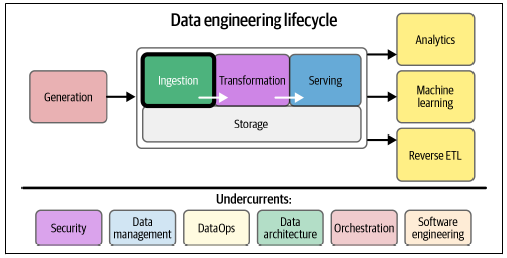
# Fundamentals of Data Engineering - Reis & Housley

## Part II. The Data Engineering Lifecyle in Depth

### Chapter 7 – Ingestion

* You’ve learned about the various source systems you’ll likely encounter as a DE + about ways to store data
* Let’s now turn our attention to the **patterns + choices that apply to ingesting data from various source systems**
* We discuss **Data ingestion**, the key engineering considerations for the ingestion phase, the major patterns for batch + streaming ingestion, tech you’ll encounter, whom you’ll work w/ as you develop your data ingestion pipeline, + how the undercurrents feature in the ingestion phase



#### What is Data Ingestion?

* **Data ingestion** = the **process of moving data from one place to another**
* It **implies data movement from source systems into storage** in the DE lifecycle, with **ingestion as an intermediate step**
* It’s worth quickly contrasting data ingestion with data *integration*
* Whereas **data ingestion**is **data *movement*** from point A to B, **data integration****combines data from disparate sources into a new dataset**
* Ex: You can use data integration to combine data from a CRM system, advertising analytics data, + web analytics to create a user profile, which is saved to your DW
* Furthermore, using **reverse ETL**, you can send this newly created user profile *back* to your CRM so salespeople can use the data for prioritizing leads
* See Chapter 8 for more about data integration + Chapter 9 for more about reverse ETL
* We also point out that **data ingestion is different from internal ingestion *within a system***
* **Data stored in a database is copied from one table to anoth**er, or **data in a stream is temporarily cached**
* Consider this another part of the general data transformation process covered in Chapter 8

##### Data Pipelines Defined

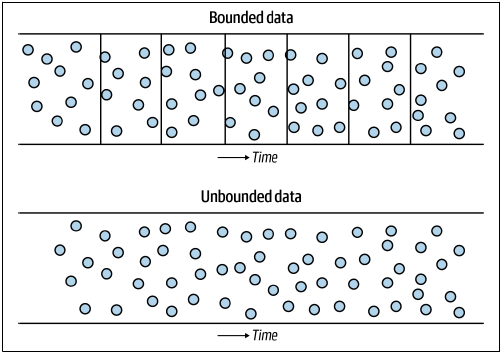
* **Data pipelines** *begin* in source systems, but **ingestion is the stage where DE’s begin actively designing data pipeline activities**
* In the DE space, a good deal of ceremony occurs around **data movement** **+ processing patterns**, w/ established patterns such as **ETL**, newer patterns such as **ELT**, and new names for long-established practices (**reverse ETL**) + **data sharing**
* **All the above concepts are encompassed in the idea of a data pipeline**
* **It is essential to understand the details of these various patterns + know that a modern data pipeline includes all of them**
* As the world moves away from a traditional monolithic approach w/ rigid constraints on data movement, + **toward an open ecosystem of cloud services that are assembled like LEGO bricks to realize products, DE’s prioritize using the right tools to accomplish the desired outcome over adhering to a narrow philosophy of data movement**
* In general, here’s a definition of a data pipeline: ***A data pipeline is the combination of architecture, systems, + processes that move data through the stages of the data engineering lifecycle***
* This definition is deliberately fluid (+ intentionally vague) to allow DE’s to plug in whatever they need to accomplish the task at hand
* A data pipeline **could be a traditional ETL system**, where **data is ingested from an on-prem transactional system, passed through a monolithic processor, and written into a DW**
* Or **could be a cloud-based data pipeline that pulls data from 100 sources, combines it into 20 wide tables, trains 5 other ML models, deploys them into production, + monitors ongoing performance**
* **A data pipeline should be flexible enough to fit any needs along the DE lifecycle**

#### Key Engineering Decisions for the Ingestion Phase

* When preparing to architect or build an ingestion system, here are **some primary considerations and questions to ask yourself related to data ingestion:**
* What’s the **use case for the data** I’m ingesting?
* **Can I** **reuse this data** + avoid ingesting multiple versions of the same dataset?
* **Where** is the data **going**? What’s the **destination?**
* **How often** should the data be **updated from the source?**
* What is the **expected data volume?**
* **What format** is the data in?
* Can **downstream** storage and transformation **accept this format?**
* Is the source data in **good shape for immediate downstream use?**
* That is, is the **data of good quality?**
* What **post-processing is required** to serve it?
* What are **data-quality risks** (e.g., could bot traffic to a website contaminate the data)?
* Does the data **require in-flight processing** for downstream ingestion if the data is **from a streaming source?**
* These questions undercut batch + streaming ingestion and apply to the underlying architecture you’ll create, build, and maintain
* **Regardless of how often the data is ingested,** you’ll want to **consider the following factors** when designing your ingestion architecture

##### Bounded Vs. Unbounded Data

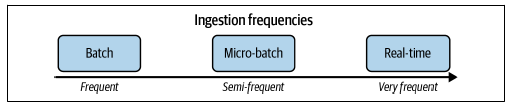
* Recall from Chapter 3, **data comes in two forms: bounded and unbounded**
* **Unbounded data**is **data as it exists in reality, as events happen, either sporadically or continuously, ongoing and flowing**
* **Bounded data**is a **convenient way of bucketing data across some sort of boundary, such as time**



* Let us adopt this mantra: **All data is unbounded until it’s bounded**
* Like many mantras, this one is not precisely accurate 100% of the time
* A grocery list scribbled this afternoon is bounded data, written as a stream of consciousness (unbounded data) onto a piece of scrap paper, where the thoughts now exist as a list of things (bounded data) one needs to buy at the grocery store.
* However, **this idea is correct for practical purposes for the vast majority of data you’ll handle in a business context**
* Ex: An online retailer will process customer transactions 24 hours a day until the business fails, the economy grinds to a halt, or the sun explodes
* **Business processes have long imposed artificial bounds on data by cutting discrete batches**
* **Keep in mind the *true* unboundedness of your data**
* Streaming ingestion systems are simply a tool for preserving the unbounded nature of data so that subsequent steps in the lifecycle can also process it continuously

##### Frequency

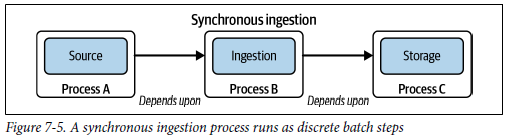
* **One of the critical decisions DE’s must make in designing data ingestion processes is the data-ingestion frequency**
* Ingestion processes can be **batch**, **micro-batch**, or **real-time**
* Ingestion frequencies vary dramatically from slow to fast



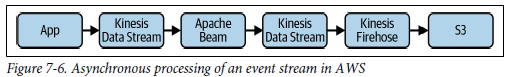
* On the slow end, a business might ship its tax data to an accounting firm once a year
* On the faster side, a CDC system could retrieve new log updates from a source database once a minute
* Even faster, a system might continuously ingest events from IoT sensors + process these w/in seconds
* **Data-ingestion frequencies are often mixed in a company, depending on the use case + technologies**
* Note that **“real-time” ingestion patterns are becoming increasingly common**
* “Real-time” is in quotation marks because no ingestion system is *genuinely* real-time
* Any database, queue, or pipeline has inherent latency in delivering data to a target system
* It is more accurate to speak of **near real-time**, but we often use *real-time* for brevity
* The **near real-time pattern generally does away with an explicit update frequency, as events are processed in the pipeline either one by one as they arrive or in micro-batches** **(i.e., batches over concise time intervals)**
* We will **use *real-time* and *streaming* interchangeably**
* **Even w/ a streaming data-ingestion process, batch processing downstream is relatively standard**
* As of 2022-2023, ML models are typically trained on a batch basis, although continuous online training is becoming more prevalent
* **Rarely do DE’s have the option to build a *purely* near real-time pipeline w/ no batch components**
* **Instead,** they **choose where batch boundaries will occur** (i.e., the DE lifecycle data will be broken into batches)
* **Once data reaches a batch process, the batch frequency becomes a bottleneck for all downstream processing**
* In addition, **streaming systems are the best fit for many data source types**
* In IoT applications, the typical pattern is for each sensor to write events or measurements to streaming systems as they happen
* While this data can be written directly into a database, a streaming ingestion platform such as Amazon Kinesis or Apache Kafka is a better fit for the application
* Software applications can adopt similar patterns by writing events to a message queue as they happen rather than waiting for an extraction process to pull events + state information from a backend database
* This pattern works exceptionally well for event-driven architectures already exchanging messages through queues
* And again, **streaming architectures generally coexist with batch processing**

##### Synchronous vs. Asynchronous Ingestion

* With **synchronous ingestion**, the **source, ingestion, + destination have complex dependencies and are tightly coupled**
* As you can see below, each stage of the DE lifecycle has processes A, B, + C directly dependent upon one another



* If process A fails, processes B and C cannot start, + if process B fails, process C doesn’t start
* **This type of synchronous workflow is common in older ETL systems, where data extracted from a source system must then be transformed before being loaded into a DW**
* **Processes downstream of ingestion can’t start until *all* data in the batch has been ingested**
* **If the ingestion or transformation process fails, the entire process must be rerun**
* Mini case study of how *NOT* to design your data pipelines
* At a company, the transformation process itself was a series of dozens of tightly coupled synchronous workflows, w/ the entire process taking over 24 hours to finish
* If any step of that transformation pipeline failed, the whole transformation process had to be restarted from the beginning
* In this instance, process after process failed, + b/c of nonexistent or cryptic error messages, fixing the pipeline was a game of whack-a-mole that took over a week to diagnose and cure
* Meanwhile, the business didn’t have updated reports during that time, + people weren’t happy
* With **asynchronous ingestion**, **dependencies can now operate at the level of individual events, much as they would in a software backend built from microservices**
* **Individual events become available in storage as soon as they are ingested individually**
* Ex: A web application on AWS that emits events into Amazon Kinesis Data Streams (here acting as a buffer)



* The stream is read by Apache Beam, which parses + enriches events, + then forwards them to a second Kinesis stream
* Then Kinesis Data Firehose rolls up events and writes objects to Amazon S3.
* The **big idea is that rather than relying on asynchronous processing, *where a batch process runs for each stage as the input batch closes and certain time conditions are met*, each stage of the asynchronous pipeline can process data items as they become available in parallel across the Beam cluster.**
* The **processing rate depends on available resources**
* The Kinesis Data Stream acts as the shock absorber, moderating the load so that event rate spikes will not overwhelm downstream processing
* Events will move through the pipeline quickly when the event rate is low, + any backlog has cleared
* NOTE: Could modify the scenario and use a Kinesis Data Stream for storage, eventually extracting events to S3 before they expire out of the stream

##### Serialization and Deserialization

* Moving data from source to destination involves **serialization** and **deserialization**
* As a reminder, **serialization**means **encoding the data from a source + preparing data structures for transmission + intermediate storage stages**
* **When ingesting data, ensure that your destination can deserialize the data it receives**
* Data can be ingested from a source but then sit inert + unusable in the destination b/c the data cannot be properly deserialized
* See the more extensive discussion of serialization in **Appendix A**

##### Throughout and Scalability

* *In theory*, ingestion should *never* be a bottleneck
* **In practice, ingestion bottlenecks are pretty standard**
* **Data throughput + system scalability become critical as your data volumes grow + requirements change**
* **Design your systems to scale + shrink to flexibly match the desired data throughput**
* ***Where* you’re ingesting data from matters a lot**
* *If receiving data as it’s generated,* will the upstream system have any issues that might impact your downstream ingestion pipelines?
* Ex: Suppose a source database goes down
* When it comes back online + attempts to backfill the lapsed data loads, will your ingestion be able to keep up with this sudden influx of backlogged data?
* Another thing to **consider is your ability to handle burst-y data ingestion**
* **Data generation rarely happens at a constant rate + often ebbs and flows**
* **Built-in buffering is required to collect events during rate spikes to prevent lost data**
* Buffering bridges the time while the system scales + allows storage systems to accommodate bursts even in a dynamically scalable system
* **Whenever possible, use managed services that handle the throughput scaling for you**
* While you can manually accomplish these tasks by adding more servers, shards, or workers, often this isn’t value-added work, + there’s a good chance you’ll miss something
* **Much of this heavy lifting is now automated**, so don’t reinvent the data ingestion wheel if you don’t have to

##### Reliability and Durability

* **Reliability and durability are vital in the ingestion stages of data pipelines**
* **Reliability** entails **high uptime + proper failover for ingestion systems**
* **Durability**entails **making sure that data isn’t lost or corrupted**
* Some data sources (e.g., IoT devices + caches) may not retain data if it is not correctly ingested
* Once lost, it is gone for good
* **In this sense, the *reliability* of ingestion systems leads directly to the *durability* of generated data**
* If data is ingested, downstream processes can theoretically run late if they break temporarily
* **Advice = Evaluate the risks + build an appropriate level of redundancy + self-healing *based on the impact + cost of losing data***
* **Reliability + durability have both direct + indirect costs**
* Ex: Will your ingestion process continue if an AWS zone goes down? How about a whole region? How about the power grid or the internet?
* ***Of course, nothing is free*. How much will this cost you?**
* You might be able to build a highly redundant system + have a team on call 24 hours a day to handle outages
* This also means cloud + labor costs become prohibitive (direct costs), + the ongoing work takes a significant toll on your team (indirect costs)
* There’s no single correct answer, + you need to evaluate the costs + benefits of your reliability + durability decisions
* **Don’t assume that you can build a system that will reliably + durably ingest data in *every* possible scenario**
* Even the nearly infinite budget of the US federal government can’t guarantee this
* In many extreme scenarios, ingesting data actually won’t matter
* There will be little to ingest if the internet goes down, even if you build multiple air-gapped data centers in underground bunkers with independent power
* **Continually evaluate the trade-offs and costs of reliability + durability**

##### Payload

* A **payload**is **the dataset you’re ingesting and has characteristics such as kind, shape, size, schema and data types, + metadata**

###### i) Kind

* The **kind****of data you handle directly impacts how it’s dealt with downstream in the DE lifecycle**
* **Kind consists of type and format**
* **Data has a type** (tabular, image, video, text, etc.), + the **type directly influences the data format or the way it is expressed in bytes, names, + file extensions**
* Ex: A tabular kind of data may be in formats such as CSV or Parquet, w/ each of these formats having different byte patterns for serialization + deserialization
* Another kind of data = image, which has a format of JPG or PNG + is inherently unstructured

###### ii) Shape

* **Every payload has a shapethat describes its dimensions** +is **critical across the DE lifecycle**
* Ex: An image’s pixel and red, green, blue (RGB) dimensions are necessary for training DL models
* Ex: If trying to import a CSV file into a database table, + your CSV has more columns than the database table, you’ll likely get an error during the import process
* Here are some examples of the shapes of various kinds of data:
* **Tabular:** Number of rows + columns in a dataset, commonly expressed as *M* rows, *N* columns
* **Semi-structured JSON:** The key-value pairs + nesting depth occur with sub-elements
* **Unstructured text:** Number of words, characters, or bytes in the text body
* **Images:** The width, height, and RGB color depth (e.g., 8 bits per pixel)
* **Uncompressed audio:** Number of channels (e.g., 2 for stereo), sample depth (e.g., 16 bits per sample), sample rate (e.g., 48 kHz), + length (e.g., 10,003 seconds)

###### iii) Size

* The **size of the data describes the number of bytes of a payload, which may range in size from single bytes to terabytes + larger**
* **To *reduce* size of a payload, it may be compressed into various formats such as ZIP and TAR** (see discussion of compression in Appendix A)
* **A massive payload can also be split into chunks, which effectively reduces the size of the payload into smaller subsections**
* When loading a huge file into a cloud object storage or a DW, this is a common practice as the small individual files are easier to transmit over a network (especially if they’re compressed)
* The smaller chunked files are sent to their destination + then reassembled after all data has arrived

###### iv) Schema and Data Types

* **Many data payloads have a schema, such as tabular and semi-structured data**
* As mentioned earlier, a **schema describes the fields + types of data w/in those fields**
* **Other data, such as unstructured text, images, and audio, will NOT have an explicit schema or data types**
* However, they might come w/ technical file descriptions on shape, data + file format, encoding, size, etc.
* Although you can connect to databases in various ways (file export, CDC, JDBC/ODBC, etc.), **the** **connection is easy**, while **the great engineering challenge is understanding the underlying schema**
* **Applications organize data in various ways, + DE’s need to be intimately familiar with the organization of the data + relevant update patterns to make sense of it**
* **This current problem has been somewhat exacerbated by the popularity of object-relational mapping (ORM)**, which **automatically generates schemas based on object structure** in languages such as Java or Python
* **Natural structures in an OOP language often map to something messy in an operational database**
* **DE’s may need to familiarize themselves with the class structure of application code**
* ***Schema is not only for databases***
* As discussed, **APIs present their schema complications**
* Many vendor APIs have friendly reporting methods that prepare data for analytics
* In other cases, DE’s are not so lucky
* The API is a thin wrapper around underlying systems, requiring DE’s to understand application internals to use the data
* **Much of the work associated w/ ingesting from source schemas happens in the DE lifecycle *transformation* stage**, discussed in Chapter 8
* **But DE’s need to begin studying source schemas *as soon they plan to ingest data from a new source***
* **Communication is critical for understanding source data, + DE’s also have the opportunity to reverse the flow of communication and help SWE’s improve data where it is produced**

a) Detecting and Handling Upstream and Downstream Schema Changes.

* **Changes in schema frequently occur in source systems + are often well out of a DE’s control**
* Examples of schema changes include the following:
* Adding a new column
* Changing a column type
* Creating a new table
* Renaming a column
* It’s becoming **increasingly common for ingestion tools to automate the detection of schema changes + even auto-update target tables**
* Ultimately, this is something of a **mixed blessing**
* **Schema changes can still break pipelines downstream of staging + ingestion**
* **DE’s must still implement strategies to respond to changes automatically + alert on changes that cannot be accommodated automatically**
* Automation is excellent, but **analysts + data scientists who rely on this data should be informed of the schema changes that violate existing assumptions**
* **Even if automation can accommodate a change, the new schema may adversely affect the performance of reports and models**
* **Communication between those making schema changes + those impacted by these changes is as important as reliable automation that checks for schema changes**

b) Schema Registries

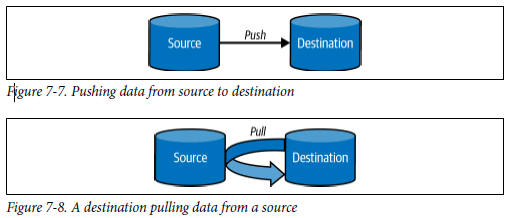
* **In streaming data, every message has a schema, + these schemas may evolve between producers + consumers**
* A **schema registry**is a **metadata repository used to maintain schema and data type integrity in the face of constantly changing schemas**
* They **can also track schema versions and history**
* They **describe the data model for messages, allowing consistent serialization + deserialization between producers + consumers**
* They used in most major data tools and clouds

###### v) Metadata

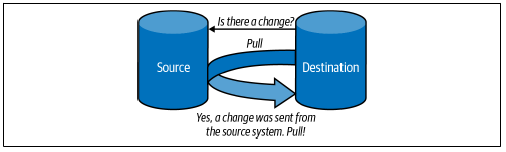
* In addition to the apparent characteristics just covered, **a payload often contains metadata** (Ch. 2)
* Metadata is **data about data, + can be as critical as the data itself**
* One of the significant limitations of the early approach to the data lake (or data swamp) was a complete lack of attention to metadata
* **Without a detailed description of the data, it may be of little value**

##### Push Vs. Pull Vs. Poll Patterns

* We mentioned **push vs. pull** when we introduced the DE lifecycle in Chapter 2
* A **push**strategy involves a **source system sending data to a target**, while a **pull**strategy entails **a target reading data directly from a source**
* As we mentioned in that discussion, the lines between these strategies are blurry

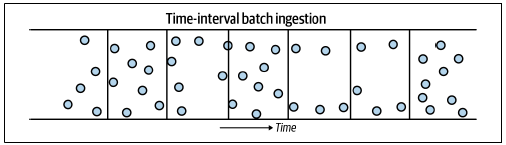


* Another pattern related to pulling is **polling**for data, which **involves periodically checking a data source for any changes**
* **When changes *are* detected, the destination pulls data as it would in a regular pull situation**

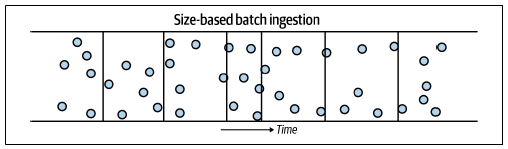


#### Batch Ingestion Considerations

* **Batch ingestion** (involves **processing data in bulk**) is often a **convenient** way to ingest data
* This means **data is ingested by taking a *subset* of data from a source system, based either on a time interval *or* the size** **of accumulated data**
* **Time-interval batch ingestion**is **widespread in traditional business ETL for DW’s**
* This *pattern is often used to process data once a day, overnight during off-hours, to provide daily reporting,* but other frequencies can also be used



* **Size-based batch ingestion** is quite **common when data is moved from a streaming-based system into object storage**



* Ultimately, you must cut the data into discrete blocks for future processing in a data lake
* Some size-based ingestion systems can break data into objects based on various criteria, such as the size in bytes of the total number of events
* Some commonly used batch ingestion patterns (discussed next), include the following:

##### Snapshot or Differential Extraction

* **DE’s must choose whether to capture *full* snapshots of a source system or *differential* (sometimes called “incremental”) updates**
* W/ **full snapshots**, DE’s **grab the *entire* current state of the source system on *each* update read**
* W/ the **differential update**pattern, DE’s **can pull only the updates + changes since the last read from the source system**
* While **differential updates are ideal for minimizing network traffic + target storage usage**, **full snapshot reads remain extremely common because of their simplicity**

##### File-Based Export and Ingestion

* **Data is quite often moved between databases and systems using files**
* **Data is serialized into files in an exchangeable format, + these are provided to an ingestion system**
* We **consider file-based export to be a push-basedingestion pattern**
* This is because **data export + preparation work is done on the source system side**.
* File-based ingestion has several potential advantages over a direct database connection approach
* It’s **often undesirable to allow direct access to backend systems for security reasons**
* W/ file-based ingestion, export processes are run on the data-source side, giving source system engineers complete control over what data gets exported + how the data is preprocessed
* Once files are done, they can be provided to the target system in various ways
* **Common file-exchange methods are object storage, secure file transfer protocol (SFTP), electronic data interchange (EDI), or secure copy (SCP)**

##### ETL Vs. ELT

* ETL and ELT = both extremely common ingestion, storage, + transformation patterns you’ll encounter in batch workloads
* Brief definitions of the extract and load parts of ETL and ELT:
* **Extract: getting data from a source system**
* **While extractseems to imply pullingdata, *it can also be push based***
* Extraction **may also require reading metadata and schema changes.**
* **Load:** Once **data** is extracted, it **can either be transformed** (ETL) **before loading** it into a storage destination **or simply loaded into storage for future transformation**
* When loading data, **be mindful of the type of system you’re loading, the schema of the data, and the performance impact of loading**
* We cover ETL and ELT in greater detail in Chapter 8

##### Inserts, Updates, and Batch Size

* **Batch-oriented systems often perform poorly when users attempt to perform many small-batch operations rather than a smaller number of large operations**
* **Ex: While common to insert one row at a time in a transactional database, this is a bad pattern for many columnar databases**, as it **forces the creation of many small, suboptimal files + forces the system to run a high number of create objectoperations**
* **Running many small *in-place* update operations is an even *bigger* problem b/c it causes the database to scan each existing column file to run the update**
* **Understand the appropriate update patterns for the database or data store you’re working w/**
* Also, **understand that certain technologies are purpose-built for high insert rates**
* Ex: Apache Druid and Apache Pinot can handle high insert rates
* SingleStore can manage hybrid workloads that combine OLAP + OLTP characteristics
* BigQuery performs *poorly* on a high rate of vanilla SQL single-row inserts *but* extremely well if data is fed in through its stream buffer
* **Know the limits and characteristics of your tools**

##### Data Migration

* **Migrating data to a new database or environment is not usually trivial, + data needs to be moved in bulk**
* Sometimes this means moving data sizes that are hundreds of TB or much larger, often involving the migration of specific tables + moving *entire databases and systems*
* **Data migrations probably aren’t a regular occurrence as a DE, but you be familiar with them**
* **As is often the case for data ingestion, schema management is a crucial consideration**
* Suppose you’re migrating data from one database system to a different one (say, SQL Server to Snowflake)
* **No matter how closely the two databases resemble each other, subtle differences almost always exist in the way they handle schema**
* **Fortunately, it is generally easy to test ingestion of a sample of data + find schema issues before undertaking a complete table migration**.
* **Most data systems perform best when data is moved in bulk rather than as individual rows or events**
* **File or object storage is often an excellent intermediate stage for transferring data**
* Also, **one of the biggest challenges of database migration is not the movement of the data itself but the movement of *data pipeline connections* from the old system to the new one**
* Be aware that **many tools are available to automate various types of data migrations**
* Especially for large + complex migrations, look at these options before doing this manually or writing your own migration solution

#### Message and Stream Ingestion Considerations

* **Ingesting event data is common**, + this section covers issues to consider when ingesting events

##### 1) Schema Evolution

* **Schema evolution is common when handling event data, as fields may be added or removed, or value types might change** (say, a string to an integer)
* Schema evolution **can have unintended impacts on your data pipelines and destinations**
* Ex: An IoT device gets a firmware update that adds a new field to the event it transmits, or a 3rd-party API introduces changes to its event payload, or countless other scenarios
* **All of these potentially impact your downstream capabilities**
* Here are a few suggestions to alleviate issues related to schema evolution
* **1) If your event-processing framework has a schema registry (discussed earlier), use it to version your schema changes**
* **2) A dead-letter queue (described later) can help you investigate issues w/ events that are not properly handled**
* 3) The low-fidelity route (and the most effective) is **regularly communicating w/ upstream stakeholders about potential schema changes + *proactively* addressing schema changes w/ teams introducing these changes instead of reacting to the receiving end of breaking change**

##### 2) Late-Arriving Data

* Though you probably prefer all event data to arrive on time, event data might arrive late
* A group of events might occur around the same time frame (similar event times), but some might arrive later than others (late ingestion times) because of various circumstances
* Ex: An IoT device might be late sending a message because of internet latency issues, which is common when ingesting data
* **Be aware of late-arriving data + the impact on downstream systems and uses**
* Suppose you assume that ingestion or process time is the same as the event time
* You **may get some strange results if your reports or analysis depend on an accurate portrayal of when events occur**
* **To handle late-arriving data, you need to set a cutoff time for when late-arriving data will no longer be processed**

##### 3) Ordering and Multiple Delivery

* **Streaming platforms are generally built out of distributed systems, which can cause some complications**
* Specifically, messages may be delivered out of order and more than once (at-least-once delivery)
* See the event-streaming platforms discussion in Chapter 5 for more details

##### 4) Replay

* **Replay****allows readers to request a range of messages from the history, allowing you to rewind your event history to a particular point in time**
* Replay is a **key capability in many streaming ingestion platforms + is particularly useful when you need to re-ingest + reprocess data for a specific time range**
* Ex: RabbitMQ typically deletes messages after all subscribers consume them
* Kafka, Kinesis, + Pub/Sub all support event retention and replay

##### 5) Time to Live

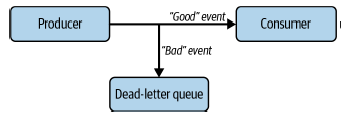
* *How long will you preserve your event record?*
* A **key parameter is maximum message retention time, also known as the time to live (TTL)**
* **TTL is usually a configuration you’ll set for how long you want events to live before they are acknowledged + ingested**
* **Any *unacknowledged* event that’s *not ingested* after its TTL expires automatically disappears**
* This is **helpful to reduce backpressure and unnecessary event volume in your event-ingestion pipeline**
* **Find the right balance of TTL impact on our data pipeline**
* An extremely short TTL (milliseconds or seconds) might cause most messages to disappear before processing
* A very long TTL (several weeks or months) will create a backlog of many unprocessed messages, resulting in long wait times
* Let’s look at how some popular platforms handle TTL as of 2022-2023.
* Google Cloud Pub/Sub supports retention periods of up to 7 days
* Amazon Kinesis Data Streams retention can be turned up to 365 days
* Kafka can be configured for indefinite retention, limited by available disk space
* Kafka also supports the option to write older messages to cloud object storage, unlocking virtually unlimited storage space + retention

##### Message Size

* **Message size is an easily overlooked issue**
* **Must ensure that the streaming framework in question can handle the *maximum expected* message size**
* Amazon Kinesis supports a maximum message size of 1 MB
* Kafka defaults to this maximum size but can be configured for a maximum of 20 MB or more
* *Configurability may vary on managed service platforms*

##### Error Handling and Dead-Letter Queries

* **Sometimes events aren’t successfully ingested**
* Perhaps an event is sent to a nonexistent topic or message queue, the message size may be too large, or the event has expired past its TTL
* **Events that cannot be ingested need to be rerouted + stored in a separate location called a dead-letter queue**, which **segregates problematic events from events that can be accepted by the consumer**



* **If events are *not* rerouted to a dead-letter queue, these erroneous events risk blocking other messages from being ingested**
* DE’s can use a dead-letter queue to diagnose why event ingestions errors occur + solve data pipeline problems, + might be able to reprocess some messages in the queue after fixing the underlying cause of errors

##### Consumer Pull and Push

* **A consumer subscribing to a topic can get events in 2 ways: push and pull**
* Let’s look at the ways some streaming technologies pull and push data
* Kafka + Kinesis support only **pull subscriptions, where subscribers read messages from a topic + confirm when they have been processed**
* In addition to pull subscriptions, Pub/Sub and RabbitMQ support **push subscriptions, allowing these services to write messages to a listener**
* **Pull subscriptions are the default choice for most DE applications**, but you **may want to consider push capabilities for specialized applications**
* ***NOTE: Pull-only message ingestion systems can still push if you add an extra layer to handle this***

##### 9) Location

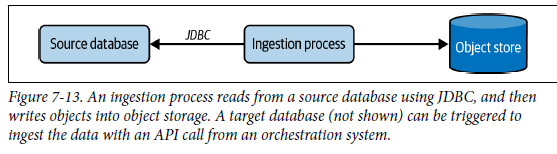
* It is **often desirable to integrate streaming across several locations for enhanced redundancy + to consume data close to where it is generated**
* **As a general rule, the closer your ingestion is to where data originates, the better your bandwidth and latency**
* However, you **need to balance this against the costs of moving data between regions to run analytics on a combined dataset**
* As always, **data egress costs can spiral quickly**
* **Do a careful evaluation of the trade-offs as you build out your architecture**

#### Way to Ingest Data

* Now that we’ve described some of the significant patterns underlying batch + streaming ingestion, let’s focus on *ways* you can ingest data
* Although we will cite some common ways, **keep in mind that the universe of data ingestion practices and technologies is vast and growing daily**

##### Direct Database Connection

* **Data can be pulled from databases for ingestion by querying + reading over a network connection**
* **Most commonly**, this connection is made using **ODBC** or **JDBC**.
* **ODBC** **uses a driver hosted by a client accessing the database to translate commands issued to the standard ODBC API into commands issued to the database**
* The database returns query results over the wire, where the driver receives them + translates them back into a standard form to be read by the client
* For ingestion, the application utilizing the ODBC driver is an ingestion tool
* The ingestion tool may pull data through many small queries or a single large query
* **JDBC** is conceptually **remarkably similar to ODBC**
* **A Java driver connects to a remote database + serves as a translation layer between the standard JDBC API + the native network interface of the target database**
* It might seem strange to have a database API dedicated to a single programming language, but there are strong motivations for this
* The **Java Virtual Machine (JVM)** is **standard, portable across hardware architectures + OS’s, + provides the performance of compiled code through a just-in-time (JIT) compiler**
* The **JVM is an extremely popular compiling VM for running code in a portable manner**
* JDBC provides **extraordinary database driver portability**
* ODBC drivers are shipped as OS + architecture native binaries, so database vendors must maintain versions for *each* architecture/OS version that they wish to support
* On the other hand, vendors can ship a single JDBC driver that is compatible with *any* JVM language (e.g., Java, Scala, Clojure, or Kotlin) and JVM data framework (i.e., Spark.)
* **JDBC has become so popular that it is also used as an interface for non-JVM languages such as Python**
* The Python ecosystem provides translation tools that allow Python code to talk to a JDBC driver running on a local JVM
* **JDBC + ODBC are used extensively for data ingestion from *relational* databases**, returning to the general concept of direct database connections
* **Various enhancements are used to accelerate data ingestion**
* Many data frameworks can **parallelize several simultaneous connections + partition queries to pull data in parallel**
* On the other hand, **nothing is free, + using parallel connections also increases the load on the source database**
* JDBC and ODBC were long the gold standards for data ingestion from databases, but **these connection standards are beginning to show their age for many DE applications**
* These connection standards **struggle with nested data**, + they **send data as rows, which means that native nested data must be reencoded as string data to be sent over the wire, + columns from columnar databases must be reserialized as rows**
* As discussed in earlier, **many databases now support native file export that bypasses JDBC/ODBC and exports data directly in formats such as Parquet, ORC, + Avro**
* Alternatively, **many cloud DW’s provide direct REST APIs**
* **JDBC connections should generally be integrated w/ other ingestion technologies**
* Ex: We commonly use a reader process to connect to a database with JDBC, write the extracted data into multiple objects, + then orchestrate ingestion into a downstream system
* The reader process can run in a wholly ephemeral cloud instance or in an orchestration system



##### Change Data Capture

* **Change data capture (CDC)** is the **process of ingesting changes from a source database system**
* Ex: We might have a source PostgreSQL system that supports an application + periodically or continuously ingests table changes for analytics
* Note that the discussion here is by no means exhaustive + this introduces common patterns, but **be sure to** **read the documentation on a particular database to handle the details of CDC strategies**

###### a) Batch-Oriented CDC

* If the database table in question has an ***updated\_at* field** containing the last time a record was written or updated, we **can query the table to find all updated rows since a specified time**
* We set the filter timestamp based on when we last captured changed rows from the tables
* This process allows us to pull changes + differentially update a target table.
* **This form of batch-oriented CDC has a key limitation**: while we **can easily determine which rows have changed since a point in time, we don’t necessarily obtain *all* changes that were applied to these rows**
* Consider the example of running batch CDC on a bank account table every 24 hours
* This *operational* table shows the current account balance for each account
* When money is moved in and out of accounts, the banking application runs a transaction to update the balance
* When we run a query to return all rows in the account table that changed in the last 24 hours, we’ll see records for each account that recorded a transaction
* But suppose one customer withdrew money 5 times using a debit card in the last 24 hours
* Our query will return *only* the *last* account balance recorded in the 24 hour period, + other records over the period won’t appear
* This issue can be mitigated by utilizing an **insert-only schema**, where each account transaction is recorded as a new record in the table (discussed earlier)

###### b) Continuous CDC

* **Continuous CDC captures *all* table history + can support *near real-time* data ingestion, either for real-time database replication or to feed real-time streaming analytics**
* Rather than running periodic queries to get a batch of table changes, **continuous CDC treats *each write* to the database as an event**
* We **can capture an event stream for continuous CDC in a couple of ways**
* One of the most common approaches w/ a transactional database such as PostgreSQL is **log-based CDC**
* The database binary log records every change to the database sequentially
* A CDC tool can read this log + send the events to a target, such as the Apache Kafka Debezium streaming platform
* **Some databases support a simplified, *managed* CDC paradigm**
* For instance, many cloud-hosted databases can be configured to directly trigger a serverless function or write to an event stream every time a change happens in the database
* This completely frees DE’s from worrying about the details of how events are captured in the database + forwarded

###### c) CDC and Database Replication

* **CDC can be used to replicate between databases: events are buffered into a stream + *asynchronously* written into a *second* database**
* ***However*, many databases natively support a *tightly coupled* version of replication (synchronous replication) that keeps the replica fully in sync w/ the primary database**
* Synchronous replication **typically requires that the primary database and the replica are of the same type** (e.g., PostgreSQL to PostgreSQL)
* **Advantage of synchronous replication = the secondary database can offload work from the primary database by acting as a read replica (read queries can be redirected to the replica)**
* The **query will return the same results that would be returned from the primary database**
* **Read replicas are often used in batch data ingestion patterns to allow large scans to run w/out overloading the primary production database**
* In addition, **an application can be configured to fail over to the replica if the primary database becomes unavailable**
* **No data will be lost in the failover because the replica is entirely in sync w/ the primary database**
* The **advantage of *asynchronous* CDC** **replication** is **a loosely coupled architecture pattern**
* While the **replica might be slightly delayed from the primary database**, this is **often not a problem for analytics applications, + events can now be directed to a variety of targets**
* We might run CDC replication while simultaneously directing events to object storage + a streaming analytics processor

###### d) CDC Considerations

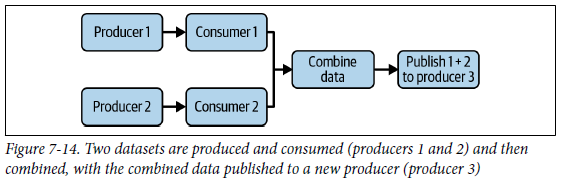
* **Like anything in tech, CDC is *NOT* free**
* CDC **consumes various database resources, such as memory, disk bandwidth, storage, CPU time, + network bandwidth**
* **DE’s should work w/ production teams + run tests before turning on CDC on production systems to avoid operational problems**
* *Similar considerations apply to synchronous replication*
* For **batch CDC**, be aware that **running *any* large batch query against a transactional production system can cause excessive load**
* **Either run such queries only at off-hours or use a read replica to avoid burdening the primary database.**

##### APIs

* As mentioned in Chapter 5, **APIs are a data source that continues to grow in importance + popularity**
* A typical organization may have hundreds of external data sources such as SaaS platforms or partner companies
* **The hard reality is that no proper standard exists for data exchange over APIs**
* DE’s can spend a significant amount of time reading documentation, communicating w/ external data owners, and writing + maintaining API connection code
* 3 trends are slowly changing this situation
* **1) Many vendors provide API client libraries for various programming languages that remove much of the complexity of API access**
* **2) Numerous data connector platforms are available now as SaaS, open source, or managed open source**
* These **platforms provide turnkey data connectivity to many data sources + offer frameworks for writing custom connectors for unsupported data source**s (we talk about **Managed Data Connectors** soon)
* **3) The emergence of** **data sharing** (Chapter 5), or the **ability to exchange data through a standard platform** such as BigQuery, Snowflake, Redshift, or S3
* Once data lands on one of these platforms, it is straightforward to store it, process it, or move it somewhere else
* **Data sharing has had a large + rapid impact in the DE space**
* ***Don’t reinvent the wheel when data sharing is not an option + direct API access is necessary***
* **While a managed service might look like an expensive option, consider the value of your time + the opportunity cost of building API connectors when you could be spending your time on higher-value work**
* In addition, **many managed services now support building custom API connectors**
* This may provide API technical specifications in a standard format or writing connector code that runs in a serverless function framework (e.g., AWS Lambda) while letting the managed service handle the details of scheduling + synchronization
* Again, **these services can be a huge time-saver for DE’s, both for development + ongoing maintenance.**
* **Reserve custom connection work for APIs that aren’t well supported by existing frameworks** (you will find that there are **still plenty of these to work on**)
* **Handling custom API connections has 2 main aspects: software development + DevOps**
* **Follow SWE best practices** (use version control, continuous delivery (CI/CD), + automated testing, etc.)
* **In addition to following DevOps best practices, consider an orchestration framework**, which **can dramatically streamline the operational burden of data ingestion**

##### Message Queues and Event-Streaming Platforms

* **Message queues + event-streaming platforms are widespread ways to ingest real-time data from web and mobile applications, IoT sensors, and smart devices**
* **As real-time data becomes more ubiquitous, you’ll often find yourself either introducing or retrofitting ways to handle real-time data in your ingestion workflows**
* As such, it’s **essential to know how to ingest real-time data**
* **Popular real-time data ingestion includes message queues or event-streaming platforms** (Chapter 5)
* Though these are **both *source systems*, they *also* act as ways to ingest data**
* In both cases, you **consume events from the publisher you subscribe to**
* Recall the **differences between messages and streams**
* **A messageis handled at the *individual event level* and is meant to be *transient***
* **Once a message is consumed, it is acknowledged + removed from the queue**
* On the other hand, a **stream****ingests events into an *ordered log***
* The **log persists for as long as you wish**, **allowing events to be queried over various ranges, aggregated, + combined w/ other streams to create new transformations published to downstream consumers**
* Below we have 2 producers sending events to 2 consumers
* These events are combined into a new dataset + sent to a 3rd producer for downstream consumption



* The last point is an **essential difference between batch and streaming ingestion**.
* **Whereas batch usually involves static workflows (ingest data, store it, transform it, + serve it), messages and streams are fluid**
* **Ingestion can be nonlinear, w/ data being published, consumed, + also can be *re*-published + *re*-consumed**
* **When designing real-time ingestion workflows, keep in mind *how* data will flow**
* **Another consideration** is the **throughput of your real-time data pipelines**
* **Messages + events should flow w/ as little latency as possible, meaning you should provision adequate partition (or shard) bandwidth + throughput**
* **Provide sufficient memory, disk, + CPU resources for event processing, + if managing your real-time pipelines, incorporate autoscaling to handle spikes + save money as load decreases**
* For these reasons, **managing a streaming platform can entail significant overhead**
* **Consider *managed services* for your real-time ingestion pipelines, + focus your attention on ways to get value from your real-time data**

##### Managed Data Connectors

* **These days, if considering writing a data ingestion connector to a database or API, ask yourself: *has this already been created?***
* Furthermore, ***is there a service that will manage the nitty-gritty details of this connection for me?***
* We mentioned the popularity of **managed data connector platforms + frameworks**.
* These tools **aim to provide a standard set of connectors available out of the box to spare DE’s building complicated plumbing to connect to a particular source**
* Instead of creating + managing a data connector, you outsource this service to a 3rd party
* Generally, **options in this specific space allow users to set a target and source, ingest in various ways (e.g., CDC, replication, truncate + reload), set permissions + credentials, configure an update frequency, + begin syncing data**
* The vendor or cloud behind the scenes fully manages + monitors data syncs
* If data synchronization fails, you’ll receive an alert w/ logged information on the cause of the error
* **Try to use managed connector platforms instead of creating + managing your connectors**
* Vendors + OSS projects each typically have hundreds of prebuilt connector options + can easily create custom connectors
* The **creation + management of data connectors is largely undifferentiated heavy lifting these days + should be outsourced whenever possible**

##### Moving Data with Object Storage

* **Object storage** is a **multitenant system in public clouds, + it supports storing massive amounts of data**
* This makes **object storage ideal for moving data in + out of data lakes, between teams, and transferring data between organizations**
* Can even provide short-term access to an object with a signed URL, giving a user temporary permission
* **Object storage is one of the most optimal + secure way to handle file exchange**
* Public cloud storage implements the latest security standards, has a robust track record of scalability + reliability, accepts files of arbitrary types + sizes, + provides high-performance data movement
* See more in Chapter 6

##### EDI

* Another practical reality for DE’s is **electronic data interchange (EDI)**, a term vague enough to **refer to any data movement method**
* It usually refers to somewhat archaic means of file exchange, such as by email or flash drive
* DE’s will find that **some data sources do not support more modern means of data transport, often because of archaic IT systems or human process limitations**
* DE’s can at least **enhance EDI through automation**
* Ex: Can set up a cloud-based email server that saves files onto company object storage as soon as they are received
* This can trigger orchestration processes to ingest + process data
* This is much more robust than an employee downloading the attached file + manually uploading it to an internal system, which is still frequently sees

##### Databases and File Export

* DE’s should **be aware of how the source database systems handle file *export***
* **Export involves large data scans that significantly load the database for many transactional systems**
* Source system engineers must assess when these scans can be run w/out affecting application performance + might opt for a strategy to mitigate the load
* **Export queries can be broken into smaller exports by querying over key ranges or via one partition at a time**
* Alternatively, a **read replica can reduce load**
* Read replicas are **especially appropriate if exports happen many times a day + coincide w/ a high source system load**
* **Major cloud DW’s are highly optimized for direct file export**
* Ex: Snowflake, BigQuery, Redshift, and others support **direct export to object storage in various formats**

##### Practical Issues with Common File Formats

* DE’s should also **be aware of the file formats to export**
* **CSV is still ubiquitous + *highly error prone*** as of 2022-2023
* Namely, CSV’s default delimiter is also one of the most familiar characters in the English language, the comma
* But it gets worse, since **CSV is by no means a uniform format**
* **DE’s must stipulate the delimiter, quote characters, + escaping** to appropriately handle the export of string data
* **CSV also doesn’t natively encode schema information or directly support nested structures**
* **CSV file encoding + schema information must be configured *in the target system* to ensure appropriate ingestion**
* **Autodetection is a convenience feature provided in many cloud environments but *is inappropriate for production ingestion***
* As a **best practice**, DE’s should **record CSV encoding + schema details in file metadata**
* **More robust + expressive export formats** include **Parquet, Avro, Arrow, + ORC or JSON**
* These formats **natively encode schema information + handle arbitrary string data w/ no particular intervention**
* Many of them **also handle nested data structures natively** so that JSON fields are stored using internal nested structures rather than simple strings
* ***For columnar databases*, columnar formats** (Parquet, Arrow, ORC) **allow more efficient data export because columns can be directly transcoded between formats**
* These formats are also **generally more optimized for query engines**
* The Arrow file format is designed to map data directly into processing engine memory, providing high performance in data lake environments
* The ***disadvantage* of these newer formats** is that **many of them are not natively supported by source systems**
* **DE’s are often forced to work w/ CSV data + then build robust exception handling + error detection to ensure data quality on ingestion**
* See Appendix A for a more extensive discussion of file formats

##### Shell

* The **shell**is an **interface by which you may execute commands to ingest data**
* It **can be used to script workflows for virtually any software tool**, + **shell scripting is still used extensively in ingestion processes**
* A shell script **might read data from a database, reserialize it into a different file format, upload it to object storage, + trigger an ingestion process in a target database**
* While storing data on a single instance/server is *not* highly scalable, many data sources are not particularly large, + such approaches work just fine
* In addition, **cloud vendors generally provide robust CLI-based tools**
* Ex: It is possible to run complex ingestion processes simply by issuing commands to the AWS CLI
* **As ingestion processes grow more complicated + the SLA grows more stringent, DE’s should consider moving to a proper orchestration system**

##### SSH

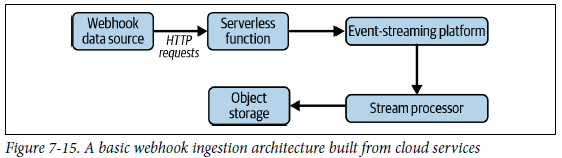
* **Secure shell (SSH)**is *not* an ingestion strategy but **a protocol used with other ingestion strategies**
* We use SSH in a few ways
* **1) SSH can be used for file transfer with SCP, as mentioned earlier**
* **2) SSH tunnels are used to allow secure, isolated connections to databases**
* **Application databases should *NEVER* be directly exposed on the internet**
* Instead, **DE’s can set up a bastion host (an intermediate host instance that can connect to the database in question)**
* This **host machine *IS* exposed on the internet, although locked down for minimal access from only specified IP addresses to specified ports**
* To connect to the database, a remote machine first opens an SSH tunnel connection to the bastion host + *then* connects from the host machine to the database

##### SFTP and SCP

* **Accessing + sending data both from secure FTP (SFTP) and secure copy (SCP) are techniques you should be familiar with, even if DE’s do not typically use these regularly (*IT or security/SecOps will handle this*)**
* DE’s rightfully cringe at the mention of **SFTP** (occasionally, you even hear instances of FTP being used in *production*)
* Regardless, **SFTP is still a practical reality for many businesses**
* They work w/ partner businesses that consume or provide data using SFTP + are unwilling to rely on other standards
* **To avoid data leaks, security analysis is critical in these situations**
* **SCP** is a **file-exchange protocol that runs over an SSH connection**
* It **can be a secure file-transfer option *if it is configured correctly***
* Again, **adding additional network access control (defense in depth) to enhance SCP security is highly recommended**

##### Webhooks

* **Webhooks** are **often referred to as reverse APIs**
* **For a *typical* REST data API, the data provider gives engineers API specifications that they use to write data ingestion code**
* The **code makes requests + receives data in responses**
* **With a *webhook*, a data provider again defines an API request specification, but the data *provider* is who *makes* API calls, rather than receiving them, + it’s the *data consumer’s* responsibility to provide an API endpoint for the provider to call**
* The ***consumer* is responsible for ingesting each request + handling data aggregation, storage, + processing**



* **Webhook-based data ingestion architectures can be brittle, difficult to maintain, + inefficient**
* ***Using appropriate off-the-shelf tools,* DE’s can build more robust webhook architectures w/ lower maintenance + infrastructure costs**
* Ex: An AWS webhook pattern might use a serverless function framework (Lambda) to receive incoming events, managed event-streaming platform to store + buffer messages (Kinesis), stream-processing framework to handle real-time analytics (Flink), + an object store for long-term storage (S3)
* ***Notice this architecture does much more than simply ingest the data***
* This **underscores ingestion’s entanglement w/ the other stages of the DE lifecycle**
* It is **often impossible to define your ingestion architecture w/out making decisions about storage + processing**

##### Web Interface

* **Web interfaces** for data access **remain a practical reality for DE’s**
* We frequently run into situations where not all data + functionality in a SaaS platform is exposed through automated interfaces such as APIs + file drops
* Instead, someone must manually access a web interface, generate a report, + download a file to a local machine
* This has obvious drawbacks, such as people forgetting to run the report or a laptop dying
* **Where possible, choose tools + workflows that allow for *automated* access to data**

##### Web Scraping

* **Web scraping****automatically extracts data from web pages, often by combing the web page’s various HTML elements**
* Might scrape ecommerce sites to extract product pricing information or scrape multiple news sites for a news aggregator
* **Web scraping is widespread**, and you may encounter it as a DE, but It’s **also a murky area where ethical + legal lines are blurry**
* Here is some top-level advice to be aware of before undertaking any web-scraping project
* **1) Ask yourself if you *should* be web scraping or if data is available from a 3rd party**
* If your decision is to web scrape, be a good citizen
* **Don’t inadvertently create a denial-of-service (DoS) attack, + don’t get your IP address blocked**
* **Understand how much traffic you generate + pace your web-crawling activities appropriately**
* Just because you can spin up thousands of simultaneous Lambda functions to scrape doesn’t mean you *should*, since excessive web scraping could lead to disabling of an AWS account
* **2) Be aware of the legal implications of your activities**
* Again, generating DoS attacks can entail legal consequences
* Actions that violate terms of service may cause headaches for an employer or you personally
* **3) Web pages constantly change their HTML element structure, making it tricky to keep your web scraper updated**
* *Ask yourself, is the headache of maintaining these systems worth the effort?*
* **Web scraping has interesting implications for the DE lifecycle *processing* stage**, so DE’s should **think about various factors at the beginning of a web scraping project**
* What do you intend to do with the data?
* Are you just pulling required fields from the scraped HTML by using Python code + then writing these values to a database?
* Do you intend to maintain the complete HTML code of the scraped websites + process this data using a framework like Spark?
* **These decisions may lead to very different architectures downstream of ingestion**

##### Transfer Appliances for Data Migration

* **For massive quantities of data (100 TB or more), transferring data directly over the internet may be a slow and costly process**
* ***At this scale*, the fastest, most efficient way to move data is not over the wire but by *truck***
* **Cloud vendors offer the ability to send data via a physical “box of hard drives”**
* Simply **order a storage device** (called a **transfer appliance**), **load your data from your servers, and then send it back to the cloud vendor, which will upload your data**
* The suggestion is to **consider using a transfer appliance if your data size hovers ~100 TB**
* On the extreme end, AWS even offers Snowmobile, a transfer appliance sent to you in a semitrailer!
* Snowmobile is intended to lift + shift an *entire* data center, in which data sizes are in the PB range or greater
* **Transfer appliances are handy for creating hybrid-cloud or multi-cloud setups**
* Ex: Amazon’s data transfer appliance (AWS Snowball) supports import + export
* To migrate into a 2nd cloud, users can export their data into a Snowball device + then import it into a 2nd transfer appliance to move data into GCP or Azure
* This might sound awkward, but **even when it’s feasible to push data over the internet between clouds, data egress fees make this a costly proposition**
* ***Physical* transfer appliances = a cheaper alternative when data volumes are significant**
* Remember that **transfer appliances + data migration services are *one-time* data ingestion events and are NOT suggested for ongoing workloads**
* Suppose you have workloads requiring constant data movement in either a hybrid or multi-cloud scenario
* In that case, your data sizes are presumably batching or streaming much smaller data sizes on an ongoing basis

##### Data Sharing

* **Data sharingis growing as a popular option for consuming data** (see Chapters 5 and 6)
* **Data providers will offer datasets to 3rd-party subscribers, either for free or at a cost**
* These datasets are **often shared in a read-only fashion**, meaning you **can integrate these datasets w/ your own data (+ other 3rd-party datasets), but you do not *own* the shared dataset**
* In the strict sense, ***this isn’t ingestion***, where you get physical possession of the dataset
* **If the data provider decides to remove your access to a dataset, you’ll no longer have access to it**
* Many cloud platforms offer data sharing, allowing you to share your data + consume data from various providers
* Some of these platforms also provide **data marketplaces** where companies + organizations can offer their data for sale

#### Whom You’ll Work With

* Data ingestion sits at several organizational boundaries
* **In developing + managing data ingestion pipelines, DE’s will work with both people + systems sitting upstream (data producers) and downstream (data consumers)**

##### Upstream Stakeholders

* **A significant disconnect often exists between those responsible for *generating data* (typically SWE’s) + the DE’s who will prepare this data for analytics and data science**
* SWE’s + DE’s usually sit in separate organizational silos
* If SWE’s think about DE’s, they **typically see DE’s simply as downstream consumers of the data exhaust from their application, *NOT* as stakeholders**
* This current state of affairs is a problem *and* a significant opportunity
* **DE’s can improve the quality of their data by inviting SWE’s to be stakeholders in DE outcomes**
* The **vast majority of SWE’s** are well aware of the value of analytics + data science but **don’t necessarily have aligned incentives to contribute to DE efforts *directly***
* Simply **improving communication is a significant first step**
* **Often SWE’s have already identified potentially valuable data for downstream consumption**
* **Opening a communication channel encourages SWE’s to get data into shape for consumers + to communicate about data changes to prevent pipeline regressions**
* Beyond communication, **DE’s can highlight the contributions of SWE’s to team members, executives, + especially *product* managers**
* **Involving product managers in the outcome + treating downstream data processed as part of a product encourages them to allocate scarce SWE to collaboration with DE’s**
* ***Ideally*, SWE’s can work partially as *extensions* of the DE team, which allows them to collaborate on various projects, such as creating an event-driven architecture to enable real-time analytics**

##### Downstream Stakeholders

* Who is the ultimate *customer* for data ingestion?
* **DE’s focus on data practitioners and technology leaders such as data scientists, analysts, + CTOs**.
* They **would do well also to remember their broader circle of business stakeholders such as marketing directors, VPs over the supply chain, + CEOs**
* Too often, we see DEs pursuing sophisticated projects (e.g., real-time streaming buses or complex data systems) while digital marketing managers next door are left downloading Google Ads reports manually
* **View DE as a business, + recognize who your customers are**
* **Often basic automation of ingestion processes has significant value, especially for departments like marketing that control massive budgets + sit at the heart of revenue for the business**
* **Basic ingestion work may seem tedious, but delivering value to these core parts of the company will open up more budget + more exciting long-term DE opportunities**
* **DE’s can also invite more executive participation in this collaborative process**
* For a good reason, data-driven culture is quite fashionable in business leadership circles
* Still, **it is up to DEs + other data practitioners to provide executives w/ guidance on the best structure for a data-driven business.**
* This means **communicating the value of lowering barriers between data producers + DEs while supporting executives in breaking down silos + setting up incentives to lead to a more unified data-driven culture**
* Once again, **communication**is the watchword 🡪 **Honest communication early + often w/ stakeholders will go a long way to ensure that your data ingestion adds value**

#### Undercurrents

* Virtually ***ALL* the undercurrents touch the ingestion phase**, but we emphasize the most salient ones

##### Security

* **Moving data introduces security vulnerabilities because you have to transfer data between locations, + the *last* thing you want is to capture or compromise the data while moving**
* **Consider where the data *lives* and where it is *going***
* **Data that needs to move w/in your virtual private cloud (VPC) should use secure endpoints and *never* leave the confines of the VPC.**
* **Use a VPN or a dedicated private connection if you need to send data between the cloud + an on-prem network**
* This might cost money, but the **security is a good investment**
* **If your data traverses the public internet, ensure that the transmission is encrypted**
* It is ***always* a good practice to encrypt data over the** **wire**

##### Data Management

* Naturally, **data management *begins* at data ingestion,** as **it’s the starting point for lineage + data cataloging**
* So, from this point on, **DE’s need to think about schema changes, and ethics, privacy, + compliance**

###### a) Schema changes

* **Schema changes (such as adding, changing, or removing columns in a database table) remain (from some perspectives), an unsettled issue in data management**
* **Traditional approach = a careful command-and-control review process**
* Working w/ clients at large enterprises, you can potentially be quoted lead times of 6 months for the addition of a single field
* *This is* ***an unacceptable impediment to agility***
* On the **opposite end** of the spectrum, **any schema change in the source triggers target tables to be re-created with the new schema**
* This **solves schema problems at the ingestion stage *but can still break downstream pipelines and destination storage systems***
* **One possible solution is an approach pioneered by Git version control**
* When Linus Torvalds was developing Git, many of his choices were inspired by the limitations of **Concurrent Versions System (CVS)**
* **CVS is completely centralized (it supports only one current official version of the code, stored on a central project server)**
* To **make Git a *truly* distributed system**, Torvalds used the notion of a **tree, where each developer could maintain their processed branch of the code + then merge *to* or *from* other branches**
* *A few years prior to 2022-2023, such an approach to data was unthinkable*
* ***On-prem* MPP systems are typically operated at close to maximum storage capacity**
* However, **storage is cheap in big data and cloud DW environments**
* **One may quite easily maintain *multiple* versions of a table w/ different schemas and even different upstream transformations**
* Teams can support various “development” versions of a table by using orchestration tools such as Airflow
* **Schema changes, upstream transformation, + code changes can appear in development tables before official changes to the *main* table**

###### b) Data Ethics, Privacy, and Compliance

* Clients often ask for advice on encrypting sensitive data in databases, which generally leads DE’s to ask a fundamental question: **do you *need* the sensitive data you’re trying to encrypt?**
* As it turns out, *this question often gets overlooked when creating requirements + solving problems*
* **DE’s should always train themselves to ask this question when setting up ingestion pipelines**
* They **will inevitably encounter sensitive data**, + the natural tendency is to ingest it + forward it to the next step in the pipeline
* **But if this data is not needed, why collect it at all?**
* Why not simply drop sensitive fields before data is stored?
* *Data cannot leak if it is never collected*
* **Where it is *truly* necessary to keep track of sensitive identities, it is common practice to apply tokenization to anonymize identities in model training + analytics**
* But **DE’s should look at *where* this tokenization is used**
* **If possible, hash data at *ingestion* time**
* **DE’s cannot avoid working w/ highly sensitive data in some cases**
* Some analytics systems *must* present identifiable, sensitive information
* **DE’s must act under the highest ethical standards whenever they handle sensitive data**
* In addition, they **can put in place a variety of practices to reduce the *direct* handling of sensitive data**
* **Aim as much as possible for touchless productionwhere sensitive data is involved**
* This means **DE’s develop + test code on *simulated* or *cleansed* data in development + staging environments but *automated* code deployments to production**
* **Touchless production is an ideal that DE’s *should* strive for, but situations inevitably arise that *cannot* be fully solved in development and staging environments**
* Some bugs may not be reproducible w/out looking at the live data that is triggering a regression
* For these cases, put a **broken-glass process** in place: **require at least 2 people to approve access to sensitive data in the production environment**
* This access should be tightly scoped to a particular issue + come w/ an expiration date
* **One last bit of advice on sensitive data: be wary of naive *technological* solutions to *human* problems**
* Both encryption + tokenization are often treated like privacy magic bullets
* **Most cloud-based storage systems + nearly all databases encrypt data at rest *and* in motion by default**
* **Generally**, you **don’t see *encryption* problems but *data access* problems**
* Is the solution to **apply an extra layer of encryption to a single field** *OR* to***control access to that field?***
* After all, **one must still tightly manage access to the encryption** **key**
* Legitimate use cases exist for single-field encryption, but watch out for *ritualistic* encryption
* On the tokenization front, use common sense + assess data access scenarios
* If someone had the email of one of your customers, could they easily hash the email + find the customer in your data?
* **Thoughtlessly hashing data without salting + other strategies may not protect privacy as well as you think**

##### DataOps

* ***Reliable* data pipelines are the cornerstone of the DE lifecycle**
* **When they fail, all downstream dependencies come to a screeching halt**
* **DW’s + data lakes aren’t replenished w/ fresh data, and data scientists + analysts can’t effectively do their jobs, so the business is forced to fly blind**
* **Ensuring that your data pipelines are properly monitored is a crucial step toward reliability and effective incident response**
* **If there’s *one* stage in the DE lifecycle where monitoring is *critical*, it’s in the ingestion stage**
* **Weak or nonexistent monitoring means the pipelines may *or may not* be working**
* Referring back to our discussion on *time*, **be sure to *track* the various aspects of time (event creation, ingestion, process, + processing times)**
* Your **data pipelines should *predictably* process data in batches or streams**
* There are countless examples of reports + ML models generated from **stale data**
* In an extreme case, an ingestion pipeline failure may not be detected for 6 months, which is **very much avoidable through proper monitoring**
* ***What* should you monitor?**
* **Uptime**, **latency**, + **data volumes processed** are good places to start
* ***If an ingestion job fails, how will you respond?***
* In general, **build monitoring into your pipelines *from the beginning* rather than waiting for deployment**
* So, **monitoring is key, but *so* *is knowledge of the behavior of the upstream systems you depend on and how they generate data***
* **You should be aware of the number of events generated per time interval you’re concerned with (events/minute, events/second, + so on) *and* the average size of each event**
* Your **data pipeline should handle both the frequency + size of the events you’re ingesting**
* This need for knowledge **also applies to third-party services**
* W/ such services, **what you’ve gained in terms of lean operational efficiencies (reduced headcount) is replaced by systems you depend on being *outside of your control***
* **If using a third-party service (cloud, data integration service, etc.), how will you be alerted if there’s an outage? What’s your response plan if a service you depend on suddenly goes offline?**
* Sadly, **no universal response plan exists for third-party failures**
* **If you can fail over to other servers, preferably in another zone or region, definitely set this up**
* **If your data ingestion processes are built *internally*, do you have the proper testing + deployment automation to ensure the code functions in production?**
* And **if the code is buggy or fails, can you roll it back to a working version?**

###### Data-Quality Tests

* We often refer to **data as a silent killer**
* **If quality, valid data is the foundation of success in today’s businesses**
* **Using *bad* data to make decisions is much worse than having *no* data**
* Bad data has caused untold damage to businesses, + such data disasters are sometimes called “datastrophes”
* **Data is entropic (it often changes in unexpected ways without warning)**
* **One of the inherent differences between DevOps and DataOps is that we expect software regressions *only* when we *deploy* changes, while data often presents regressions independently *because of events outside our control***
* ***DevOps* engineers are typically able to detect problems by using binary conditions**
* *Has the request failure rate breached a certain threshold? How about response latency?*
* **In the *data* space, regressions often manifest as subtle statistical distortions**
* Is a change in search-term statistics a result of customer behavior? Or of a spike in bot traffic that has escaped the net? Or of a site test tool deployed in some other part of the company?
* Like system failures in DevOps, ***some* data regressions are immediately visible**
* Ex: The early 2000s, Google provided search terms to websites when users arrived from search
* In 2011, Google began withholding this information in some cases to protect user privacy better
* Analysts quickly saw “not provided” bubbling to the tops of their reports
* **The *truly dangerous* data regressions are *silent* + can come from inside *or* outside a business**
* Application developers may change the meaning of database fields w/out adequately communicating w/ data teams
* Changes to data from 3rd-party sources may go unnoticed
* **In the *best-case* scenario, reports break in obvious ways**
* **But often business metrics are distorted unbeknownst to decision makers**
* ***Whenever possible,* work w/ SWE’s to fix data-quality issues at the *source***
* It’s **surprising how many data-quality issues can be handled by respecting basic SWE best practices, such as logs to capture the history of data changes, checks (nulls, etc.), + exception handling (try, catch, etc.)**
* ***Traditional* data testing tools are generally built on simple binary logic**
* *Are nulls appearing in a non-nullable field? Are new, unexpected items showing up in a categorical column?*
* Meanwhile, ***statistical* data testing is a new realm, but one that is likely to grow dramatically in the next 5 years (as of 2022-2023)**

##### Orchestration

* Ingestion generally sits at the beginning of a large + complex data graph
* **And since ingestion is the 1st stage of the DE lifecycle, ingested data will flow into many more data processing steps, + data from many sources will commingle in complex ways**
* As we’ve emphasized so far, **orchestration is a crucial process for coordinating these steps**
* Organizations in an **early stage of data maturity** may choose to deploy ingestion processes as simple scheduled **CRON jobs**
* However, it is crucial to recognize that **this approach is brittle + can slow the velocity of DE deployment + development**
* **As data pipeline complexity grows, *true* orchestration is necessary**
* By “true” orchestration, we mean **a system capable of scheduling *complete task graphs* rather than individual tasks**
* An orchestration can start each ingestion task at the appropriate scheduled time
* **Downstream processing + transform steps begin as ingestion tasks are completed**
* **Further downstream, processing steps lead to additional processing steps**

##### Software Engineering

* **The ingestion stage of the DE lifecycle is engineering intensive**
* This stage **sits at the edge of the DE domain + often interfaces w/ external systems, where SWE’s + DE’s have to build a variety of custom plumbing**
* **Behind the scenes, ingestion is incredibly complicated, often w/ teams operating OSS frameworks** like Kafka or Pulsar (or some of the biggest tech companies running their own forked or homegrown ingestion solutions)
* But, as discussed earlier, **managed data connectors have simplified the ingestion process**, such as Fivetran, Matillion, and Airbyte
* **DE’s should take advantage of the *best available* tools (primarily, managed tools + services that do a lot of the heavy lifting for you) + develop high SWE competency *in areas where it matters***
* **Pays to use proper version control + code review processes and to implement appropriate tests for any ingestion-related code**
* **When writing software, your code needs to be decoupled**
* **Avoid writing monolithic systems w/ tight dependencies on the source or destination systems**

#### Conclusion

* **In your work as a DE ingestion will likely consume a significant part of your energy + effort**
* At the heart, ingestion is plumbing, connecting pipes to other pipes, ensuring that data flows consistently + securely to its destination
* At times, the minutiae of ingestion may feel tedious, but the exciting data applications (like analytics and ML) cannot happen without it
* As emphasized, we’re also in the midst of a sea change, moving from batch toward streaming data pipelines
* This is an opportunity for DE’s to discover interesting applications for streaming data, communicate these to the business, + deploy exciting new tech

#### Additional Resources

* Airbyte’s “Connections and Sync Modes” web page
* Chapter 6, “Batch Is a Special Case of Streaming,” in *Introduction to Apache Flink* by Ellen Friedman and Kostas Tzoumas (O’Reilly)
* “The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing” by Tyler Akidau et al.
* Google Cloud’s “Streaming Pipelines” web page
* Microsoft’s “Snapshot Window (Azure Stream Analytics)” documentation