that the person themselves can't make the call and can't figure out why they're not getting anywhere in the project. If you're experiencing a project with little to no progress, your team may be hitting this issue.

CHAPTER 4

Productive Data Engineering Teams

Advice for Small Teams and Startups on Data Engineering

Much of my writing is geared toward large companies and teams. Small data engineering teams require different tactics. How should a startup or small data engineering team in a big company be set up and work? What, if anything, should be done differently?

Your First Data Engineer

Your first data engineering hire is a crucial decision. This person gives the team its starting direction because they're making many of the initial technology decisions. Some of these you can change later, and others will be pretty difficult to change.

As you hire more engineers, your first hire will be interviewing and evaluating the rest of the team. Conversely, your interviewees will be evaluating the technology decisions and directions you've undertaken. The environment erected by the first hire will either repel or bring in other competent data engineers. Weak engineers often hire other weak engineers.

The Effects of a Bad Hire

The obvious most significant difference between small and large teams is sheer size. A single wrong hire can disproportionately drag down the team. For example, if a team has three total people and 1 of them isn't competent, you're getting only 66% of the team's productivity. In my experience, the decrease in productivity is more like 50-60% because the incompetent person ties up the other people's resources. In big data, tasks are more complicated, and the person can't just do a web search on every question.

Hiring competent people is crucial early on. You might hit an ability gap and need to make some changes.

Looking Internally

Some companies try to look internally at the people they already have. These companies tend to put their data warehouse team, DBA, or other SQL-focused people on big data initiatives. I've talked about why this is a bad idea in this book. If you're dealing with big data, an SQL-focused person is best employed as part of a larger data engineering team. With a small team, that decision has an even more profound effect. The lack of various skills and of other team members to rely on will cause either an outright failure or a severely underperforming big data initiative.

If you have to promote from within, a better bet is to look internally at your software engineers. Look for people who have experience with multithreading. Hopefully, they'll have the required programming and perhaps the distributed systems experience needed.

No matter what, these people won't have the skills you need for big data technologies. You'll need to give them the resources and time to learn the new technologies. Too often, small teams are in a hurry and skip the necessary skill acquisition. Overlooking this step leads teams to get stuck.

Too Many Junior People

Some companies try to solve their small team woes by throwing many junior engineers at the problem. A group of 10 novices doesn't average out to 1 expert; a group of 10 novices averages out to 1 novice ten times with the same knowledge and skill levels. The junior staff's skills overlap rather than complement each other. This lack of experience is a killer to productivity on data engineering teams.

When novices design big data projects, the magnitude of the mistakes is the issue. A bad design doesn't cost a few days to rewrite. They can take months to fix and rectify.

I've worked with junior designs and code. Junior engineers often copy other peoples' designs and don't understand the differences or why they might not work for their use case. A junior person can understand the what but not the why or whether the design is correct. There are problems with the plans, and you might not see the issues until you're in production.

As I say elsewhere in this book, a project veteran is crucial to a data engineering team's success.

Operations

When you only have a few people, you need to make every person count. As soon as you go to production, you'll have to figure out who should handle operations. I highly recommend that small teams use managed services on the cloud. Many of the popular open source technologies are available through managed cloud products.

Relying on the expertise of a third-party vendor saves you from hiring an operations person. The majority of your operational load can be offloaded onto the managed service. Hire a person with a DevOps background to set up both the development and operational tasks.

Outside Help

Sometimes small teams or companies eschew outside help. But a small team needs to take advantage of whatever resources are available. Getting outside help is a great way to accelerate your team's productivity. You can exploit outside help to:

- Provide architecture reviews
- Interview and hire the right data engineers
- Code data pipelines

The Chicken-And-Egg Problem of Small Teams

What do you do when you have a data scientist but not a data engineer? Or the reverse: a data engineer, but not a data scientist? Remember that a data engineer has a difficult, if not impossible, task trying to be a data scientist. Likewise, a data scientist can't do the job of a data engineer.

You'll have to understand each person's limitations and decide what they can do and what's well within their limitations. Otherwise, you'll have to do significant rewrites to their products.

Some companies try to find that legendary person with data engineering and data science skills. These people do exist, but they're few and far between. They were also expensive. I suggest using outside help to fill these gaps until you hire someone.

Hiring Your First Data Engineer

All of this advice feed into hiring your first data engineer. Ideally, you'll hire someone with previous experience as a data engineer. Other desirable traits include:

- Has put a big data project into production
- Was a senior software engineer before becoming a data engineer
- Has an established pedigree from another competent company
- Has experience in the same industry or domain as your company to leverage direct previous experience

When interviewing your first hire, you might not be able to identify a suitable candidate from a bad one on the technical side. Focus on demonstrable skills during the technical interview.

You also need to be clear about what you're trying to hire. A data scientist is not a data engineer. If you don't have a data scientist already, hire a data engineer first. You can buy yourself time by getting started on the necessary data engineering tasks so that the data scientist has data products and data infrastructure to use once they're hired.

Every team has to start with their first hire. Getting your first hire right is crucial. It sets the pace for the rest of your project.

Themes and Thoughts of a Data Engineering Team

Data engineering teams have to change their thinking when dealing with big data. In my courses, my chapter on distributed systems is called "Thinking in Big Data." It's an odd name for a chapter because most teams don't realize that they need to change their thinking about data and systems to succeed with big data. They need to think about how data is valuable and the scale at which information is coming in.

When thinking about scale, I encourage teams to think in terms of:

- 100 billion rows or events
- Processing 1 PB of data
- Jobs that take 10 hours to complete

I'll talk about each one of these thoughts in turn.

When processing some data, in real-time or batch, you need to imagine that you're processing 100 billion rows. These can be 100 billion rows of whatever you want. Planning for data at that scale affects how you think about reduces or their equivalent in your technology of choice. If you reduce inefficiently or when you don't have to, you'll experience scaling issues.

When you think about amounts of data, think in terms of 1 PB. Although you may have substantially less data stored, you'll want to think about your processing in those terms. As you're writing a program to process 100 GB, you'll want to make sure the same code can scale to 1 PB. One common architectural manifestation of the difference comes if you plan to cache data in memory for an algorithm. While that may work at 100 GB, it probably will get an out-of-memory error at 1 PB.

Finally, when you are thinking about the duration of processes, I encourage teams to think of them running for 10 hours to complete. This consideration has quite a few consequences.

The most significant consequence concerns coding defensively. A job should be coded to check assumptions whenever possible to avoid an exception from exiting a job. If you have an exception at 9.5 hours into a 10-hour job, you have to fix the error and rerun the 10-hour job.

Here's a typical example of the errors that turn up in big data: you receive unvetted data in a string-based format such as JSON and XML, and you are trying to extract data in a specific format. At some point, the program might cast a string to a number. If you don't check that string beforehand with a regular expression, you could hit an exception. And then, you need to decide what to do about data that doesn't fit the expected input.

Most frameworks don't handle data errors by themselves. This error handling is something the team has to solve in code. Some common options are to skip the data and move on, log the error and the data, or die immediately. The decision depends heavily on your use case. If losing data or not processing every single piece of data is the end of the world, you'll have to fix any bad data manually.

Every engineering decision needs to be made through these lenses. I teach this to every team, even if their data isn't at these massive levels. If they truly are experiencing big data problems, they will hit these levels eventually. You want to be able to keep going without changing your code. Even using a big data framework, teams can still write code that doesn't scale or scale well.

Another important theme is that a qualified data engineer needs to know around 30 different technologies at varying proficiencies to choose the right tool for the job. These range from batch frameworks and real-time frameworks to NoSQL technologies.

A final theme is that a subtle nuance in a technology can make a world of difference in the use case. That's the difference between a data engineer and a software engineer. A data engineer can see these subtle nuances and what they mean to a system much faster. A software engineer can't.

Hub of the Wheel

Sometimes I teach at large enterprises that disagree that there should be a separate data engineering team. They're usually thinking just about the technical skills required. Thinking of a data engineering team as the hub of a wheel puts a data engineering team in a completely different light (Figure 5.1). This viewpoint shows how a data engineering team becomes an essential part of the business process.

When you ask CxO-level executives what they would put at their organization's "hub" or center, they rarely consider the data engineering team. But with the importance of data, this is where modern companies are headed. As the other parts of the organization begin to consume the data or use the data pipeline, they will need help. This help comes from an ability gap, a team with no programming skills, or another programming team that needs help.

Nor does a data pipeline stay at its initial use after its first release; it almost always grows. The pipeline releases pent-up demand for data products. New datasets and data sources will get added, requiring further processing and consumption of data. If you allow a complete technical free-for-all, you will have issues. Often teams that lack qualified data engineers will completely misuse or misunderstand the technologies.

As the hub in the wheel, the data engineering team needs to understand the

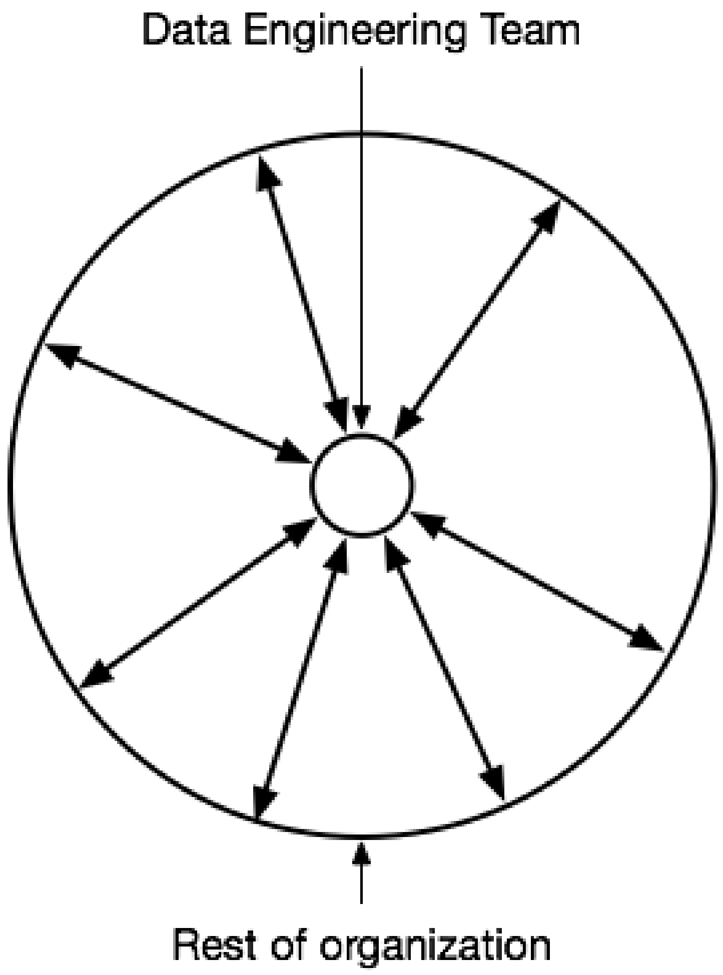


Figure 5.1: The data engineering team as the hub of data pipeline information for the organization

whole data pipeline. They need to disseminate this information to other teams. They need to help other teams know what data is available and the format. Finally, they have to review the code or write it for other teams.

Organizations that fail to establish a data engineering team create an inefficient use of data or hinder the use of data for decision-making. There is a massive demand for data and data products in these organizations, but they are either completely lacking or low-performing. As a direct result, the organization never fully utilizes its data for decision-making.

Other times, companies balk at training or hiring Software Developers who specialize as data engineers. The companies don't understand how much their qualified data engineers will need to help other teams or how much time that will take.

How to Work with a Data Science Team

Before I talk about their relationship to data engineering teams, I want to briefly define a data scientist and data science team.

A data scientist is someone with a background in math and probability who also knows how to program. They often know big data technologies to run their algorithms at scale.

A data science team is multidisciplinary, just like a data engineering team. The team has the variety of skills needed to prevent any gaps from surfacing. It's unusual to have a single person with all of these skills, and you'll usually need several different people.

A data engineer is different from a data scientist in that a data engineer has to be a much better programmer and distributed systems expert than a data scientist does. A data scientist has to be much better at the math, analysis, and probabilities than a data engineer.

The teams are more complementary than heavily overlapping. That isn't to say there isn't some crossover, but my experience is that data scientists usually lack the hardcore engineering skills to create big data solutions. Conversely, data engineers lack the math background to do advanced analysis.

Figure 5.2 shows how tasks are distributed between the data science and data engineering teams. I could, and have, talked about this diagram for an hour.

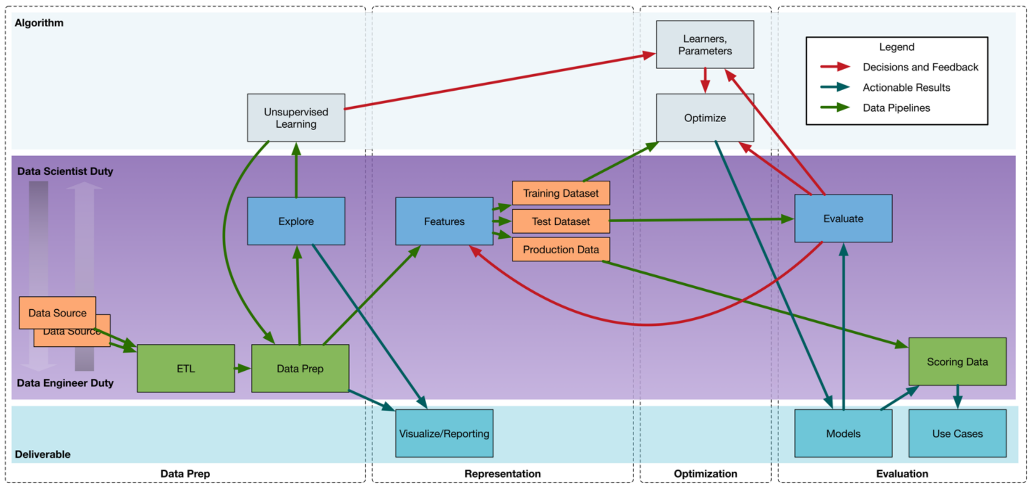


Figure 5.2: How data scientists and data engineers work together. Based on Paco Nathan's original diagram. Used with permission.

Tasks closer to the top of the purple center panel are associated more with the data science team, while tasks closer to the bottom are associated more with the data engineering team. Note that very few responsibilities are assigned solely to one or the other.

There are a few points I want you to take away from this diagram. A data engineering team isn't just there to write the code. They need to be able to analyze data, too.

Likewise, data scientists aren't just there to just make equations and throw them over the fence to the data engineering team. data scientists need to have some level of programming. If this becomes the perception of reality, there can be a great deal of animosity between the teams. Figure 5.3 shows how there should be a high bandwidth and significant level of interaction between the two teams.

Ideally, there's a great deal of admiration and recognition of talent between the two teams. Each acknowledges the other's abilities. It's magic on both sides. Data scientists have mathematical magic, and data engineers have programming magic. The two teams should form a symbiotic relationship rather than an adversarial one. Neither side will completely understand what the other did. That's completely understandable, given each team's skills and abilities. Data science teams have a tradeoff to make. They need to be agile in their analysis.

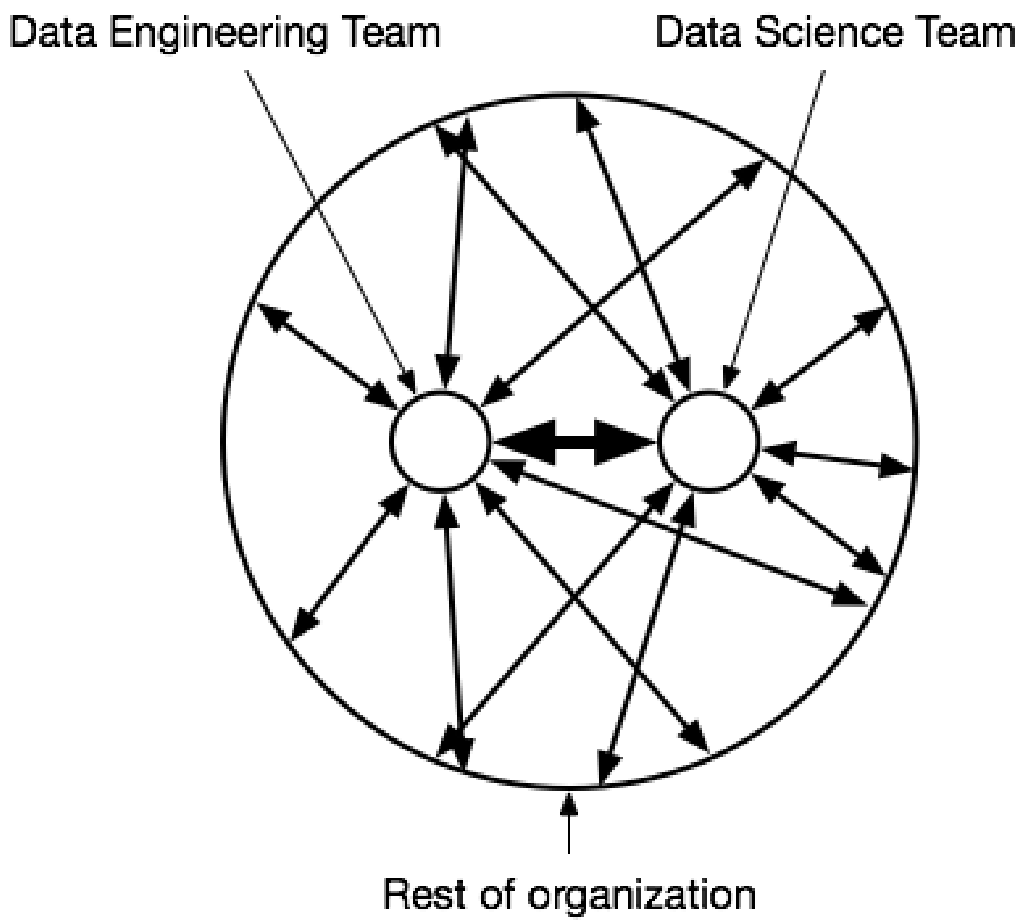


Figure 5.3: How a data engineering team and data science team should interact with each other and the rest of the company

Data scientists tend not to use software engineering practices because they believe it will weigh them down. Data engineers come from hardcore engineering disciplines and use software engineering best practices. The two teams need to strike a balance between the agility craved by the data scientists and sticking to the excellent engineering practices upheld by the data engineers.

The majority of data scientists come out of academic environments where they haven't needed hardcore engineering and will need help understanding how and why a robust engineering process is necessary for a production environment. As I'll discuss in upcoming sections, the data scientists will need to learn how much time and effort to allow for data engineering processes. Both teams need to know how to compromise.

Sometimes data science and data engineering teams are together in the same team. The size of the teams usually depends on whether the entire data operation is small or is big enough to split into separate teams. Productivity depends on several factors, including corporate politics, personalities, and how well each side understands the importance of the other side.

Together or separate?

I find the usual reason for data engineering and data science being together versus being separate comes down to how they were started. Usually, the data science team comes out of the organization's analytics or business intelligence side, and the data engineering team comes from its engineering side. For the two teams to be merged, they'd have to cross organizational boundaries, which doesn't happen very often.

One suggestion that I don't see used often is to pair-program. I'm not saying everything should be done as pair programming, but I suggest that teams do so when it makes sense. A data engineer and a data scientist would program together. The code would be checked as it's written for quality by the data engineer and correctness by the data scientist. The resulting code is much better, and the knowledge is transferred immediately.

Why Some Data Science Teams Don't Think They Need Data Engineering

Some of my most exciting consultations come when I help data science teams who don't think they need data engineering. I've compiled a list of some of the more common reasons why data science teams believe they don't need data engineering and why those reasons might not be valid.

Data science teams must have data engineering because the data scientists might just be getting by or severely underperforming. The results from missing the data engineering team are not impressive and leave much to be desired. Commonly, data scientists will create technical debt that data engineers will have to spend time fixing.

Lack of Understanding

There is a total lack of understanding of what data engineers do for some data scientists. This lack of knowledge comes from a cursory understanding of programming and maybe some distributed system the data scientist has encountered. Ignorance leads to a "how hard could it be?" question that downplays the complexities that data engineers hide from data scientists.

To help data scientists understand the various between a data engineer and a data scientist, I created some visualizations that clearly show the differences.

Repeatable Data Science

Some data science is repeatable. I mean that automation and consistent data products are being created and maintained. On the other hand, some data science is ad hoc and not repeatable. In these scenarios, every project is started from scratch, and once the project is done, it is completely discarded.

For ad hoc projects, there's no big engineering onus. The projects live for only hours, days, or weeks. There's no real need for any long-lived planning. But I'd argue that organizations lose much of the value of data science when everything is so ephemeral.

When ad hoc organizations transition to long-term projects, they hit the brunt of their engineering mistakes. They've been able to escape the data engineering rigors of projects that need to be repeatable and run consistently. They find out the hard way that data engineering isn't over-engineering; it's making sure that the data products are maintainable. Creating repeatable data products requires data engineers.

There's No Scale…Yet

Sometimes organizations start out with small or medium data and don't have to deal with scale issues (count yourselves lucky). They've been able to get by with Excel, single processes, or waiting longer for results. The transition to big data and scale catches them by surprise.

The transition to big data technologies comes with a significant increase in complexity due to the need for distributed systems. At first, the data scientists think they can handle the growth. It should become quickly apparent that they can't deal with the complexity increase and that they need data engineers.

Creating scalable data products requires data engineers.

What The Problem Looks Like and What to Do

If your team is experiencing one of these problems, it will look like the data science team is stuck. They'll spend a week on something that seems like it should take hours or a day. They'll spend hours googling or searching on Stack Overflow for answers (these sorts of solutions aren't findable on Google or Stack Overflow). The data scientists simply won't be technically competent enough to realize the issue. These sorts of problems fall right into the wheelhouse of data engineering.

Managers and data scientists will need to take an honest look at the team's productivity and skills. They more than likely will need data engineers and need to establish a data engineering team.

What Happens When Data Science Teams Add A Data Engineer

Organizations are gradually getting the message about the critical nature of data engineering. Data science teams are getting that message too[1]. Sometimes, that message gets muddled, and data science teams think they just need to add a

[1] I co-wrote this section, originally as a blog posting, with machine learning expert Mikio L. Braun.)

data engineer or two to their team. In their mind, this solves the problem, and they can go back to business as usual. I'd like to share our experiences when this happens and why this isn't the right course of action.

Buy-In

The core issue is that data science teams don't fully believe that data engineering is critical to success. Instead, they cling to a "there I did it" or "there I fixed it" mentality. Naturally, their minds are on the actual data science work, and they often don't have enough knowledge to fully understand the challenges of data engineering. In addition, the data scientists often perceive the amount of time needed for data engineering compared to the data science side of things as a problem, again without fully understanding why. I recommend a ratio of 2-5 data engineers per data scientist.

Hiring

Many problems trace their way back to hiring. Data scientists often hire the wrong engineers, and the situation just gets worse from there.

Hiring the wrong people can have all sorts of root causes. For example, data scientists may not believe data engineering can help or is all that difficult. Or they could completely misunderstand data engineering and have worked with the wrong kind of data engineer, further promulgating a flawed archetype of a data engineer. We've also seen data science teams change the title of the most data engineering savvy data scientist to a data engineer. Usually, this assigns the job to the most competent data scientist, but one who is still unqualified compared to data engineers.

The poor hiring decision becomes a self-fulfilling prophecy. Not knowing how to evaluate a data engineering candidate, the data science team chooses the wrong person, leading to underperformance. It is hard to learn how to do it right from that experience. This cycle repeats itself to create a strong bias.

Getting The Project Underway

The project gets underway. There are so many technologies to choose from. Too many things to be done and fixed. How should the data engineer start to make headway when they can't even understand the data scientists' needs? Projects with the wrong people end up as questions on Reddit. They usually say, "I was just hired, and I don't know what to do. Here is what they're asking for. Could you help me choose some technologies?" The responses are well-meaning but miss crucial information because the original post leaves them out. Some suggestions are flat-out wrong. The online exchange leaves the unqualified data engineer to try to implement something they couldn't understand or vet in the first place. This failure leaves the business and value creation in the same place or worse than before.

Performing Surgery

Being the first data engineer to start working with data scientists' code and architecture can be daunting. In addition, the data scientists could have created a mountain of technical debt.

Getting anywhere can be the most delicate surgery of fixing technical debt while not breaking the entire system. From a personnel standpoint, it takes a qualified data engineer even to attempt to fix the system. The task will more likely require a whole team of data engineers to make the necessary fixes and rearchitecting. As a result, you will be worse off with the wrong person than before (see the self-fulfilling prophecy previously described).

Outnumbered and Outgunned

When data engineers are outnumbered, they're often outvoted and outgunned. As a result, the issues, tasks, and challenges significant to data engineers aren't understood and considered essential by the data scientists on the team.

A data engineer's issues are perceived by the data scientists as too expensive, slowing down the data science, or over-engineering. Without a more prominent voice on the team, the data engineers can be easily overlooked or shouted down. For example, data engineers will see the issues and poor design that led to the data scientists' technical debt in the first place. The data scientists will veto the fixes or changes because they will slow the data scientists down or will perceive them as unnecessary in the first place.

Some of the worst-case scenarios are that all of the data engineer's ideas and changes are ignored while the data engineer is assigned the more menial tasks the data scientists don't want to do. It creates a poor match on both sides of the equation.

What Do The Problems Look Like?

If you're a data engineer on one of these teams, you already know what it looks like. Nothing is changing; you're frustrated and looking for a new position.

For management, this looks like you've added a data engineer who thought that the hire would fix a problem, and there's no change. You've simply added a person without fixing the deeper organizational issues that got you there. I've helped many organizations in this situation, but there isn't a one-size-fits-all fix. The initial steps start with management and organizational change, not the individual contributors.

How to Work with a Data Warehousing Team

In contrast to the data science team, there is a great deal of overlap between the tasks assigned to a data engineering and those assigned to a data warehousing team–but the skills and tools required to solve the tasks differ because of the size of the data. Often, the entire reason for creating a data engineering team and moving to big data solutions is to move off of data warehousing products.

Big data technologies can do everything a data warehousing product can do and much more. However, the skillsets are very different. Whereas a data warehousing team focuses on SQL and doesn't program, a data engineering team focuses on SQL, programming, and other necessary skills.

The data warehousing team is almost always separate from the data engineering team. Some companies rename their data warehousing team as a data engineering team. As noted, the skillsets and levels of complexity are very different. Elements from a data warehousing team can sometimes fill in skills gaps in a data engineering team; however, the data warehouse team often brings domain knowledge and analysis skills.

If you decide to rename your data warehousing team as a data engineering team or merge it into one, you must check for ability gaps. These ability gaps are a common reason why data warehousing teams have low success rates with big data projects.

How to Work with an Analytics or Business Intelligence Team

Usually, a data engineering team is creating a data pipeline that, in whole or part, will help the analytics or business intelligence (BI) teams. These teams have hit some kind of performance issue or bottleneck in their small data systems. The data engineering team creates the infrastructure to enable the analytics or BI teams to scale.

The analytics or BI teams usually lack the programming skills necessary to adapt or submit data to the data pipeline. The data engineering team must provide the programming support the other teams need.

The data engineering team should set up the pipeline and documentation to ensure that the analytics or BI teams can self-serve. They shouldn't have to get a member of the data engineering team to run a query for them. That's a waste of the data engineering team's time and doesn't engender a spirit of cooperation between the teams.

The data engineering team needs to reach out with help, but the analytics and BI teams should also learn the technologies up to their abilities. Many big data technologies offer SQL interfaces that give nonprogrammers a way to access the data without knowing how to program it. Even if the teams already know SQL, they might need to take extra effort to learn how to use the SQL interfaces provided by the pipelines.

How I Evaluate Teams

I consistently look for certain things when evaluating a data engineering team for effectiveness. I determine their probability of success given what I know about their use case, the complexity of the solution, and the organization.

Key Skills

Does the team have the right people? If you have a critical skills gap in the team, your probability of success plummets. In my experience, teams with a gap in programming and distributed systems skills have a 5% to 25% chance of success. Other skills gaps aren't as big of a hit on the probability of success.

Unit Testing

I also check whether the team is unit testing their code. Data pipelines are really complex. How do you know whether the code changes you made will cause other problems? The team will have a complex system with no guarantees that it will work. Without good unit tests, the team's productivity will grind to a halt. Unfortunately, an absence of unit testing is one of the most common failures in data engineering teams. A large portion of the blame goes to the big data technologies, which don't build unit testing in as part of project development.

Let me give you an example of why this is important. I evaluate data engineering teams on how fast they can replicate a bug. If you have a problem in your production data pipelines and your data engineer can't reproduce a bug in their IDE in 10 minutes or less, you have two problems: a production problem and a unit testing problem.

Data pipelines should be logging problems and problem data. A data engineer should be able to take that log output, put it into an already written unit test, and replicate the problem. If your team can't do that, you will forever chase and fix issues.

Another core reason for unit testing is a full-stack or integration test. If you don't have a unit test that flows all the way from the very beginning of the data pipeline to its end, you won't be able to find and fix the most challenging problems. You'll recall that data pipelines comprise 10 to 30 different technologies. Each one of those technologies has a different API for doing unit tests.

A data engineering team with comprehensive testing won't just be initially successful; they'll continue to be successful as the data pipeline grows over time.

Automation

Automation is another multifaceted and critical metric for a team. Going from small data to big data represents a significant increase in complexity. The current team is creating and dealing with that complexity as it happens. Experiencing these changes in deltas, instead of all at once, makes it easier for the current team to understand the system.

A significant difficulty arises when new people join the team. How long will it take for them to be productive? Let's say with your small data system that it took one month. With big data, you could create a system that takes six months to understand—apart from all the necessary domain knowledge.

This thought should factor into your hiring and overall system design. Are you creating a system so difficult only its designers will understand it?

How long does it take for a new person on the team to set up their development environment? Setting up a big data environment on a developer's machine isn't easy; it can take several days of trial and error. Many different processes need to run, and the steps to configure them might not be documented well. Additionally, system setup isn't a developer's core competency. Therefore, at least the initial setup should be scripted or, in my opinion, done on a virtual machine (VM) or container with Kubernetes. This setup allows developers to jump into the job with a VM already set up and running the necessary processes. More importantly, it ensures that every developer runs identical software versions.

A related issue is how fast you can deploy a fix or new code to production. How many manual steps are there? With small data systems, teams can squeak by doing things manually. Big data changes the scenario for deployment because you're dealing with 30 different computers or instances. Using the old tools and techniques just breaks down. You'll need to be able to deploy code quickly—ideally with the push of a button.

A team that lacks production automation will suffer from having 30 machines configured in slightly different ways. It will have program versions that are potentially different codebases. It will be soaking up operations time on deploying instead of fixing the problem. All these are recipes for complex production problems.

Equipment and Resources

Equipping and giving sufficient resources to a data engineering team is the key to its productivity. A common misunderstanding is that big data is cheap. It isn't. Yes, your licensing fees will drop dramatically because the software is open source, but your costs elsewhere will go up.

A common issue is that data engineering teams need sufficient hardware to develop and run data pipelines. This hardware includes their development machines. I can't tell you how often I've taught at companies where they're expecting their data engineering team to use the same hardware they used for small data development in their big data development. The hardware needs to be able to run the software for the entire data pipeline simultaneously. If it can't, you won't be able to debug an entire data pipeline. Explain that to the CxO when there's an issue in production.

Another odd but consistent issue is that developers lack administrative access to their own development machines. They will need this access to install new software.

As your data pipeline becomes more complex, you'll need to use VMs to install new software or quickly switch between the production and the latest versions for development. Again, a smooth solution to these barriers is to provide VMs to developers for development. I teach developers in my classes how to use VMs efficiently to develop code, try out new frameworks, and switch between different software versions. These tricks are what make data engineering teams productive.

Teams also need clusters. Overall, the teams will need development, quality assurance, and load-testing clusters in addition to the production cluster.

I've seen so many teams unable to rate the performance of their software because they don't have the same hardware as in production. What is the maximum number of messages, events, widgets, or whatever the software can handle or process per unit of time? The only way to know is through load testing. The load-testing and production clusters should have the exact specifications as the production systems.

You don't throw out good engineering practices when creating your data engineering team. You need to use your continuous integration server to test your entire data pipeline with unit tests more than ever. This virtuous cycle is how data engineering teams gain velocity in development.

Last, not all resources are physical. A key ingredient to data engineering resources is learning materials. Big data is constantly changing, and your team will need to keep up with the changes. You need to give the team the training they require. Getting in-person training is the cheapest way to test it out if you're starting on a brand-new technology. Few things are more expensive than finding out halfway into a project that the technology isn't a good fit or that you're misusing it. Books are another great resource, so make sure the team has access to the relevant texts.

CHAPTER 5

Creating Data Pipelines

Thought Frameworks

Thought frameworks are some decision-making techniques I teach data engineering teams. They help teams make decisions about data pipelines, write scalable code, and create good technical processes.

One framework concerns optimization in big data. Optimizations should focus on “a little of a lot” or “a lot of little.” Teams often optimize prematurely or on the wrong piece of the puzzle.

“A little of a lot” means that a small part of your job takes up your processing time. For instance, a small loop takes up to 10 minutes of a 20-minute job. That’s a great place to start optimizing.

“A lot of a little” means that a piece of code may seem too small to be worth optimizing but is used so often that the small improvement you can make will significantly impact the pipeline. For instance, you call a method that takes 20 ms to execute and is executed 1 million times during your program. That 20 ms doesn’t sound like a lot, but the program spends 20 million ms or about 5.5 hours executing the method. If you were to reduce that 20 ms to 15 ms (4.1 hours overall) or 10 ms (2.7 hours overall), you would make a huge difference in runtime.

When technology is brand new to the team, they undergo a feeling-out period. What can this technology do? How hard can I push it? What are its limitations? During this period, the team will try to see what cases a technology can be used for and how.

It’s during this learning period that I get asked very unbounded questions. Can I use technology X to do things Y? Sometimes, the question is ideal for the technology. Other times, it’s really stretching what the technology should be used for and how. I give students a two-question framework to help them figure it out: Is it technically possible? Should you do it?

"Is it technically possible?" means, can the technology do what you ask? That's usually a quicker pass/fail when you know and understand the technology. "Should you do it?" means, is it wise operationally, technically, or code-wise to use the technology like that? In other words, yes, that idea will work, but it will be an absolute abomination, and I'm going to have to take your engineering card away. Or, yes, you could write that code, but your operations person will never speak to you again.

Speaking of abominations, sometimes big data is abused with small-data use cases. For these times, I give students another two-question framework: Is it big data? Will it become big data? Using small data solutions leads to all sorts of scaling headaches. But what happens when you use big data solutions for small data? Your capital expenditures go up, your personnel costs go up, and your project times go up. Those are just some of the high points. It is such massive overkill if it isn't big data, and this isn't the time to pad your resume.

Building Data Pipelines

The primary goal of a data engineering team is to build data pipelines. A data pipeline is how your data sources move through your system. It can be done in real-time, batch, or both. Part of a data pipeline's job is to make the data ready to be processed and then expose that data.

Data pipelines are not made up of a single technology. A software engineer who knows a single big data technology is not a qualified data engineer. You need to use many different technologies to create a solution. A data pipeline needs to bring together 10 to 30 different technologies to do the job, whereas a small data solution will only need 1 to 3. This is part of what makes big data so complex.

When we add in NoSQL databases, the situation becomes even more complex. Most organizations think in terms of a single NoSQL database for all of their data. For large organizations with diverse use cases, you'll probably need several different databases that are specific to the use cases you have. Yes, that will increase operational expenses and the amount of data you're storing. However, using the correct database for each job makes the use cases perform faster or even be necessary to enable the use case

Knowledge of Use Case

In-depth knowledge of the use case is one of the critical differences between big data and small data projects. This knowledge is one of the primary ways I've seen data engineering teams fail. They're creating a data pipeline for a use case that they don't understand, that they don't fully understand, or that is changing.

Starting the creation of the data pipeline before you understand the use case leads you to waste time and money. You'll spend a month writing code that doesn't get used.

What happens when you don't have a use case

A large retail company started its big data project without a use case. They already had a data warehouse and a data warehousing team in place. They thought that going to big data was just a matter of moving from technology A to technology B, but their move to big data required more than just switching technologies.

They also thought they would find a use case along the way, as they had done with small data. They made all the wrong technology choices by not knowing the use case. By not having clear-cut goals, they accomplished none of their goals. The project was eventually shut down.

With small data, you can use virtually whichever technology you already use. You come in with a technology stack and use it for everything. For big data, almost every engineering decision comes down to the use case. If you don't know your use case, you won't be able to choose the right combination of technologies. When I teach data engineering teams, I tell them not to start looking at technologies until they've clearly identified the use case.

Choosing technologies first and then looking at use cases often leads to failure. There is a broad spectrum of possible levels of failure. The best-case failure is that the project takes longer or much longer to complete. The worst-case failure is that the use case isn't possible with the technology choices. I've seen all levels in the spectrum, from teams who've lost a million dollars and six months by misusing a technology, to those who lost a month and $150,000 by not understanding the technology's limitations.

You've probably heard the saying that an ounce of prevention is worth a pound of cure. With big data solutions, a pound of prevention is worth a ton of cure. Yes, you will spend far more time understanding the use case, but the payoff is enormous.

Another common mistake is to treat the design of a big data solution as you would for a relational database. When designing for a relational database, you're thinking about how data (tables) relate and how to normalize that data. When you design for big data, you're thinking about the use case and how data is accessed, processed, and consumed. You are entering a very different mindset and engineering focus.

When designing your data layout, failure to think about the use case also leads to poorly performing data pipelines.

It's worth noting that use cases can grow if a business has to make concessions on its original goals due to technical limitations. For example, the business initially wanted to process 20 years of data and two datasets, but technical limitations permitted only five years of data and a single dataset to be processed. Once you have big data technologies in place, those limitations are removed. The business will want to return to its original ask, and the use case will grow.

Another way the use case will grow is when other organization members hear about the capabilities and want to use them. This expansion happens when I'm training a team. They'll get email messages from other teams that have heard they're learning a big data technology and now want to start using it themselves.

Right Tool for the Job

A significant portion of a qualified data engineer's value is to choose the right tool for the job. For qualified data engineers, the choice comes down to knowing the use case. When creating a big data solution, they must choose from various tools and technologies that perform similar tasks while offering different tradeoffs and abilities.

Another value add of a qualified data engineer is to help figure out whether a solution requires big data or small data technologies. In the best case, using the wrong tool costs you hours or weeks of time; in the worst case, you waste months before learning the task is impossible to complete with that tool. Spend the money and time to hire a qualified data engineer and save yourself some heartache.

I'm trying to keep this book technology-free, but I want to give a typical example I teach teams. Let's say you need to process some data. You could use a big data SQL engine or hand-code a program in this use case. Which one should you choose? Consider these questions:

- How often will the program run? Is this a one-off call to a program, or is it run hourly or daily?
- If you were to hand-code it, what level of performance improvement would you see?
- Is there a time when the SQL would become so complex that you'd be better served hand-coding?

These are the sorts of questions I go through with data engineering teams. Sometimes engineers think they're selling themselves short or not showing their full skills by using SQL technology. I explain that there's no shame in using SQL as long as it does the job. Use the time it saved you from improving another part of the pipeline.

In this example, depending on the complexity of the query or code, it would save or lose a few hours. That's obviously not a make-or-break decision for the team, but you get significant time savings by compounding many choices like this one. In more extreme examples, using the wrong technology for the job means the team loses six months of work.

Big data technologies are constantly changing and improving. New technologies are being added to address niches in the market. Proper training also helps the data engineering team choose the right tool for the job.

There's a certain sense of bravado that I run into. I'll teach a team that tells me they're advanced and already know the technology. They don't need any help. A few outliers understand the technology and don't need any help, but the vast majority don't; they need some help and training. There is no shame in getting coaching or training on a brand-new technology. Nor is it an admission that you're not intelligent. An intelligent data engineering team knows that early is the best time to get help.

I see this all the time at companies that start coding before I come in to train them. The average amount wasted before training is $100,000 to $200,000. I go through the use case with the team and make sure the technology and designs are correct. I stopped teams from putting designs and systems in production where it would have cost $500,000 to $1 million in operational and development costs. The worst loss I have seen in such a situation is a team wasting $1 million to $1.5 million and six months of development time.

A big reason for wasted time and money are the subtle nuances of the big data technologies. It’s hard for a data engineer to absorb these nuances by picking up a book. Sometimes these subtleties revolve around use cases, are inherent in the technology, or spring from implicit assumptions in the technology. I can’t talk about every team’s use case because that isn’t practical, and I don’t want to bore you to death. Books like this one are general in nature.

In light of your use case, subtle technological nuances may make something either difficult or impossible. When I said data engineers need to know 10 to 30 different technologies, this is why. The data engineer needs to know these subtleties and then figure out the best tool for the job. It bears repeating: that tool may or may not be part of your company’s current stack.

CHAPTER 6

Running Successful Data Engineering Projects

Crawl, Walk, Run

If your data engineering team moves from small to big data, you'll need to train them well and give them intermediate projects to set them up for success. Going from zero to big data is a common way that big data projects fail; you'll need to get them going in a less jarring way. Attacking a big data project with an all-or-nothing mindset leads to an absolute failure. I highly suggest breaking the overall project into more manageable phases. These phases are called *crawl*, *walk*, and *run*.

If you go directly from crawling to running, you'll continually trip and fall and not know why. This direct route is a common issue with new data engineering teams. They don't set up a good foundation before moving on to the advanced phases.

I'm going to talk about what each phase is, offer some suggestions on what it should include, and discuss what your focus in each should be

Crawling

The crawling phase is doing the absolute minimum to start using big data. This phase might be as simple as getting your current data into a big data technology. For highly political organizations, this is the phase where you try to get groups to start working together. Ideally, you're getting other groups opposed to big data to buy into its value.

A good stretch goal for this phase is to automate data movement to and from your data pipeline. You'll want to bring in data continually.

You'll start on your ETL coding and data normalization in this phase. This phase will set you up for success as you begin analyzing the data. This phase has minimal amounts of analysis. Your focus is on creating a system that will make it as simple as possible to analyze the data.

Walking

The walking phase builds on the solid foundation you laid while crawling. Everything is ready for you to start gaining value from your data. If you didn't build a solid foundation, everything will come crashing down.

In this phase, you're starting to analyze your data. You're using the best analysis tool because the cluster is already running the right tools.

You're also creating data products. These aren't just entries in a database table; you focus on in-depth analysis that creates business value. At this point, you should be creating direct and quantifiable business value.

Last, you're starting to look at data science and machine learning as ways to improve your analysis or data products.

Running

You're moving into the advanced parts of big data architectures in the running phase. You're gaining the maximum amount of value from your data.

You're also figuring out where batch-oriented systems are holding you back and starting to look at real-time systems.

You're focusing on the finer points of your data pipeline. This phase includes optimizing data storage and retrieval. The process might involve choosing a better storage format or working with a NoSQL database. You're also using machine learning and data science to their fullest potential.

Iterating

The next question is how to maintain this velocity. Data pipelines aren't really ever done. They're constantly changing as you add new datasets and technologies and as other teams start using the data.

The team needs to iterate and repeat the crawl, walk, run again. The crawl may not be as straightforward or time-consuming; it could be more about validating that the task is possible. The walk and run phases would be mostly the same.

Technologies

This book is not about specific big data technologies. They will come and go. However, the general patterns will stay the same. The subtle nuances will change. The tradeoffs of one technology or another will change.

Technologies should be chosen only after looking at the use case and verifying which technologies will work. This choice is a common issue where companies zero in on a specific technology. They're used to small data solutions where everything can be accomplished with the same stack. That isn't the case with big data. A big data solution is completely geared around the use case.

The team's data engineers will need to keep up with the rapid technological change, especially in real-time systems. There are many different technologies to keep up with. Only by specializing in big data can the data engineers stay current with everything that's happening.

There is a tendency to blame technology for project or data pipeline failures. The technologies work just fine. It's far more challenging to look inwardly at a team and company to understand why they failed. There are usually several reasons and pieces to the answer, but technology isn't one of them—people and processes are at the top of the list of why teams fail. The team may not have followed the steps I outlined in this book and therefore failed.

In my experience, companies need a wide range of solutions, and they'll need them from several different companies. The focus on specific technologies often comes from the technology vendors themselves. They are pushing their own solution as the solution to everything.

KPIs Every Business Should Have For Its Data Teams

Data teams can be challenging. KPIs (Key Performance Indicators) for these terms are different from the criteria for other teams. The team's value creation and performance are distinct. This sidebar lists some KPIs that you'll find helpful when evaluating the success or failure of a data team.

Before measuring KPIs, create baseline numbers for all metrics. This preparation will allow you and your team to track maturity and growth. Without knowing where you started, you won't know how far you've come.

You may need to break your KPIs down to a project or product level as you look at your projects or data products.

Business-Focused

The following are suggested business-focused KPIs:

- *Percentage of self-supported queries*: The percentage of queries the business teams could do entirely by themselves, without asking the data team. Data teams should be focusing on offloading queries and providing infrastructure for the business to run queries themselves.
- *Business value created*: How much business value is created by the data products. The actual business value is highly business-dependent.
- *Increase in X business metric*: The increase in some metrics that can be directly tied to using data products. This could include metrics such as conversion or average sale price.
- *Increase in value to the customer*: The increase in some metrics that can be directly seen or experienced by the customer. This could include personalization or service/logistical speed.

Data Engineering-Focused

Some KPIs are more focused on the data engineering side:

- *Data product usage*: How much other teams or programs use your data product. Becoming a data-driven organization will increase this number.
- *Errors per unit/amount*: The number of errors in your data product. This shows how reliable and correct the data products are.
- *Perceived data quality*: How the users of the data products perceive the data quality. This is a subjective value.
- *Quantitative data quality*: Shows whether the contents of the data product are of high quality. For example, does the entry have the correct foreign key, or is it within defined constraints?
- *Number of useful data products*: The number of data products in use by the rest of the organization. Some organizations will have many data products, but an unknown percentage of those data products are being used.
- *Number of data model changes*: The number of times the data model changes over time. Too many data model changes may show a missing person or process for data models.

Operations-Focused

The operations team will need to create a set of KPIs around operational excellence:

- *Outages*: The number or percentage of time the data products are inaccessible. This is a total outage, not just a deterioration of service.
- *Framework and service uptime*: A framework or service loss might not represent a total outage. The operations team will want to track their framework and service uptimes.
- *Percentage of automation of tasks*: The percentage of tasks that are automated. Operations should increase the number of automated functions for reproducibility and consistency.
- *Incidents related to data quality*: The number of incidents related to a data quality issue. For example, how many outages could be traced back to a data quality issue?

Data Science-Focused

The data science team needs to create specific KPIs around AI and machine learning (ML):

- *Reduction of humans in the loop*: A reduction in the time humans used to spend that is now done by AI. For example, an inventory decision done entirely by a human is now augmented with AI to reduce the human's overall time spent.
- *Ease of getting data*: A subjective view of how difficult it is for a data scientist to access the data product exposed with the proper infrastructure. This can be quantified as how much time data scientists spend on data engineering.
- *Ease of deploying models*: A subjective measurement of how long or how much effort it takes to deploy a model into production. For example, this could be the mean time to train and deploy a model for the data science team.

Focusing Data Teams

KPIs help focus data teams on what they need to improve or show velocity. By choosing the right metrics and goals, management can show progress. Through focus, data teams can improve in all aspects of performance.

Why Do Teams Fail?

Part of being successful in a project is to learn from others' failures. I'm writing this book partly because I'm tired of seeing teams repeat the same pattern of failure. Most teams fail for the same or similar reasons, be it one reason or a combination of several.

When you're creating small data solutions, you have a much lower technical bar and level of complexity. If the data engineering team couldn't do something well, other means could cover it. With big data, anything you don't do well with small data just gets magnified. For example, if the team can't create or use complex systems, a big data solution will just magnify this deficiency. And as noted, data pipelines include 10 to 30 complex technologies.

A widespread recipe for failure is when the entire data engineering team comprises DBAs or Data Warehouse Engineers. The data engineering team needs to be multidisciplinary, so you must check for skills gaps.

Related to the skills gaps is when a team doesn't program or have Java experience. Big data technologies require programming, and Java is their predominant language. Not having any programming skills on the team is a big red flag. Not knowing Java (or Python or Scala) is an issue but not a showstopper. Programmers in one language can learn enough Java to get by. However, you need to give the team the resources and time to succeed with a brand-new language. You'll also need to verify that any project timelines consider the time spent by the team learning a new language.

Other teams lack a clear use case. They've been told to find one or thrown into big data as seen as a silver bullet that will solve any problem. These projects are doomed because there is no finish line. Without a clear use case, technologies cannot be chosen, solutions written, or value created.

Other teams face an albatross in the form of extensive legacy systems. The new data pipeline is expected to be backward compatible or support all previous systems. This legacy stymies execution, leading to failure when projects' plans and managers don't account for the increase in time. Heavy legacy systems integrated with data pipelines often take 50% longer to complete.

Perhaps my biggest pet peeve when working with data engineering teams is when they're set up for failure. There's little to no preparation for the project. The engineering team doesn't have a use case and hasn't been given the resources to figure it out. I hate seeing doomed projects.

Other teams lack qualified data engineers or don't have a project veteran. These teams may be initially successful, but over the long term, they will fail. They'll get stuck at the crawl phase. They won't have the skills or abilities to progress beyond the basics. A project veteran can help lead the team through the difficult transitions of the walk and run phases.

Some teams lack members who understand the entire system and the importance of schemas. These failures don't happen initially; they manifest in the walk and run phases when other parts of the organization start using the data pipeline. A change to a file will have a ripple effect on the rest of the data pipeline. The team that didn't engineer their schema correctly will have a maintenance nightmare on their hands.

Some organizations think big data is simple and cheap. These organizations fail to pay data engineers sufficiently and don't get qualified ones. The organization fails to give the team the resources they need to succeed and then holds them accountable for failing. They think a big data project can be done in the same timeframe as a small data project. They're wrong, and they'll repeatedly fail until they make changes.

Still, other organizations are incredibly ambitious. They'll show me their use case and data pipeline. It will be the holy grail of data pipelines. I'll ask them questions about their data engineering team and their experience and discover that the team will be absolute beginners. This mismatch of ambition and actual skill causes failures.

Why do companies fail?

Sometimes, it's the entire company that causes the failure. The data team could be productive and provide a data pipeline that doesn't get used. The company needs a data-driven culture and a data-centric attitude for a successful project. Without this attitude, the data engineering team's insights and data won't be transformed into business value. Other times, you need to make changes to the different parts of the organization to leverage your data fully.

Why Do Teams Succeed?

In my experience, a team's success starts and ends with the people on it. If you have qualified people who work hard, you will have a good chance at success. If you have the wrong people, the project won't go anywhere.

Training is the biggest key to success. A team that has been trained on a technology has a high chance of success. They've been able to ask the right questions and compress the time needed to learn the technology. I compare this successful training to questions I get and see from untrained people. The question itself often shows a general misunderstanding of the technology. I could give these students the correct answer, but they won't be able to apply it due to their lack of understanding. Yes, some people can learn on their own. But they're few and far between, and they're outliers.

It’s never too late to turn the ship around

I taught at a large media and digital advertising company. As I was preparing to go there, the salesperson told me that the team was six months into the project, and their CxO and VPs were closely watching this project. I thought that was very odd. Usually, a team will train early on in the project or not.

I arrived to start the first day of class. I talked to the class about the project and what was going on. The team was six months into the project, and it was woefully behind schedule. The team had their backs to the wall.

A few things were clear about the team. They were intelligent, dedicated, and working hard. It was also clear they had fundamental misunderstandings of the technology and how it should be used. I explained to the class that I would be teaching the class differently from my usual process. I would be going deeper into the things they didn’t understand.

I also told them to schedule a meeting for the following Monday. I told them the meeting should take four to six hours because they would have to assess how long it would take to rewrite their software once they correctly understood the technology.

Throughout the class, I would say, "Take note of that. That’s something big you’re going to have to change. Make sure to talk about it on Monday."

I kept up with the team after that class. I wanted to know whether the project turned around, and it did. After a month, the team took off in the right direction and shipped the product.

Before I arrived, the team had wasted six months and 1*millionto*1.5 million. Worse yet, the team’s CxO and VPs weren’t happy. After the team turned itself around, they became the toast of the company. Many of the members were promoted based on their ability to create successful big data projects. They could have saved themselves time and money by getting trained.

Another characteristic of success is preparing the team and the expectations for the tasks it needs to do. The organization hasn't just thrown the team into a no-win situation. They've given the team the resources and time to complete the tasks.

A final characteristic is that the organization has rational expectations for the team. The team has an achievable and known finish line. There will be many different races over a project's life, but each should have a well-understood, reasonable finish line. There is nothing more depressing and morale-shattering than going into a project you know the team can't finish. An appropriate, achievable goal gives the team confidence and greater velocity in completing the project or task.

Paying the Piper

Some teams' productivity is hampered by postponing the adoption of big data techniques and processes. This procrastination leads to massive technical debt. In these situations, the company has been experiencing big data problems for some time but refused to believe that its old ways of handling data were insufficient. Instead of solving the problem by taking the leap to big data, they've been patching and propping up small data solutions.

I've seen and experienced these delays before. I've worked at companies where they're investing in propping up a small data solution by shoring up their database. They had a DBA dedicated to looking for queries to optimize and any other scaling issues. That came at the expense of putting money toward a real solution to a big data problem.

I've taught at other companies doing a similar thing. They're so busy shoring up their small data solution that they can't look far enough into the future to fix the problem with a big data solution.

I tell these companies they're a train speeding toward a reinforced concrete wall. It may be a year, two, or five years before they hit that wall, but they will. If you can do a back-of-envelope calculation of how fast your data grows, you can calculate when you'll hit that wall.

Let me give you a typical and concrete example. Financial organizations often need to run end-of-day reports. These are time-boxed and have to be done at a specific time, or there's a monetary penalty. You have five hours to run the job, and it's taking four hours to complete. The team is putting their finger in the dike by vertically scaling the processing and trying to optimize the database. The expected growth of data and the increase in complexity of the report can lead you to see that you'll hit the wall in four years.

You breathe a sigh of relief because four years is a long time away. Except:

- The organization could grow much faster than you expected.
- The organization could purchase another organization and reach the maximum capacity of the current system sooner.
- Given the legacy systems and complexity, it could take two to four years to finish a solution.

If you have big data problems that you're attempting to solve with small data solutions, you'll eventually have to pay the piper. Doing so too late could mean crashing into a reinforced concrete wall of catastrophic failure.

Some Technologies Are Just Dead Ends

Some teams thought they would never have to rewrite their pipeline. They could be early adopters of a technology and chose what they believed were the winners. Some of these big data technologies range from graduate school thesis code to code written in an enterprise donated to an open source project. These technologies' varying degrees of completeness and enterprise readiness contribute to their demise.

This quality may surprise you, but not all graduate school thesis code is well thought out or production-ready. Unless you or your team has the ability or resources to evaluate technology at this level, you will have to rewrite.

Sometimes the vendors have the right idea and the wrong implementation. Let's face it; the vendors aren't immune from advancing their own agendas and policies. This situation is when the worst abominations are born. Other times, the vendors have good ideas but don't invest enough in the technology.

If you imagine a spectrum with conservative bets on one end and aggressive bets on the other, that's how you can think about big data technologies. The bolder, the higher the odds of wholesale rewrites and technology changes.

Some technologies pride themselves on how much they change. If they didn't get it right two years ago, what makes you think they will get it right next time?

But sometimes, you'll need cutting-edge technology. To that end, you'll want to take steps to mitigate your risks. One simple step is to use conservative, already proven technologies.

Some technologies put a layer between your code and their APIs. Apache Beam is one such technology. Because your team doesn't write directly to the technology's API, the team can move code around without having to rewrite it.

What if You Have Skills Gaps and Still Have to Transition to Big Data?

Sometimes I'll teach at a company where the team is wholly unprepared for the tasks ahead of them. In some companies, the gap is small and needs to be traversed. The gap is a canyon for other companies, and crossing it will require a significant amount of help.

The first step is to identify which skills were missing and how critical those gaps are to the team. This step requires a great deal of honesty.

Let's go through a few common examples I've seen with data engineering teams.

No Programming Skills

The most common scenario I've seen is data engineering teams with no programming skills. The team is replete with DBAs and SQL skills but doesn't know how to program complex systems. This team makeup often results from taking data warehousing teams and calling them data engineering teams.

At a large organization, I'll look around for data-focused programming teams. Failing that, I search for programming teams with multithreading skills. If I still can't find that team, I look for any Java or Python team with a preference for Java.

Each of these teams will need training; it's just a matter of how much they'll need. To the degree that the data's complexity exceeds the people's experience assigned to the team, the odds increase that the team will not complete their project. Not all people with programming skills can handle the complexity of big data.

If you don't have any programming skills in the company, you'll need to hire one or more qualified data engineers as a consultant. You can also hire a company to write the code for you. I urge caution, as some consulting companies will say they do big data solutions but are just as lost as their clients. Look for consulting companies with a proven track record in big data projects.

Actual Programming Skills

I mentored a large financial company that didn't have the programming skills to create the data pipeline. They decided to hire a consulting company before I started helping the team.

During the kickoff meeting, I started asking rudimentary questions about the technology choices and the use cases because the consulting company had already started the project. They couldn't answer any of my foundational questions. Confused, I asked the team whether someone else was tasked with the code and if I was talking to the wrong person—no, I was talking to the right person.

It turned out that the team hired a consulting company that did only data warehousing. They didn't have the programming skills, or ability, to create the solution. The consultants were hoping I'd train them and show them how to create the solution. I had to tell the company they made a mistake in hiring the consulting company and needed to fire them.

No Project Veteran

Hiring a project veteran can be tricky. One may not be available, or you may not be able to afford one. You'll have to contend with your less experienced team.

When no project veteran is present, I keep a sharp eye out for abuses of the technology. I also keep an eye out for the team missing the subtle nuances of the use case and technology. These teams tend to create solutions held together with more duct tape and hope than you'd like to see.

Absent a veteran, you can get the team some mentoring to help teach them over time. You can get training; we go through at least one use case in my training classes. You can also get specific consulting. This consulting could range from a second pair of eyes to go through your design to having a long-term relationship with a consultant.

Lack of Domain Knowledge

Another scenario I've seen is where a data engineering team is brand new and hired from outside of the company. A lack of understanding of the domain can lead to misunderstanding the use case. That can spell disaster for the project.

This scenario often happens when a data engineering team replaces a legacy system. The team that created, supports, and makes their living off the legacy system feels threatened. I've trained at many companies where this is the case.

I've helped them by training both teams at the same time. I'll have members of the legacy and data engineering teams attend to get the same knowledge. My hidden agenda is that I'm breaking down the political barriers between the teams. Sometimes companies don't communicate effectively internally. The training session is a great time to handle this breakdown.

I suggest you handle this lack of skill politically rather than through business or technical means. You score allies from the legacy team to help you by breaking down the political walls. You'll learn how and why they engineered the solution the way they did. You'll learn some of the pitfalls they hit while creating.

You'll gain the legacy team's domain knowledge for your team by solving this problem.

CHAPTER 7

Steps to Creating Big Data Solutions

I hope you didn't just skip to this chapter thinking you didn't need to read the rest of the book. Without reading the preceding material, you won't understand the whys and hows of these steps. That's an excellent way to make your big data solution fail.

These steps are at the 30,000-foot level, not the granular day-to-day actions. Those steps are specific to your development methodology, such as Waterfall or Agile.

Agile Data Engineering Teams

The most productive data engineering teams use Agile methodologies. If you are still using Waterfall and are starting with big data, you'll need to switch. You need to respond to new issues with more speed than Waterfall allows. Big data is too complex to plan everything ahead of time

Pre-project Steps

Before starting your big data project, you'll need to take care of some tasks when creating the team.

Create a Data Engineering Team

Often companies don't understand the need to create an actual data engineering team. If you don't already have a team in place, you will need to create one. I've already outlined why this should be done earlier in the book in Chapter 2, "The Need for Data Engineering," and Chapter 3, "Data Engineering Teams."

Check for Gaps

If you already have a team or are in the process of creating it, you will need to check for gaps.

The first gaps to check for are skills gaps. This check requires enormous honesty about the team's skills. Use the skills gap analysis mentioned earlier in the book.

The next gap to check for is an ability gap. You may have people on the team who will never be able to perform. The significant uptick in complexity puts big data out of their reach, and no amount of time and help will change that. It would be unfair to expect people with an ability gap to perform on a team.

You didn't skip one of the most crucial parts?

If you didn't perform the skills gap and ability gap checks in Chapter 3, "Data Engineering Teams," you need to. This exercise isn't optional.

Use Case

A significant difference between big data and small data projects is how deeply you need to know and understand your use case. Skipping this step leads to failed projects. Every decision from this point on should be viewed through the lens of the use case. Furthermore, there may be many different use cases, and you will need to understand each one.

Questions you should ask about your use case include the following:

- What are you trying to accomplish?
- How fast do you need the results of each calculation? Do they need to be in real-time, batch, or both?
- What is the business value of this project?
- What difference will it make in the company?
- In what time frame does the project need to be completed?
- How much technical debt do you have?
- How many different use cases are there?
- How much data will you need to store initially? How much data do you plan to grow on a daily basis?

- Will you need to store a wide variety of data types?
- How big is your user base? If you don't have users, how many sources of data or devices will you have?
- How computationally complex are the algorithms you're going to run?
- How can you break up your system into crawl, walk, run?
- Is the use case changing often?
- How is data being accessed for the use case?
- How secure and encrypted does the data need to be?
- How will you handle the data's management and governance?
- Who will own the data, and are they encouraging other teams to use it?

Teaching and Use Cases

When I teach, I have to make an effort to learn the organization's use case. This learning is because the answer to almost all of their questions depends on it. Whenever they ask me a question, the answer isn't a one-size-fits-all response. It comes down to what they're doing and how they're doing it. Only by knowing their use case can I remotely answer their question.

Is It Big Data?

Now that you understand your use case, you'll need to determine whether it requires big data technologies. This situation is where a qualified data engineer is crucial. Given your knowledge of the use case, the data engineer will help you decide.

When compiling the use case information, a common word to look for is "can't." When talking to the other team members about the use case, they'll often talk about how they're trying to do this now or have tried it before. These scenarios "can't" because of some technical limitations related to using small data technologies for big data problems. By changing the technologies to big data ones, the answer will change to "can."

Will It Become Big Data?

Sometimes an organization will not have or be experiencing big data problems yet, but they're projecting big data problems shortly. These are the organizations that I characterized as a train hitting a reinforced concrete wall. Or they're startups that are expecting to grow exponentially. Either way, they are an organization that doesn't have big data now but will someday.

They're faced with the difficult decision of when to start using big data technologies. I've seen what happens in both cases.

For startups, I've seen them use small data technologies and hit it big. Then the startup can't move over to a big data technology fast enough, so it starts losing customers and traffic. I've also seen startups deliberately choose not to implement big data technologies due to the complexity and increased project times. Those companies went out of business before they needed big data technologies.

For larger organizations, I've seen them delay moving to big data technologies due to the difficulty of leaving behind their legacy systems. They create so much technical debt for themselves that they spend years moving to a new system. They just have to hope that they don't hit a reinforced concrete wall during that time.

Evaluate the Team

Now that you have decided that your use case must be solved with one or more big data technologies, which are correct and why? This portion requires an honest and realistic look at the team's makeup and abilities. To determine whether your team can execute a big data project, ask yourself:

- Does my team have a background in distributed systems?
- Does my team have a background in multithreading?
- What programming language(s) does my team use?
- What small data technologies is my team already familiar with?
- Does my team suffer from an unrealistic view of their abilities?
- Do we all realize the increase in complexity associated with big data?

Training, Mentoring, Consulting

After a very honest look at the team, you will need to see how you can set the team up for success. Failing to set the team up for success is a common way to make projects fail.

An average data engineering team needs training and mentoring. An advanced team may just need training on new technologies. In the "How I Evaluate Teams" section of Chapter 5, I talk about how to figure out what level of help a team needs.

A team with a very low probability of success will need in-depth training and consulting. After honestly assessing the team's abilities, you may need to hire a consulting company. This company could either write the portions of the data pipeline that are too difficult for the team or create the whole project.

Choose the Technologies

After your honest and realistic look at the team, you need to decide on technologies. Make this a starting point for discussion with the team. This discussion is another step where it's crucial to have qualified data engineers. Think about these questions:

- What are the technologies that will make the project successful?
- What are the technologies that my team can be successful with?
- Does my team or company suffer from NIH (not invented here)? The team may want to roll their own or create their own distributed system. (Usually not a good idea.)
- Does the team have the training necessary to be successful?

Write the Code

This is the fun step for data engineering teams—it's where they code out the use case. It's where they get into the specifics of the big data technologies and implement things. They start to wire the various pieces and technologies together.

Writing the code is another common place where teams fail. Instead of doing the preceding steps, they jump in and start directly writing the code. They've already chosen the technology and haven't looked over their team or use case. They get six months into the project and realize they've made a mistake.

Yes, this is the make-or-break step. You find out the hard way if you followed everything I've outlined.

Evaluate

This step is where you look at the data pipeline you've written. Then you compare it against the goals for that phase of the project. You don't evaluate the project just at a technical level. You need to assess it in relation to the use case and business.

It's imperative to evaluate the project, yet it's often not done. This evaluation is an important opportunity for the team to learn from previous issues. In Agile methodology, this would be similar to a retrospective, except that it would be for the entire POC.

Ask yourself the following questions:

- Did the iteration solve the business need?
- Did the iteration take the amount of time you thought it would?
- What is the quality of the code that makes up the iteration?
- Would you put this level of code quality in production?
- Was the iteration so simple that it gave the CxOs or upper management a false sense of timelines and complexity?

Repeat

In this step, you figure out how to break the project down into small pieces. This step enables an iterative approach to create a data pipeline instead of an all-or-nothing one.

Projects that have an all-or-nothing approach often fail due to timelines and complexity. It is crucial to break up the overall design or system into phases—crawl, walk, run. I encourage you to segment your development phases to create the system gradually rather than all at once. In approaching this, ask these questions:

- How can you break up your system into crawl, walk, run?
- Has the rest of the management been informed which parts or features of the system will be in which phase?
- Is the next phase in the project vastly more complex or time-consuming than the previous phase?
- Do the phases have a logical progression?
- Is one feature dependent on a feature that currently appears later in the schedule?

Probability of Success

When I work with teams, I create a probability of success. This probability takes into account:

- The team's abilities
- The gaps in the team's skills
- The complexity of the use case
- The complexity of the technologies you need to use
- The complexity of the resulting code
- The company's track record in doing complex software
- How well the team is set up for success
- How much external help the team is going to receive (training, mentoring, consulting)

In the various real environments I have evaluated, probabilities of success have ranged from 1% to 99%. Calculating your probability of success with reasonable accuracy requires a frank look at things. As you looked over the questions and topics in this book:

- Did you have deep reservations about the team succeeding?
- Did they remind you of your team and their skills as they currently stand?

Now, I want you to think of ways to increase that probability:

- Would more qualified data engineers improve it?
- Would getting more or better external help increase it?
- Do you need to vastly decrease the scope or complexity of the use case?
- Is there a gap in skill or ability, that's making success improbable?

Suppose you have a high probability, congratulations. You'll likely be successful. However, many teams fool themselves with a dishonest assessment of their probability of success. They significantly overestimate their abilities.

If you have a low probability, you should think hard about undertaking the project. This situation is a time to get extensive external help. At a minimum, your team will need training and mentorship. If the probability is very low, you should honestly consider having a competent consulting company handle the entire project. You'll be setting the team up for failure without doing that, which isn't fair to you or the team.

CHAPTER 8

Conclusion

Best Efforts

Big data systems are incredibly complex. Part of your job will be to educate your coworkers about this fact. Failing to do so will make everyone compare your progress to easier projects, like a mobile or web project.

Despite your best efforts and planning, big data solutions are still problematic. This eventuality isn't admitting failure or an issue with the team; big data is really that hard. When you start falling behind in the project or hitting a roadblock, I highly suggest getting help quickly. I've helped companies that had been stuck for several months. Had they reached out for help sooner, they could have saved months of time and money.

Best of luck on your big data journey.

About the Author

Jesse Anderson is a Data Engineer, Creative Engineer, and Managing Director of Big Data Institute.

He mentors companies all over the world ranging from startups to Fortune 100 companies on Big Data. This includes projects using cutting-edge technologies like Apache Kafka, Apache Hadoop, and Apache Spark.

He is widely regarded as an expert in the field and for his novel teaching practices. Jesse is published on Apress, O'Reilly, and Pragmatic Programmers. He has been covered in prestigious publications such as The Wall Street Journal, Harvard Business Review, CNN, BBC, NPR, Engadget, and Wired.

CHAPTER A

Appendix A: Criteo's Data Engineering Journey

An interview with Justin Coffey and François Jehl

Introduction

This interview is the continuation of the case studies and interviews I did for Data Teams. I'm sharing the ideas of the people who are out there dealing with the messy real world on a daily basis.

Some of the opinions and experiences will go directly against or will appear to go against the recommendations I've made in this book. I decided to keep these passages to show there are several different starting points and viewpoints on data teams. There are a plethora of reasons why there is a difference. I made the decision not to editorialize on what the interviewees said.

All of these interviews represent the views of the individuals and not necessarily those of the companies where they worked.

About This Interview

People: Justin Coffey and François Jehl Time period: 2012-2020 Project management frameworks: Scrum Companies covered: Criterio

Background

Justin Coffey was an Engineering Director in Data Processing at Criteo. He is currently the Head of Data Privacy Infrastructure for Siri AI/ML at Apple.

François Jehl is the Senior Engineering Manager in Data Platform at Criteo. He has a master's degree in Databases and Artificial Intelligence.

*About Criteo: "Founded by a small group of great minds at a start-up incubator ir

Organizational Structure

In 2012, Criteo started feeling the issues of scale. Their relational databases and DBAs were having trouble keeping up with the growing scale of data and increased demand for data products. The company had hit the limit of what they could do with relational databases and needed to find new solutions.

To deal with their scaling issues, Criteo hired Justin to manage and create a new team to process data. In the beginning, Criteo's management team envisioned it as an analytics or business intelligence team that was supporting the sales organization. The team started with three analysts and a detachment of engineers from research and development dedicated to the analysts.

Later on, the Research and Development (R&D) group subsumed Justin's team. The move created an analytics group within R&D. Before he left, Justin gave the analysts the opportunity to stay in the analytics group. One of the analysts decided to move over to R&D.

At that point, the R&D team was using their engineers to do machine learning. It was simple machine learning based largely on tree logic. Criteo realized that they could do even more with machine learning. They hired a VP of Research in AI and started scaling up that organization. They began hiring data scientists to apply advanced analytics like machine learning.

At this point, there were two parallel data science organizations. Justin's team was doing the data engineering and creating data products for both organizations. It also fell to the data engineers to onboard the data science and analytics teams onto their infrastructure.

In its current form, Criteo's data engineering team has five sub-teams with a total of about two dozen engineers. Justin has left Criteo, and François has replaced him as the data engineering team's overall manager.

Prologue to Change

Criteo was a bit slow to modernize their data processing because they tried too long to preserve their old SQL-based queries and relational databases beyond a scale that these tools could sustain.

These limitations came to a head with a single business-critical use case. Because Criteo served banner ads, they needed to count the number of users who saw each ad. The ad views were a key metric and hadn't been surfaced to their customers (the buyers of ads).

The teams were satisfying the metrics through a relational database's distinct count feature, which is resource-intensive. The DBA teams used as many hacks and workarounds as possible to make these distinct counts continue to work as data sets swelled. Their original solution was to scale vertically and create bigger and bigger data marts. But Criteo realized that this solution wasn't going to scale any further because their data volumes were doubling every year. They needed to onboard more customers every year.

The Difficulties of Distinct Counts

Doing distinct counts on a large amount of data is a difficult problem. They're also commonly solved through big data tools because of small data systems' inability to handle them without issues. Distinct counts are difficult because you have to store every item you're counting and their counts. At a large scale, the number of things you're counting can be a huge number, called *high cardinality*. When doing counts, every row or piece of data has to be read and counted. An added complexity is that these counts have to be absolutely accurate because their billing is tied to the precise results.

A Big Data Approach

The turning point for adopting a new solution started with a sit-down meeting with the CTO, where the management and technical staff asked themselves what they were going to do. They decided that the traditional mechanisms weren't working anymore. They read a blog posting about Hadoop and decided to switch to more modern tools to solve their big data problems. They increased their cluster size from 60 to 160 nodes (today, they have more than 6,000 nodes).

It was Justin's job to take Hadoop from being a curiosity at the company to creating business value. His team started by tackling the count problem. There were two engineers focused on Hive. Two more wrote MapReduce code and eventually started using Cascading.

One of Justin's first actions was to remove the legacy counting tools that the teams had built. The tools used replicas of the production database as their data source. Under the old system, DBAs had sharded the replicas to deal with the scale, which prevented getting a holistic view of all data. There were associated ETL tools and corresponding technical debt. They started removing the databases one by one.

The need to count unique impressions of ads was easier to solve with Hadoop. They created an hourly transfer of all logged data of ad impressions. At the end of the day, the batch process would tell them how many unique impressions they had. With a simple solution in place, they then started to optimize their processing and data layout.

They changed the scheduling to be more data-driven instead of time scheduled. The scheduling was facilitated by a homegrown orchestration program Justin's team created. At that time, there weren't many good orchestration technologies for Hadoop.

Initially, the engineers writing Hive and MapReduce were under different managers. Criteo realized that they need to tackle the problem more holistically. Thus, they merged all these engineers into a single team under Justin.

Criteo was lucky and "stumbled into best practices" because the software they created was robust. This virtuous cycle created "early success convinced the business to invest more in this area because we were solving problems."

This success led the team to the next hurdle and another meeting to deal with more production use cases. Criteo created a new architecture that needed massive data processing to create a new data product. During the meeting, everyone looked to Justin's team to create a new data product. He tried to say they did back office and non-production data products. "We realized that MapReduce and batch processing could actually be Hadoop-based and part of a critical path."

From then on, Justin's team went from just doing back-office reports without stringent SLAs to a responsibility for production reports with strict SLAs.

Roadblocks With DBAs

Data engineering is often at odds with DBA or data warehousing teams. At Criteo, the DBAs were using SQL Server for their work. Because SQL Server wasn't able to keep up with the load, they had to do various workarounds. Once the data engineering team took over the big data workloads, the DBAs were initially roadblocks. Justin said, “they’re a roadblock, and we’ll work around that roadblock.”

The DBAs were trying to manage data as they did for traditional databases and usage. The DBAs were worried that the limitations of their technologies would carry over into the big data technologies. Their concerns included strict isolation requirements, resource contention, and people running whatever analytics they wanted. In contrast, the data engineering team allowed people full access to the data to discover new things. The big data technologies were free of the limitations the DBAs were used to.

While the DBAs were initial roadblocks, Criteo knew that wasn’t a healthy relationship. Justin remembered, “it’s true we had tough run-ins with them early on, but all parties worked hard to repair ties and by the time I left the DBAs were one of our closest partners on infrastructure topics.” Through concerted efforts and growth, the two teams were able to make repairs and create a healthy relationship.

Creating New Systems

Making Hadoop part of the critical path as a production system led to the creation of new systems.

As data and usage grew, other systems started to break under the increased load. The internal billing system was rearchitected because the SQL Server architecture couldn’t handle it. Justin advocated for Hadoop to replace it.

There were two technical problems with replacing the billing system’s SQL Server with Hadoop. First, the code for doing the actual calculations had to be written. Second, they needed to improve the monitoring of their Hadoop cluster and all of the jobs.

The decision came down to another meeting where the CTO asked Justin, “Am I going to live to regret this day?” And Justin answered, “No, it’s going to be just fine. Trust me.” After “nine painful months,” it was just fine. After those nine months, they had reduced their latency to under three hours.

But to reach that goal, they had to learn a lot of lessons the hard way. They rewrote their Ruby job scheduler because of various constraints. A big one was that they lacked the notion of catching up due to an outage or slow performance.

The team rewrote the system in Scala. Since the Scala scheduler was brand new, it had its bugs to fix, and the team fought through them.

From this nine-month crucible, “we really proved to ourselves that we could do quite a lot,” Justin said. The experience gave them the velocity to believe they were a first-class team and “can tackle these distributed problems.”

Going from time-based scheduling to data-driven scheduling came out of operational necessity. With time-based scheduling, everything had to arrive by the time the job kicked off. But because of the erratic arrival of data, they tried to set later and later times to start processing the data, which didn’t always work. There could have been problems at the data center when the files were being copied. It was operationally painful because there was a “partial computation, and you get all your clients screaming at you because you have half the data for some data center.” The entire job had to be rerun.

Executive Sponsorship

Justin believes that the team experienced many early successes because they “were laser-focused on solving problem A, then problem B, then problem C.” They sometimes experienced failure when they “underestimated the complexity in front of us” or by “biting off too much to chew.” Attempts to solve too many jobs at once sent the team into “too many parallel directions.” They experienced the most success when saying they’re going to solve “this particular workflow problem for you. And then we’ll see where that takes us.” They experienced the opposite “when we changed to saying we’re going to solve all your problems. That’s when we had a lot of stumbles.”

Criteo hired a new VP of Engineering. He took a look at the system Justin’s team created, “and he said it was quite exceptional.” They gained a sponsor in him. “That provided a pretty large umbrella for us to have legitimacy throughout the organization.” The VP’s support “carved out some space for us,” Justin remembered.

Better Tools Now

Because Criteo started its journey early on, Hadoop was the only game in town. The focus on Hadoop can lead people to wonder if all of their problems would be solved or significantly mitigated by the new big data technologies that are available. Justin believes there is an 80/20 rule to new technologies. They should allow you to accomplish 80 percent of what you're trying to achieve efficiently. The remaining 20 percent will continue to be problematic.

"We have some very large Spark jobs and some very large MapReduce jobs. Which ones do you think fail?" Justin asked. The Spark jobs would fail for various reasons, such as out of memory, network issues, or how long a job needed to run.

There are newer and better schedulers now. When they were adopting big data solutions, the team determined that they would have to write their own scheduler, but they wouldn't need to do that given today's tools.

There are also better SQL interfaces for big data now. Criteo started out using Hive, but its stability was problematic. There are now large-scale SQL execution engines such as Presto, Vertica, or Redshift.

Just choosing new technologies isn't the only consideration. The team will need new expertise, and the existing acumen can be low. Difficulties have increased for development and architecture, due to the increasing numbers of technologies and problems they're expected to solve. Justin recommends reading and understanding the technologies first and getting some external training.

At the same time, operational difficulties have decreased because of the cloud. Starting a Hadoop or Spark cluster is trivial because the cloud providers start one with the push of a few buttons. When Criteo first started, these weren't options, and all operations were more difficult. Containerization also helped reduce operational complexity. Containers can be used for scheduling and for making it easier to deploy and monitor frameworks or software.

The plethora of options brings in some difficulty too. Data engineers will need to know 20+ different options to choose the right ones. In the Hadoop timeframe, there were far limited choices. There are now even choices of programming languages. Today, you can select from Java, Scala, Python, and others.

Governance and Engagement Models

To make this big transition, it was essential to create the right governance and engagements with the rest of the organization. The data engineering team needed to work with the analytics, R&D, data science, and business teams.

For the more technical employees, the collaboration included supporting their languages of choice. For the data scientists, it was Python. For the R&D teams, it was a variety of languages, including C#.

Some teams required mentoring. It was up to the data engineers to help their internal customers understand the right tool for the job. They had to educate their users on why a particular data science algorithm wouldn't run well on a database but could be run on another technology scalably and reliably. Often, the vendors were pushing their products for every possible use case. Justin recommends never listening to vendors, but to trace what motivates the founders of the technology to build it. If you figure this out, you will find the use cases that the technology best supports.

The Anarchists of Data Versus Pirate Infrastructure

Early on, the team faced difficulties from internal teams and a significant backlog.

The analytics teams weren't happy about the data engineering team's backlog. They felt as though the data engineering team wasn't getting anything done. The analytics team had a job to do, and the data engineers were standing in their way with rules. The analytics team started building their own infrastructure that the data engineering team called "pirate infrastructure." The data engineering team tried to convince them why this is a poor idea in terms of robust engineering and why working together was a better idea.

The data engineering team had to work with the analytics team to make it easier to access the data. They had to balance the right governance and controls with something that the analytics team could use.

When the data engineering team first started giving analysts and end-users access, the DBA teams wanted nothing to do with it even though the infrastructure was supposed to be for end-users. The data engineering team was seen as the "anarchists of data." They weren't putting in heavy controls and keeping people away from the data. François said of DBAs, "their old job is about providing a given subset of queries on the master." It wasn't to provide access to all of the data, as the analysts needed.

François built his team as being different from the DBAs. Eventually, they had to adopt some of the controls the DBAs had. Initially, the data and infrastructure were wide open and without specific SLAs. They discovered the hard way that users and consumers of data products will eventually require SLAs.

To create SLAs, François first looked at how the DBAs were doing. They employed extreme amounts of time-consuming processes. He didn't want that sort of solution because it would ruin the openness of the platform. Instead, he started requiring that users get a code review from a co-worker before deploying their query. He also mandated that people use source control to check in queries. The new process solved bad queries that had led to terrible SLAs while preserving the platform's openness.

Organizational Stress Points

Criteo repeatedly found step functions or considerable increases in organizational complexity in the team. Justin said one such strain occurred when the team reached about 15 people. The team had written so much code and stood up so much infrastructure that things had gotten out of hand. They realized that there were under-observed code and infrastructure. Justin said they "rethought the architecture of our analytic stack and bolted on all the sort of best practices that you would expect in terms of deployment monitoring."

Justin said they focused on "checking off all the boxes [of] good infrastructure" because they hit a threshold. "We had been pushing the envelope in terms of what a group of, say, fifteen people could do and you could promise and then actually accomplish in a reliable way."

Improving the reliability of the system pushed the team to double in size over the following year. The staffing increase led to another threshold for the team. The team could be bifurcated into the people who had been there from the beginning and the new people. The people who had been there the longest understood the decisions and the "mythology of the team." Justin said, "There's sort of this disillusionment with the direction of the team from some of the new folks because we didn't really transmit the lessons that we learned in the right way. We didn't document. And so a bunch of people thought, oh, we can add all these features, and they're missing like, no, if we do that, we're gonna break it again. And so there was this time where I went from like 15 to 30 people, and there were a sufficient number of disillusioned people in the 15 additional people we hired that there is a big cultural problem in the team. And so that was another life lesson learned and a threshold in effect in terms of the delivery of the team."

As the team doubled in size again, Criteo tried to keep from committing the same mistake. "That's when we started looking at things more holistically and started thinking about mission statements and who stakeholders were," Justin said. They did a "reorg and got people focused where they felt they could bring value to our stakeholders and be motivated."

By focusing better, Criteo is "among the best IT infrastructure on the planet, in the top 10," and "we went from serving about 20 to 30 people to serving about a thousand people."

Passing the Baton To François

About a year before Justin left, the data engineering team was divided into three sub-teams. Justin was the overall leader of all batch data and analytics. The three sub-teams were tooling, infrastructure, and pipelines.

François started leading the infrastructure sub-team. Justin kept most of the management for tooling under him. François was promoted to a peer of Justin on the organization chart.

The pipelines sub-team was more based on the legacy of how Criteo had created data products. Both Justin and François were convinced that they should provide the services for the individual teams to develop their own pipelines. The data engineers were there to provide the right technologies and capabilities to make sure jobs ran smoothly. Eventually, the pipelines team moved to a different manager.

At that time, the machine learning team's infrastructure was managed by the data scientists. François inherited the responsibility for that infrastructure. His team became "infrastructure plus MLOps."

Despite the change in management structure, François said, "I was still running my ideas and my group by him (Justin) and taking inspiration from his ideas. I cannot think of a single decision that I didn't run through Justin's judgment."

Justin and François often agreed on technical and organizational directions, but occasionally there were disagreements. "We agree to disagree on some points," François said. They were always able to make a compromise and a go-forward plan.

Because of their healthy relationship, Justin recommended François be the one to take over when he left the company. At first, François was torn because they had made so many decisions together. François would miss that sounding board for making decisions and would have to make these decisions on his own.

The first week on his own, François started digging into the roadmaps and strategy. He already knew them like “your slippers, you know them so well. It’s so comfy. You know that perfectly. The assumptions are things you’ve been discussing for quite a while.” Once François realized that he understood things well, he realized it wouldn’t be as difficult as he thought it would be to take over Justin’s role.

Stupidly Simple

Justin cited John Carmac’s philosophy of “always take an aggressively simple approach to things.” He says, “I actually take the same approach in designing systems because every feature you remove is one less failure mode and one less thing to monitor.” And “any given feature probably has about three data points to monitor. And that means additional complexity in your runtime. It just gets Byzantine at some point.”

To keep things aggressively simple, “we looked at the primary use case and said, ‘how can we implement that with the least bells and whistles possible in a way that’s the most repeatable and most immediately understandable to the next person who’s going to come on?’ ” That led them to make extensive use of HDFS. They identified anti-patterns and didn’t let users run “crazy jobs.” They focused on standardizing how data landed and was queried so it could be usable by others. They removed one-offs to make sure the primary uses cases were handled effectively.

Part of being stupidly simple included deeply understanding the technologies they were using. They chose a technology and read the papers that the creator wrote about it. “We’re going to take that, and we’re going to take this tool, and we’re going to deploy it to do that specifically, and not do anything else with it,” Justin said.

The technology’s vendor undermined this focus by coming in and telling them about all of the tool’s other features and functionality. There is a great deal of pressure on companies to start “bolting on features, just part of a go to market” strategy. Often the further a company deviates from its original technical vision, the worst or poor performing the new features become, “just like patchwork.”

Another manifestation of stupidly simple touches on one of the core issues of software engineering: should a feature be exposed as a service or as a library that’s linked against it. The pros of a service are the large degree of control you have over how the client’s code runs within the service. The cons are that when you need to make a change, you have to talk to everyone to get their buy-in. The pros of a library is giving the developer complete control over every aspect. The cons of a library is putting too much power in the hands of potential uninformed users.

Criteo learned this lesson with their Scala scheduler. Justin said, “Whenever we need to make changes to it, it became a quarters-long battle.” One big question was who owned support or debugging failures. The stack and its potential sources of problems were deep. “That adds toil to the team as well. That, in large part, becomes more obvious when there’s a strong separation of client code and runtime. Once you get to a certain scale, I think you are almost obliged to switch to an”as a service model" as opposed to an SDK model."

Friction and Getting Into Production

In distributed systems, there is always the question of “will it scale?” Data engineering teams are faced with the difficult balancing act between making sure data products scale and not taking so long to implement the feature that they fail to provide business value. Failing to achieve this balance is how many projects get canceled. They’re too challenging technically, and they won’t deliver business value quickly enough. The business is rapidly changing, and a data engineering team that can’t keep up will be a problem.

At Criteo, it started becoming too difficult to design and get something into production. Many gatekeepers and design reviews found flaws that teams had to spend even more time designing and verifying their designs. The resulting delays led to stretched timelines. Whenever there was a new idea, there was too much process “that built a lot of anxiety into that process. And then it builds friction as well because folks just want to get stuff out the door, and you have these people saying no.”

The saying “no” eventually caught up from a business point of view. “when the growth engine stalled and it’s clear that when you do lots of new things.” The technical stakeholders “were stuck in this mindset about protecting the castle [..] that became a conflict at the end.” Justin wanted to be seen as a facilitator and “no more opaque agenda.”

Toiling

Justin introduced an observation about why data engineering teams have difficulties or high turnover rates. It comes down to toil, and data engineering is just plain hard. It isn’t just difficult in one part of the puzzle. It’s difficult at several levels all at once.

Justin says, “People I think a lot of people run to data engineering because they like this idea of big data.” The management team learns about new technologies and what them make possible. Vendors consistently leave out how difficult they are or do the opposite and say they’re easy. The vendors leave users with the perception that “it’s just building Legos together, and we’re off and running.”

Vendors give management unrealistic expectations, “but then once data engineers understand what it means to operate large scale data pipelines and the ratio of code to operations [...] they burn out pretty quickly.” These expectations get added to unrealistic deadlines, and the team feels incredible pressure. “They build amazing amounts of technical debt that become very difficult to unwind.”

All of these issues become compounding factors. “Your organization needs to understand that it’s going to take three times as much time as you think.”

Cultural Differences

In any multinational company, there will always be cultural differences. For François, this involved being French and working with American colleagues. Managers need to realize these differences and adapt their methods of communication.

François had to internalize the difference in how American culture deals with disagreements. Once François told the team, “we should do it this way. They (American team) immediately followed all those. And then one month after that, you realize that actually we disagreed, but didn’t feel like they were in a place where they could disagree with me.” In French culture, disagreements are handled differently. François had to learn that cultural difference.

The differences weren't just around decisions. The French have a different level of passion than other cultures. "We tended to be a little more passionate about the things we want to do and don't want to do," François said. And at the same time, we don't bother about the fact that other people are passionate."

Changes Due To COVID-19

COVID-19 has brought about significant changes in organizational behavior. Data teams aren't immune from these changes.

François was forced to reassess their tooling and processes. The processes were clear enough when the data engineers were physically together, and they could quickly ask clarifying questions. The processes and tools weren't clear enough for the newly remote engineers who had more difficulty asking clarifying questions. You might have a perfect tool that doesn't integrate with the rest of the tooling. This forces users to navigate between their various tools. The forced navigation is often because different companies or teams within Criteo build the tools. This friction manifests itself as teams try to move something into production. The tools and processes are often undocumented or manual because it was easier to ask questions pre-COVID. François wanted the team to focus on data products as lifecycles, to provide a clear and consistent process for making a job production-worthy.

A significant change for Criteo was how they were used to being physically together for everything. When everyone was physically present, they could gather for in-person meetings and work together on whiteboards. They didn't have to document decisions in detail because they could rely on tribal knowledge. Even before COVID-19, this was an issue for remote workers. Now, the teams are focused on documenting more and sharing knowledge that is written down.

It has been difficult for the team leads, who are forced to lead their teams remotely when they've never done this before. They were used to leading with an in-person team. No one had ever explained how to lead a virtual team.

The changes due to COVID-19 aren't all bad. Now, developers are free to concentrate on their tasks without worrying about interruptions. When developers are in the office, there can be constant distractions from coworkers and other noises. When working remotely, the developer can choose the times to be distracted. They can force their coworkers to schedule a meeting instead of tapping them on the shoulder. François already sees the resulting productivity increase, recognizing that several projects took far less than initially planned.

The focus on improved communication is paying dividends. “They are clarifying things that were informal. If you have a design discussion, we have to explicitly set a meeting that’s called ‘design discussion for feature X,’ and everybody is invited. And in the end, you have a document that’s logged in Confluence (a wiki) that summarizes the design discussion we had on Feature X.”

The written documentation is already helping new members of the team come up to speed. They’re able to read about the design discussion and the rationale behind it. Writing down and documenting decisions should have always been done, and leaders should get used to enforcing this.

François likens the changes to being a Sergeant in a war. COVID-19 has changed the way the battle is fought. Instead of being on the front line with your soldiers, you are in a separate room. As the Sergeant, you have to direct your people differently. The dev leads (Sergeants) need to spend more time mentoring their teams with one-on-ones and giving feedback. “I really think that in the end, it changes the way we work, but not for the worse.”

Effective Post Mortems

François strongly believes in doing post mortems. Some companies do them only after a catastrophic failure. He believes that they should be done more often.

These post mortems require a great deal of honesty among the team–the courage to say, “I messed up on this and share that with the team.”

Some companies focus only on sharing their successes or glossing over their failures. There are some cultures where accepting or receiving criticism isn’t particularly acceptable. François credits his roots in French culture for making it easier for people to share failures openly.

Advice

Justin credits using HDFS as a single source of truth as a significant win in scaling things on a shoestring budget. Using microservices effectively allowed them to maintain more atomic work units. “When taken to an extreme and tightly coupled, everyone builds these huge monoliths that are very difficult to understand. Whereas when you separate everything out, it becomes very obvious what's going on, with the right runtime frameworks and instrumentation." Justin said.

Justin points out that there are individual wins and team wins. "I think we were a factory for producing senior engineers and leaders." Some of Criteo's engineers became Directors of Data or a related field. "They were always impressive folks, but now they have the credentials and the kind of resume to prove it."

Criteo's reputation really helped get top-notch engineers interested in working there. They gained an excellent reputation in France and "that definitely brought the most interesting talent in the door in France." Justin said.

Justin recalls "in the end, and after early missteps, they finally did a solid job of communicating vision internally." Part of sharing this vision was giving the team the space and opportunity to critique it. This opportunity created the internal alignment of 40+ engineers and the business stakeholders. "We even got sister teams outside of our organization to pound the pavement for us on our behalf for what we were doing and things like that."

Justin warns against letting an organization calcify. One manifestation of this problem could be a zero-tolerance policy on failure. These zero-tolerance policies usually aren't written out explicitly but are internalized by teams when they see management's reaction to failure. When trying new or challenging tasks that are prevalent in big data, teams and projects will fail. Management will need to understand the implications of failures, where some failures are "inconsequential, and if it fails, it's just some lost time on the part of the team, and there's a slight chance in that it works." These are some of the vital growth factors that enable or constrain a team.

Justin recommends spending time with stakeholders. This interaction has to be collaborative but not dictatorial. At Criteo, Justin's team "consider ourselves to be the experts on the business and producing data. Not necessarily what goes into the data, but in the methodology and the sort of best practices around it. And more often than not, we were right." This wasn't the right approach as that "was alienating to large groups of people that should have been our biggest sponsors and have been getting stuff out the door." He recommends looking at the tools you're building, and the level of skill for these tools. If they're made for experts and the users aren't experts, "they use them poorly. And so it's a self-reinforcing cycle that says, look, you don't know what the hell you're doing. You should listen to me in the first place. And this just increases the bias and is difficult to break out of that."

Problems in big data can seem deceptively simple to teams and the business. Vendors have created ample snake oil, talking about how simple their tools are to use. "Don't be deceived by that. Standing up a database has always been easy. But having a reliable data transmission engine [...] has always been hard." He goes on to point out, "It has always been a source of problems for organizations, even when the tools themselves were very easy to stand up and use. None of that has changed. The tools have changed. But the domain problem remains."

He explains, "So it's very important to seed the team with folks who know what they're doing. You can't just assume you can take random people and put them in. There's so many people who claim to be big data engineers, but then they have no depth of expertise. The critical domains of expertise would be software engineering, distributed systems, big data, as well as understanding of the business domain."

Talking more about people, Justin recommends having "experienced data engineers who have built pipelines and have failed and have learned from their bumps and bruises. That's what I mean by domain expertise." When using or finding new engineers, find someone who "has a desire at least to understand the business that you're in and the capacity to do so." They'll need a "solid software engineering background" that "revolves around data." It's essential to have "at least a seed in the organization of somebody who has a strong understanding of that."

François recommends checking some of our assumptions about what is usable. It's common for business analysts to use Excel. Although you don't want to use Excel to build a data platform, you may want to make data easily accessible as an Excel spreadsheet. Data teams inevitably have to turn experiments in Excel or notebooks into something production worthy.

François offers some advice on hiring. He believes "there is no single profile that fits being a data engineer." Their data engineers come from "different backgrounds." When he's asked to advise on how to start data teams, people are "surprised that the first thing I mentioned is not picking the cloud provider or the Hadoop distribution." Instead, he tells them to "just recruit the right people, and the right people are difficult to find." He attributes much of Criteo's success to the different people they hired.

François sees big data projects fail when they're looked at only from a "technology perspective," where the business just asks a technical person to "build my Hadoop." Companies are hiring teams just to keep up because other companies are hiring data teams. Someone in finance is just buying storage and compute power. "They all fail. And that's that's magic. But they keep doing it." These continued failures harken back to the 1960s when every company was trying to use the first databases. They just wanted a database but didn't understand or know how they're going to use it.

Instead, companies should focus on hiring the right people. Those people will eventually choose the right technologies and path. They need to focus and figure out what they cannot do due to the scale of the data. By doing this, companies and teams can learn from their failures and stop repeating them.

Data Engineering Teams

Creating Successful Big Data Teams and Products

Get expert guidance on how to create, staff, and run your data engineering team. While many sources say that you need a Big Data strategy, few of them talk about how to run a successful Big Data project or create a team that is capable of getting a product into production.

This book takes years of mentoring, teaching, and consulting with teams all over the world and distills it down to an approachable book. It is a practical and field tested guide to making your team successful with Big Data projects.

Here is what you'll learn in *Data Engineering Teams*

• Why do some teams succeed and why do some fail?
• Why is it absolutely crucial that data engineering teams understand and flesh out the use cases?
• What are the exact steps to follow when creating and executing a Big Data project?
• Why is Big Data so much more complicated than small data?
• What is your team's probability of success when creating your Big Data project?
• Why do some teams appear stuck and unable to make progress?
• What is a qualified Data Engineer and why is it crucial to have one on your team?
• When should a data engineering team start looking at and choosing the technology stack?
• What is a data engineering team and which skills does a data engineering team need?

Jesse Anderson is a Data Engineer, Creative Engineer and Managing Director of Big Data Institute.

He works with companies ranging from startups to Fortune 100 companies on Big Data. This includes training on cutting edge technologies like Apache Kafka, Apache Hadoop and Apache Spark. He has taught over 30,000 people the skills to become data engineers.

He is widely regarded as an expert in the field and for his novel teaching practices. Jesse is published on O'Reilly and Pragmatic Programmers. He has been covered in prestigious publications such as The Wall Street Journal, CNN, BBC, NPR, Engadget, and Wired.

ISBN 978-1-7326496-0-6
90000
9 781732 649606

Big Data Institute
www.bigdatainstitute.io