What is Data Engineering?

by **Gergely Orosz**

Home
Newsletter
My Books
Reading List
Popular Articles
Ethics statement
Sponsoring
Investing
Contact me

👋*Hi, this is Gergely with a free issue of the Pragmatic Engineer Newsletter. Today we cover Part 1 of 'What is Data Engineering.' To get a similarly in-depth article every week, subscribe to The Pragmatic Engineer Newsletter.*

Q: I'm hearing more about data engineering. As a software engineer, why is it important, what's worth

knowing about this field, and could it be worth transitioning into this area?

This is an important question as data engineering is a field that is without doubt, on fire. In November of last year, I wrote about what seemed to be a Data Engineer shortage in the issue, More follow-up on the tech hiring market:

> "Data usage is exploding, and companies need to make more use of their large datasets than ever. Talking with hiring managers, the past 18 months has been a turning point for many organizations, where they are doubling down on their ability to extract real-time insights from their large data sets. (...)
>
> What makes hiring for data engineers challenging is the many languages, technologies and different types of data work different organizations have."

To answer this question, I pulled in Benjamin Rogojan, who also goes by Seattle Data Guy, on his popular data engineering blog and YouTube channel.

Ben has been living and breathing data engineering for more than 7 years. He worked for 3 years at Facebook as a Data Engineer and has gone independent following his work there. He now works with both large and small companies to build out data warehousing, developing and implementing models, and takes on just about any data pipeline challenge.

Ben also writes the SeattleDataGuy newsletter on Substack which is a publication to learn about end-to-end data flows, Data Engineering, MLOps, and Data Science. Subscribe here.

In this article, Ben covers:

1. What do data engineers do?
2. Data engineering terms.
3. Why data engineering is becoming more important.

In part 2 - coming next week and already out for full subscribers - we will additionally cover:

- Data engineering tools: an overview
- Where is data engineering headed?
- Getting into data engineering as a software engineer

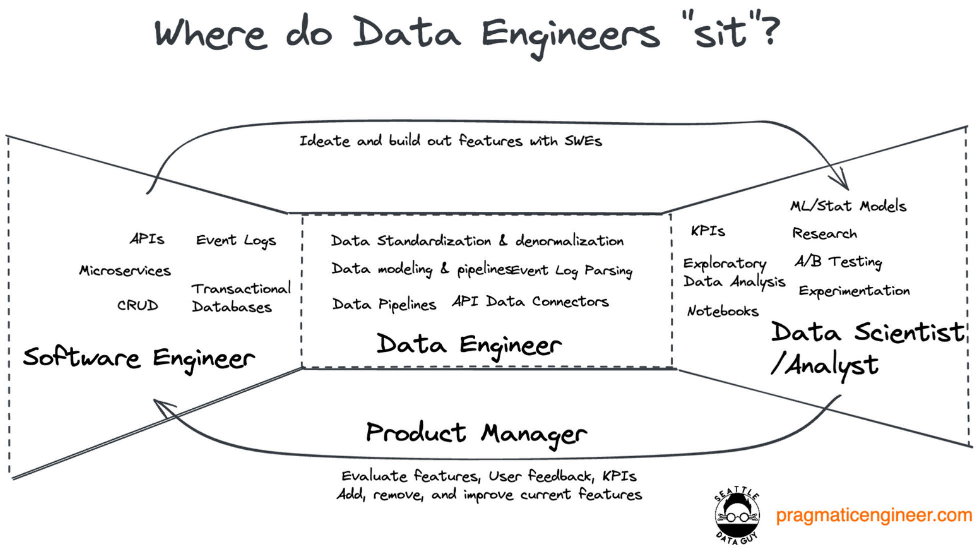
With that, over to Ben:

For the past near decade I have worked in the data world. Like many, in 2012 I was exposed to HBR's Data Scientist: The Sexiest Job of the 21st Century. But also like many, I found data science wasn't the exact field for me. Instead, after working with a few data scientists for a while I quickly realized I enjoyed building data infrastructure far more than creating Jupyter Notebooks.

Initially, I didn't really know what this role was that I had stumbled into. I called myself an automation engineer, a BI Engineer, and other titles I have long forgotten. Even when I was looking for jobs online I would just search for a mix of "SQL", "Automation" and "Big Data," instead of a specific job title.

Eventually, I found a role called "data engineer" and it stuck. Recently, the role itself has been gaining a little more traction, to the point where data engineering is growing more rapidly than data science roles. Also, companies like Airbnb have started initiatives to hire more data engineers to increase their data quality.

But what is a data engineer and what do data engineers do for a company? In this article, we dive into data engineering, some of its key concepts and the role it plays within companies.



Where do data engineers "sit"? They're typically working with software engineers and data scientists, but much less with product managers. In this article, we dive deeper into the data engineering field.

1. What do data engineers do?

How do you define data engineering? Here's how data engineer Joe Reis specifies this term in his recently released book, Fundamentals of Data Engineering:

> "Data engineering is the development, implementation, and maintenance of systems and processes that take in raw data and produce high-quality, consistent information that supports downstream use cases, such as analysis and machine learning.
>
> Data engineering is the intersection of security, data management, DataOps, data architecture,

> orchestration, and software engineering. A data engineer manages the data engineering lifecycle, beginning with getting data from source systems and ending with serving data for use cases, such as analysis or machine learning."

In short, data engineers play an important role in creating core data infrastructure that allows for analysts and end-users to interact with data which is often locked up in operations systems.

For example, at Facebook there is often either a data engineer or a data engineering team which supports a feature or business domain. Teams that support features and products are focused on helping define which information should be tracked, and then they translate that data into easy-to-understand core data sets.

A **core data set** represents the most granular breakdown of the transactions and entities you are tracking from the application side. From there, some teams have different levels of denormalization they might want to implement. For example, they might want to denormalize if they remove any form of nested columns to avoid analysts having to do so.

Also, many teams will set standards on naming conventions in order to allow anyone who views the data set to quickly understand what data type a field is. The basic example I always use is the “is_” or “has_” prefix denoting a boolean. The purpose of these changes is to treat data as a product; one that data analysts and data scientists can then build their models and research from.

Our team at Facebook produced several core data sets that represented recruiting and People data. These core data sets allowed analysts and data scientists to see the key entities, relationships and actions occurring inside those business domains. We followed a core set of principles which made it clear how data should be integrated, even though it was being pulled from multiple data sources, including internally developed products and Saas solutions.

What are the goals of a core data set? Here are the three main ones.

1. Easy to work with. Datasets should be easy to work with for analysts, data scientists and product managers. This means creating data sets that can be easily approached without a high level of technical expertise to extract value from said data. In addition, these data sets standardize data so that users don’t have to constantly implement the same logic over and again.

2. Provide historical perspective. Many applications store data which represents the current state of entities. For example, they store where a customer lives or what title an employee has. Not all applications store these changes. In turn, data engineers must create data that represent this.

The traditional way to track historical changes in data was to use what we call Slowly Changing Dimensions (SCD). There are several different types of SCD, but one of the simplest to implement is SCD type 2 which has a start and end date, as well as an "is_current" flag.

An example of an SCD is a customer changing their address when they move home. Instead of just updating the current row which stores the address for said customer or employee, you will:

1. Insert a new row with the new information.
2. Update the old row, so it is no longer marked current.
3. Ensure the end date represents the last date when the information was accurate.

This way, when someone asks, "how many customers did we have per region over the last 3 years," you can answer accurately.

3. Integrated. Data in companies come from multiple sources. Often, in order to get value from said data, analysts and data scientists need to mesh all the data together, somehow. Data engineers help by adding IDs and methods for end-users to integrate data.

At Facebook, most data had consistent IDs. This made it feel like we were being spoiled, as consistent IDs made it very easy to work with data across different sets.

When data is easy to integrate across entities and source systems it allows analysts and data scientists the ability to easily ask questions across multiple domains, without having to create complex – and likely difficult to maintain – logic to match data sets. Maintaining custom and complex logic to match data sets is expensive in terms of time and its accuracy is often dubious. Rarely have I seen anyone create a clean match across data that's poorly integrated.

One great example I heard recently was from Chad Sanderson of Convoy. Chad explained how a data scientist had to create a system to mesh email and outcome data together and it was both costly and relied on fuzzy logic which probably wasn't as accurate as possible.

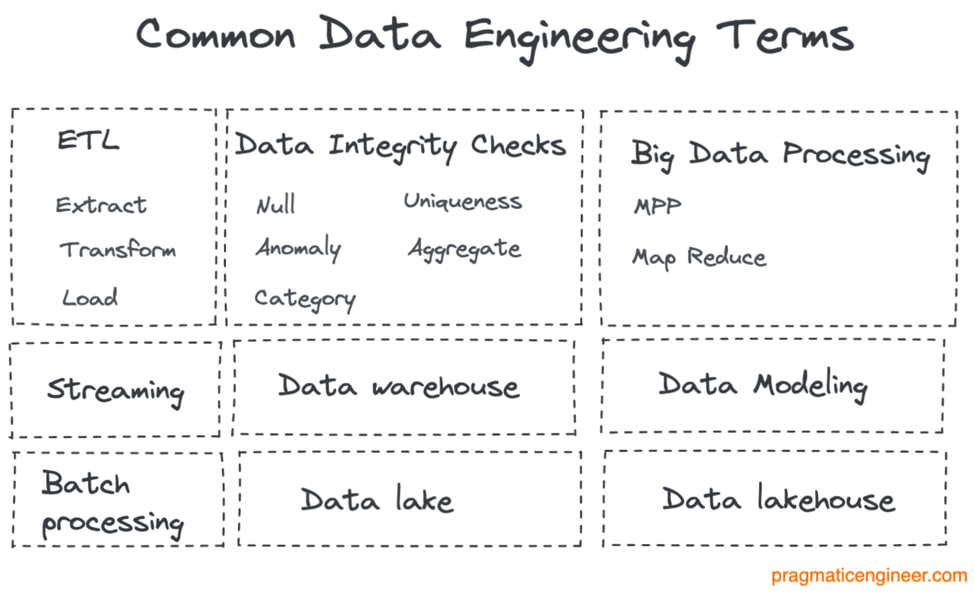
At Facebook, even systems like Salesforce, Workday and our custom internal tools, all shared these consistent IDs. Some used Salesforce as the main provider and others used internal reporting IDS. But it was always clear which ID was acting as the unique ID to integrate across tables.

But how can data engineers create core data sets which are easy to use?

Now we have discussed the goal, let's outline some of the terms you'll hear data engineers use to make your data more approachable.

2. Data engineering terms

Let's explain some commonly used data engineering terms.



Some of the more common data engineering terms

ETL\ELT\Data Pipelines

You will often hear data engineers describe most of their job as moving data from point A to point B. We do this using data pipelines.

Data pipelines are generally structured as either an Extract, Transform, Load or an Extract, Load, Transform (ETL vs ELT.) Of course, there are other design patterns we may take on, such as event pipelines, streaming and change data capture (CDC.) This is why many of us often just generalize and use the term 'data pipelines.' However, most pipelines are in the form of the steps below.

E – Extract. The extract step involves connecting to a data source such as an API, automated reporting system or file store, and pulling the data out of it.

For example, one project I worked on required me to pull data from Asana. This meant I needed to create several components to interact with Asana's multiple API endpoints, pull out the JSON and store it in a file service.

I will say Asana's API is not built for bulk data extracts. There were many cases where I would have to get all the new project IDs, as well as the current ones I had in my current table, in order to then set up a second set of calls to use the project ID to get all the tasks attached to said projects. This was plainly far more cumbersome than just saying, “give me all the tasks in this organization.”

T – Transform. The transform step (especially in an initial pipeline) will likely standardize the data (format dates, standardize booleans, etc.,) as well as sometimes starting to integrate data by adding in IDs which are also standardized, deduplicating data and adding in more human readable categories.

Transforms can also be more complex, but the examples above are standard. One example of this was that with all the various core data sets our team was managing, we always defined which ID was the core ID for a particular data set. Whether the core ID was an internal Facebook ID or an external Saas ID, we would be clear which one we used as “core”.

Because we were clear on the core ID, if we ended up having multiple IDs in a table, we would remove them to avoid further confusion, downstream.

From my own philosophical perspective, creating data as a product means creating a product someone with limited knowledge of your team's data can approach and understand. By understanding, I mean they can quickly grasp what fields mean and how one table relates to another.

Understanding the data set is an important step in a baseline transform. There are more complex transforms which can occur for analytical purposes.

L – Load. This step is meant to load data into a table in the data warehouse. There isn't anything fancy here. At Facebook, we loaded into our team's "data warehouse," but you could be loading into Snowflake, Databricks, Redshift, Bigquery or others.

Data Modeling

Software engineers who have built their own database will be familiar with some aspects of data modeling. Generally, for what is known as an OLTP (online transaction processing,) these systems are developed to be fast and robust for single transactions.

However, these data models pose problems when the data is being used for analytics.

The data in these models requires a lot of joins to get to an obscure field which a data analyst wants to know about. Also, the data model isn't usually developed to support billions of row aggregations and calculations with good performance. These models tend to be heavily normalized and in turn require heavy amounts of processing to answer even simple questions.

In return for data sets being heavily normalized, data modeling for data engineers is a combination of adding missing abilities for these data sets. By missing abilities, I mean the ability to track historical data, improving performance of analytical queries and simplifying the data model into a much more straightforward set of tables. Common end-states will be referred to as snowflake, star, activity or OBT schemas.

Data Integrity Checks

Tracking every single table and all its columns isn't feasible for a single data engineer. When I worked at Facebook, I had well over 300 tables for which I was listed as the owner. To make matters worse, bad data can enter a column and not fail because it's the same data type. Especially in the case when you're loading data via the order they come in, versus (vs) explicit name calls.

A software engineer upstream could change a source, which in turn changes the order or removes the column altogether. This change could lead to data being loaded improperly but not failing, if the data is the same data type.

Data integrity checks are a crucial first line of defense for detecting data issues like these I've just described. There are several traditional types of data integrity checks.

- **Null Checks** - These checks calculate what percentage of a column is null and can then be set to a threshold value to go off when a column has too many nulls.
- **Anomaly Checks** - Anomaly checks can be used to both check specific column values as well as metadata about a table, such as how many rows were just inserted. These checks aim to detect drastic changes in either the fields or row counts. If, for example, a table suddenly has 10x the number of rows compared to yesterday, then perhaps there is a problem.
- **Category Checks** - Many fields represent enumerators or categories which should have only specific results. One example I always use is when I worked at a company that had a "State_Code" field. You'd assume this field only provided valid states as it was most likely filled via a drop-down menu. However, we found there were errors from time to time which weren't valid states. So, we needed a data check in place to catch these issues.
- **Uniqueness Check** - Joining data across multiple granularities and data sets risks

creating duplicate rows in your data warehouse; even if you removed some duplicates, earlier. They can be easily re-introduced with the wrong join, so creating a check in your final core data layer is key to ensuring you have the correct level of unique rows. Especially when a lot of modern data warehouses don't provide the unique column constraint.

- **Aggregate Checks** - As data gets processed through multiple layers of data transforms, there is a chance that removal or changing of data can occur. In turn, creating checks which calculate aggregates such as total sales, row counts or unique customer counts is important because they detect any major changes or removals of data that occur.

Streaming Vs Batch Processing

A common question data engineers need to answer is this: does a pipeline need to be streamed or batch processed?

Batch processing is when you have a data pipeline which runs at a normal cadence, usually every hour, or daily.

At Facebook, most of our jobs would run at around midnight. We would use Dataswarm (which is similar to Airflow,) to set a scheduler using a cron-like configuration for how often the job should run. An interesting point here is that some schedulers struggle with certain constraints, such as daylight savings. I've worked on a few projects where twice a year there was a need to rerun pipelines to deal with issues caused by the pipeline running in an unexpected pattern.

Streaming involves ingesting and sometimes transforming data as soon as an event occurs.

For example, on one project we set up a Kafka topic which streamed events directly to Snowflake through its Kafka connector. This allowed the raw tables to constantly be up to date. This was important to the client as their service was providing a real-time utility that was directly connected to people's day-to-day interactions. Their users both needed to be able to use their product in real-time, as well as glean information and insights from all the actions and machine learning models occurring across the platform.

It's important to understand the use case before implementing a streaming data process. To fully process streamed data into a pipeline requires a lot more technical know-how, as well as contingencies if something goes wrong, and it can be far more difficult to recover. Whereas with batch data pipelines, if there is an error, it's pretty easy to rerun the data and the next time the pipeline will need to run is the following day.

In order to know which of these two pipeline styles your team should use, you want to understand how your end-users plan to use the data, how much data is coming in, as well as the natural state of the system you're pulling the data from. I was on a call recently with a client where initially they stated they wanted the data to be updated in real-time, then it went to every fifteen minutes and when I asked how often their users would be looking at the data, they said once a week. Not a great fit for investing in real-time data and I'd say I have a lot of conversations like this.

This isn't to say real-time isn't necessary and it has become far easier. Similar to the way processing large datasets has become far easier, thanks to a combination of the cloud and the popularization of solutions such as Hadoop.

Big Data Processing

Big Data. Some feel the phrase is more of a marketing term and others believe it is the solution to developing reliable machine learning models. At the end of the day, I often refer to it as a big problem, for which we have developed solutions. Big Data on its own was often expensive and difficult to manage, especially on-site. It required constant migrations to larger physical servers and would often limit how many queries could truly be run on a machine.

In turn, many of the techniques centered around Big Data are meant to ease some of these problems.

- **MPP (massively parallel processing)** - is a processing paradigm which as the name suggests, takes the idea of parallel processing to the extreme. It uses hundreds or thousands of processing nodes to work on parts of a computational task in parallel. These nodes each have their own I/O and OS and don't share memory. They achieve a common computational task by communicating with each other over a high-speed internet connection.
- **Map Reduce** - is another processing paradigm that can sometimes appear very similar to MPP. It also breaks down large amounts of data into smaller batches, and then processes them over multiple nodes. However, while Map Reduce and MPP appear similar, there are some distinct differences. In general, Map Reduce is done on commodity hardware, whereas MPP tends to be done on more expensive hardware. MPP also tends to refer to SQL-based query computations, while MapReduce is generally

more of a design paradigm most famously implemented in Java.

Data Warehouses

The concept of data warehouses has been around for decades. The purpose of data warehouses is to provide analysts and end-users data which tracks historical information and is integrated with multiple data sources.

A data warehouse is a central repository of information that can be analyzed to make more informed decisions. Data flows into a data warehouse from transactional systems, relational databases and other sources, typically on a regular cadence. Business analysts, data engineers, data scientists, and decision makers access the data through business intelligence (BI) tools, SQL clients, and other analytics applications.

Data Lakes

The term 'data lake' was coined around 2010. Data lakes became popular because they offer a solution to the rapidly increasing size and complexity of data. As defined by TechTarget:

A data lake is a storage repository that holds a vast amount of raw data in its native format until it is needed for analytics applications.

In addition, data lakes often provide a cheaper option in terms of data storage and computation, since they are often developed on cheaper hardware. In contrast, data warehouses are highly structured and often run on expensive hardware.

Data lakes provide the ability to store data which can have nearly no structure whatsoever, relying on the end-user to provide the schema on-read. What data lakes look like has changed rapidly from the days of Hadoop, as solutions like Delta lake are providing a very different approach to data lakes.

Data Lake Houses

Data lake houses has been a popular term recently, spearheaded by Databricks. The purpose of this paradigm is to balance the benefits of a data warehouse and data lake into a single solution.

For example, data lake houses should provide ACID (Atomicity, Consistency, Isolation, and Durability) transactions like a data warehouse while balancing the flexibility and scale of a data lake.

Newsletter

Enjoying this article? Subscribe to my newsletter to get issues like this in your inbox. It's a good read and the #1 technology newsletter on Substack.

3. Why data engineering is becoming more important

Data has grown in size, speed and complexity. Over the past decade, the complexity of data has vastly increased. In the past, most transaction systems mostly tracked baseline information. For example, they tracked information like what someone purchased at a store, or which website they visited.

Nowadays, even basic applications can track additional information or provide even deeper functionality, which is then tracked. Tracking additional information leads to much more complex events and transactions being stored. Let’s add that users now interact with mobile and IoT devices most of the day, which increases the sheer volume and has led to what’s referred to as the “5 Vs” of Big data, velocity, volume, value, variety and veracity.

These increasing dimensions require better tooling and more specialized expertise in how to handle all this data. However, it's not just the supply of data that's increased. It's also the demand.

Everyone wants access to their data. With the increasing number of analysts and data scientists, as well as the proliferation of SQL-literate employees, the demand for data has grown. Add to this the fact that today pulling and managing data is not limited to large organizations which can afford all the administration staff. Now, a small company can pay for what it uses in terms of data warehousing, storage and computation.

Tools like Snowflake and Bigquery make storing large amounts of data considerably easier and cheaper – when set up correctly. Add in the fact that tools like Salesforce and Shopify make it easier for end-users to pull data out from them. All of this has led to companies of all sizes pushing to create data storage systems for analytics. Overall – especially in the past decade with the mass adoption of the cloud – data has just become that much easier to manage and analyze.

This was it for part one. Subscribers can already read the rest of the article. We'll wrap up with part two next week: subscribe to get it in your inbox, or bookmark this page and check back for it!

Subscribe to my weekly newsletter to get articles like this in your inbox. It's a pretty good read - and the #1 tech newsletter on Substack.

The Software Engineer's Guidebook

I wrote The Software Engineer's Guidebook. Here is what Tanya Reilly, senior principal engineer and author of The Staff Engineer's Path says about it:

> "From performance reviews to P95 latency, from team dynamics to testing, Gergely demystifies all aspects of a software career. This book is well named: **it really does feel like the missing guidebook for the whole industry.**"

The Software Engineer's Guidebook
Gergely Orosz
Navigating senior, tech lead, and staff engineer positions at tech companies and startups

The Pragmatic Engineer

Get the book here.

Annual conference talk

I do one conference talk every year. In 2024, this was the keynote at Craft Conference. You can now watch the full talk, online:

The software engineering industry in 2024: what changed in 2 years, why, and what is next.

Gergely Orosz

Last updated 13 SEPTEMBER 2022. Originally published 13 Sep 2022.

Writing The Pragmatic Engineer Newsletter. Author of The Software Engineer's Guidebook. Previously at Uber, Microsoft, Skype, Skyscanner.

Amsterdam, Netherlands

READ THIS NEXT

Inside Pollen's Collapse: "$200M Raised" but Staff Unpaid - Exclusive

Exclusive details about the collapse of the formerly high-flying startup. Could tech employees have known sooner they were on a sinking ship?...

YOU MIGHT ENJOY

Inside the Shutdown of Zenly by Snap

Snap suddenly decided to shut Zenly down: an app with 40M monthly active users, growing fast. What happened, and what can other acquisitions learn from the fate of Zenly?...