Data Engineering and Data Management

17

by **Aditya Maheshwari**

In this chapter, we briefly explain some of the basic concepts that help data scientists go beyond theoretical/small scale projects (mostly used for experiments/local research/conceptual solutions) and introduce the concepts and frameworks that allow data scientists in conjunction with data teams to building data science products that process and deliver results at scale. We will discuss this in the context of exploring the role of data engineering in data projects and providing an overview of some of the types of data pipeline infrastructure commonly involved in these projects.

In the current data ecosystem, most data scientists are still not required to understand the inner workings of data engineering and data management; however, as modeling tools become increasingly automated, and as machine learning solutions move from conceptual to practical, most data project requirements become engineering focused.

We only provide a cursory look at the topic in this chapter; in-depth information is available at [1, 3, 7, 8, 5], while shorter overviews can be found at [6, 2]. Learners interested in database design should also consult [11].

17.1 Background and Context

In the 2010s, the field of data science gained prominence, with an emphasis on creating algorithms to decipher patterns from the vast data generated by digital platforms and technologies with continuous monitoring capabilities.

This marked a notable shift from traditional data analysis methods that primarily focused on smaller datasets in a scientific context. In classic statistical learning, the primary mode of data collection was through surveys, as detailed in Chapter 10, *Survey Sampling Methods*. It also included methods not directly connected to user activities, such as post-interaction surveys, evident with the post-visit survey of the *Canada Revenue Agency's My Account* service.[1]

1: Such approaches have their merits but can feel detached from real-time user actions.

These methodologies posed theoretical challenges, especially with handling modest sample sizes. However, the objective was clear: under a set of assumptions, can we determine any correlation between variables (features) and actions (outcomes)?

Historically, research often relied on fragmented or rudimentary systems. These were mostly adequate for routine, automated, or substantial tasks

using real-time datasets. However, this isolated approach wasn't always ideal due to the risk of technical issues.[2]

2: Relying solely on such systems not only constitutes poor practice but may also risk task failures from technical glitches.

Today, with digital platforms' proliferation, the volume of accessible data is unparalleled. These platforms can record every user interaction. Consider a **cross-sectional dataset** that captures phrases spoken by a user at home through devices like Amazon's *Alexa*, several Google searches over days, frequent product views across websites, and the ensuing transaction records.

Instead of selective data gathering, every interaction is catalogued. Beyond the ethical concerns surrounding such comprehensive use of personal data (as highlighted in Section 14.3, *Ethics in the Data Science Context*), there are technical challenges, such as processing massive data and deriving meaningful insights from it.

Data inquiries now predominantly fall into **reporting** ("what occurred?"), **real-time analytics** ("what's transpiring now?"), and **predictive modelling** ("what might unfold?"), as opposed to **causal inference** ("why did it happen?").

A significant challenge for data scientists today is to format these vast data repositories to be algorithm-friendly. As a result, a key focus of modern **data engineering**, as discussed in subsequent sections, pertains to the processing of this ever-growing data influx.

Once data is appropriately organized, data scientists deploy machine learning techniques to develop **proofs-of-concept**. Subsequently, AI/ML engineers transform these into **deployable models** as part of **data pipelines**, encapsulating the broader domain of data engineering.

Though data and AI/ML engineering have been around for a while, the advent of **cloud computing** places a heightened emphasis on their importance, sometimes overshadowing data science in specific sectors.

Organizations with **low data maturity** often lean on software like Excel to craft makeshift solutions for standard data pipeline tasks.[3] Such makeshift systems might suffice for their immediate needs but are insufficient when dealing with expansive datasets.

3: We could easily join those criticizing these rudimentary methods, and while we generally concur, we aren't suggesting Excel should NEVER be used. It has its place, albeit limited.

In contrast, entities with **enhanced data maturity** use a mix of SQL warehouse queries and `R/Python` scripts. They aggregate data using the entire population for reporting and then sample to build proofs-of-concept on local systems. However, even these methods don't exploit the full potential offered by contemporary tools and **data stacks**.

At its core, data engineering aims at collecting, storing, and analyzing data **at scale**.[4]

4: Scalability refers to a system's capability to handle a growing workload efficiently or its potential to expand to accommodate that growth.

Investing in data engineering components is invaluable for such extensive operations, a topic explored further below.

In smaller enterprises, roles in data engineering and data science may overlap, especially if the company's needs tilt more towards data engineering. Conversely, many larger companies maintain **dedicated** data engineers, responsible for managing **data pipelines** and overseeing **data warehouses**.

17.2 Data Engineering

Data engineering is best understood as a subset of computer engineering that emphasizes designing, constructing, and maintaining systems specifically geared towards data handling – from collection and ingestion to analysis and presentation. Given its roots in computer engineering, a grasp of certain computer engineering fundamentals can be beneficial when diving into data engineering.

At their core, computers comprise:

- **memory**, which is visualized as labeled containers holding binary data (ones and zeros), and
- **circuits**, systems that process memory content as input to produce outputs, which are then stored back in memory.

A computer processor's array of circuits constitutes its **instruction set**. When executing a computer program, these instructions are followed in a specific sequence.

In this frame of reference, data represents distinct **patterns** in memory. These patterns can be:

- **duplicated** into other memory locations;
- **relocated** through copying followed by erasure of the original, and/or
- **altered** using the patterns as inputs for a series of instructions, resulting in new patterns stored back in memory.

Notably, computer programs also manifest as data in memory. They're loaded into the processor, translated into basic hard-coded instructions, and then actualize the program's directives.

Software engineering spotlights the software side of computers. As defined by IEEE: "The systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software" [4]. Essentially, computer engineering's objective is to create programs that manipulate binary patterns via suitable instruction sets.

Data Engineering and IT How does **information technology** (IT) relate to this? Generally, IT revolves around technology that manipulates data and information, extending beyond just computer systems to encompass communication systems and even television. Thus, computer and data engineering can be perceived as IT subdomains.

However, colloquially, IT often denotes the use of **pre-existing software and hardware for data and information management**. IT professionals amalgamate these technologies to formulate comprehensive systems with specific information processing capabilities.

Data engineers, in this landscape, traverse the domains of software engineers and IT professionals. They might design **bespoke software applications** and **tailored architectures** for unique data types, but typically within the scope of **using established applications** to **construct data pipeline infrastructures**.

In the modern age, awash with data, data engineering's relevance has exploded, with applications spanning nearly every sector. Organizations, flush with vast data troves, are investing in the right talent and tools to refine this raw data, prepping it for data scientists and analysts.

Data Team Roles Within a data team, data engineers enable streamlined data collection from varied sources. Subsequently, database analysts manage this data, priming it for analytical tasks and inclusion in data projects. Let's delve deeper into these roles (for a comparison, refer to Section 13.1.3, *Roles and Responsibilities*).

5: From sources like paper tax forms manually fed into databases, or real-time data from online tax platforms that's streamed into databases.

Data engineers obtain data.[5] They organize, disperse, and store this data in data lakes and warehouses. Moreover, they craft tools and data models, aiding data scientists in data querying.

Data scientists use the data curated by data engineers to extract insights, build prototype predictive models, evaluate and enhance outcomes, and construct data models. Typically, they employ languages like `Python` or `R` and work within analytical notebooks such as RMarkdown or Jupyter. These notebooks interface with clusters, converting queries into commands for big data platforms (like Apache Spark).

ML engineers implement and deploy data models, acting as a bridge between data engineers and scientists. They take prototype ideas and upscale them, establishing feedback mechanisms to allow data scientists to monitor aggregate performance and rectify any issues in their prototype solutions.

See [10] for another perspective on these roles, in particular as they relate to the job market.

17.2.1 Data Pipelines

The work of data engineers largely revolves around data pipelines, which conduct a series of routine data manipulation tasks in an automated manner. Intriguingly, their work shares similarities with chemical systems and process engineers, but with a focus on data transformation rather than chemical transformation.

Here are typical tasks for data pipelines:

1. **acquiring** datasets that match business requirements;
2. **developing** algorithms to convert data into actionable insights;
3. **building**, **testing**, and **maintaining** database pipeline structures;
4. **collaborating** with management to grasp company goals;
5. **creating** new data validation techniques and data analysis tools;
6. **ensuring** adherence to data governance and security policies.

A functional data pipeline consists of **interfaces** and **mechanisms** that aid in information flow. Data engineers establish and manage this **data infrastructure**, using it to ready data for analysis by data analysts and scientists. It also delivers the outcomes of this data transformation and analysis to the end users.

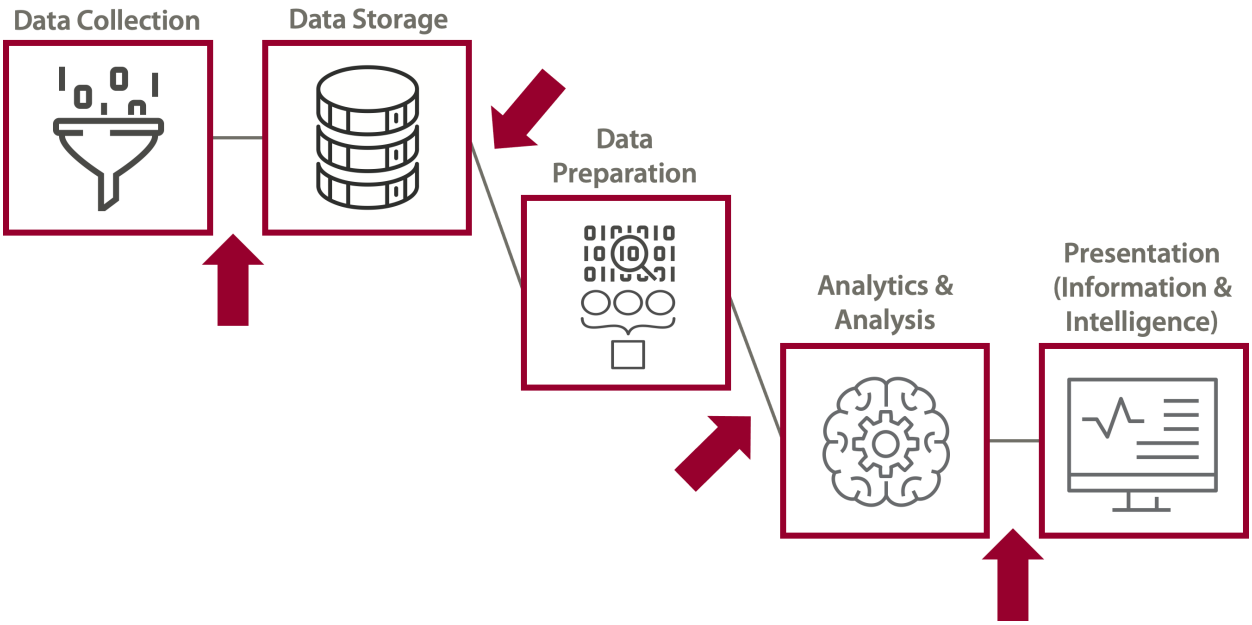

Figure 17.1: Illustration of a conceptual data pipeline highlighting the components and transitions.

Data can originate from myriad sources in diverse formats and sizes. Converting this vast amount of raw data into a usable format for data scientists is termed **building a data pipeline**.

Generally, pipelines encompass the following stages.

1. **ingestion**: collecting data from various sources;
2. **processing**: cleansing and transferring the data to a suitable data storage;
3. **storage**: placing the data in a reachable location and crafting a data model; and
4. **access**: facilitating access to cleaned data for analysis and display.

While the number and sequence of steps can differ across frameworks, they must remain consistent within a particular program.

A primary challenge for data engineers is crafting a pipeline that can **operate in near-real-time when prompted**, providing users with **current information** swiftly.[6]

6: Considering the dataset size.

Data engineers typically start by designing a prototype of a functional pipeline. Once tested, a more resilient pipeline is devised, which then undergoes **deployment** and **production**.

Related tasks include:

- checking data quality;
- enhancing query performance;
- forming a continuous integration and delivery environment;
- aggregating and storing data from various sources following a specific data model; and
- implementing machine learning and data science methods on distributed systems.

Consider this scenario: The *Canada Revenue Agency* (CRA) aims to determine the number of individuals in a region not submitting tax returns

and, as a result, forgoing net positive benefits.[7] They also want to ascertain who among them possibly missed the deadline due to unawareness rather than oversight.

7: After submitting, the benefits received exceed the taxes paid.

Potential pipeline processes include:

- gathering and storing data from third-party reports indicating tax filing numbers in the region;
- assessing historical tax filing data for that region;
- using predictive modeling to assess known non-filers' characteristics to predict unintentional filing oversights;
- presenting results via a dashboard.

Data Pipeline Connections In our framework, the links between pipeline components facilitate:

- transition from **collection** methods to an effective **storage** area;
- movement from **storage** to **preparation** where data undergoes transformation;
- transfer of **transformed** data to **analysis** or **modeling** phases; and
- use of **modeling** outcomes for **presentation**.

Typical challenges are:

- transferring data into a **data lake** can be time-consuming, especially with repeated data ingestion tasks;
- data platforms are evolving, leading to a cycle of building, maintaining, then **rebuilding** and **continuous maintenance**;
- the growing need for real-time data means **low latency** pipelines[8] become essential, making *Service Level Agreements* (SLAs) harder to establish.[9]

8: Those with minor delays.

9: Data pipeline SLAs detail client-service provider agreements integrated into client pipelines. Such SLAs necessitate regular performance checks and tuning.

Without careful planning, these challenges can quickly escalate.

Data Pipeline Operations A data pipeline is essentially an **automated sequence of data operations**. This can range from simple tasks, like moving data from one place to another, to more intricate processes that aggregate, analyze, and present data.

Common elements for each step include:

- **data sources** – applications, mobile apps, microservices, and more;
- **data integration** – ETL, stream data integration, and the like;
- **data storage** – MDM, data lakes, warehouses, etc.;
- **data analysis** – machine learning, AI, predictive analytics, etc.;
- **delivery and presentation** – dashboards, reports, notifications, and more.

Furthermore, pipelines allow breaking down a large task into manageable steps, optimizing each phase.[10] For example, it might be beneficial to use a specific language or framework for a pipeline segment. With a monolithic script, all processes, from data collection to presentation, would need to use the same tool, which might not be optimal.

10: This enhances efficiency, scalability, and reusability.

A more efficient strategy, adopted by most **data pipeline tools**, is to choose the best framework for each component.

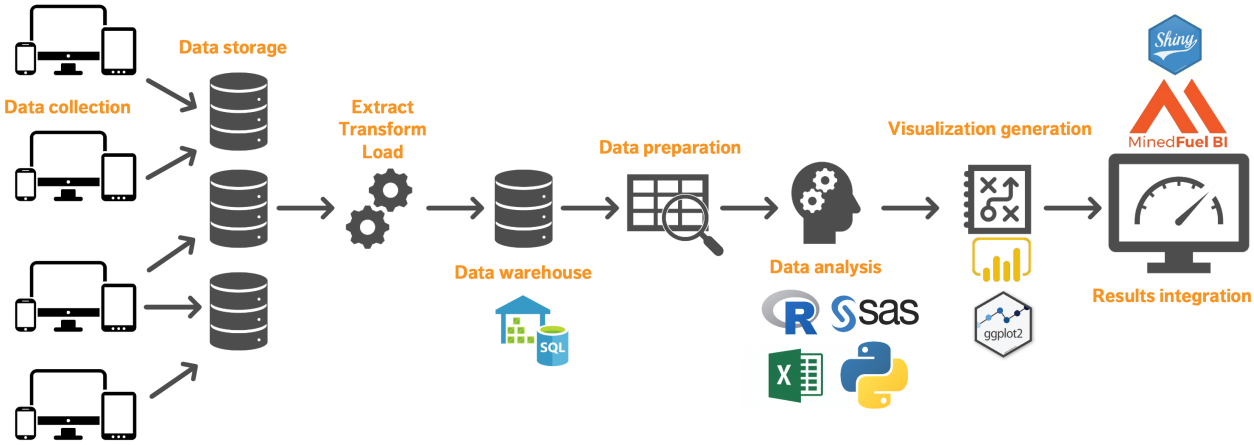

Figure 17.2: A depiction of a data visualization pipeline showcasing different component choices.

ETL Framework In the realm of data pipeline design, one cannot overlook the significance of the **ETL framework**, an acronym for **E**xtracting, **T**ransforming, and **L**oading data. While ETL has its origins in the earlier domains of **data marts** and **warehouses**, its principles remain indispensable in shaping modern-day data pipelines.

All data processes start with the **extraction** of data from a given source or temporary holding. When data comes from multiple sources, a subsequent **loading** phase often follows the extraction. This step ensures that multiple systems and processes uniformly handle the data from the same extraction point. In scenarios where data from multiple sources "converge", centralizing this data before its **transformation** proves beneficial.[11]

11: Depending on specific requirements, post-transformation, this data might either be transferred to a different location like a **data warehouse** or undergo transformations right at the source before a final load.

Once data from all pertinent sources is assembled, the onus falls on the data engineer to strategize the most effective way to merge these datasets. This strategy often encompasses the creation of data pipeline components that facilitate a seamless flow of data from the source systems to a format amenable for querying by business intelligence tools.

A pivotal role of data engineers lies in assuring the reliability and correctness of these pipelines. Often, this involves reconciling data or even deploying supplementary pipelines to corroborate against the original data sources. They are also tasked with ensuring a consistent and updated data flow, an endeavour often supported by various monitoring solutions and **site reliability engineering** (SRE) methodologies.

User-centric products typically source data from a diverse range of origins, potentially spanning multiple systems or third-party **integrations**. It's imperative that such data aligns with end-user specifications and maintains its **integrity**. However, challenges arise when data sourcing relies on inter-dependent systems. Each execution of the pipeline mandates fresh queries to these systems, potentially extending the pipeline's runtime. Even with such intricacies, the ETL framework can often render subsequent data pipelines more efficient.

Timely access to updated information is a frequent demand from business units. In navigating this, data engineers must holistically evaluate pipeline performance. This evaluation must account for the frequency of new data

inflows, the duration of transformation phases, and the time required to update the final data storage location.

Data Architecture Optimal results in data management often hinge on a shared understanding of the organizational structure of the data and its flow mechanisms. Serving as a blueprint for these aspects is the **data architecture**, which ideally encompasses the following notions.

- **Storage layout:** an overview of the methodology and locations of data storage. This should factor in the standards pertaining to file paths, file type specifics for file/object storage, as well as naming conventions for databases, schemas, views, and tables in the context of database storage.
- **Data landscape:** a depiction of how data is categorized within a specific repository.
- **Data abstractions:** a comprehensive elucidation of the components of any data abstractions present on the platform, complete with illustrative diagrams.[12]
- **Data access:** an outline detailing the authorization mechanisms of the data repository, capturing nuances like user-role mapping, the structure of role hierarchies, and privilege allocations to these roles.

12: These serve as foundational blueprints for crafting various elements on the data platform.

Additionally, a well-structured data architecture clearly outlines how data travels between different repositories within the platform. Such insights answer pivotal questions about permissible sources and destinations for data and the tools enlisted for these transfers.

Data Governance and Self-Serviceability **Data governance**[13] is a non-negotiable aspect; while data platforms are envisioned to catalyze innovation by democratizing data access, discernment is required in determining access levels, especially with sensitive datasets.

13: This pertains to a comprehensive data management strategy that empowers an organization to uphold superior data quality throughout its entire lifecycle. It encompasses controls that align with business goals. Primary areas of focus include data availability, usability, consistency, integrity, security, and regulatory compliance. It mandates protocols for upholding data quality across an organization, holding entities accountable for discrepancies and ensuring a universal data access paradigm. [12]

Specific datasets, especially from realms like sales and HR, often house sensitive information necessitating restricted access. Further, data sets comprising customer or health-related data are often tethered to **compliance protocols**, dictating access and usage parameters.

For users aiming to access a dataset on the platform, there should be robust mechanisms in place to petition for access. Concurrently, there should be workflows to review these petitions, ensuring that data access remains judicious and controlled. This ethos of **self-serviceability** is instrumental in liberating data access on any platform.

Further deepening the canvas of self-serviceability is the capability for users to request the instantiation of new structures or objects within the data repository. As evolving use cases emerge, the creation of new "workspaces" allows data engineering teams to devise new data transformations and datasets. This is but a glimpse into the myriad scenarios where self-serviceability amplifies the autonomy and dynamism of the data platform.

17.2.2 Automatic Deployment and Operations

Automating data pipelines can range from simple tasks like directing data between locations to intricate operations such as automated aggregation, transformation, and redistribution of data from various sources.

It's become increasingly viable to automate the ingestion of petabytes of ever-evolving data. This enables pipelines to efficiently provide data suitable for analytics, data science, and machine learning.

Notable **automated operations** include:

- **on-the-fly data processing** from files or real-time sources like Kafka, DBMS, NoSQL, and others;
- **automatic detection of schema** (or column) changes across different data formats;
- real-time tracking of incoming data;
- implementing **auto-backup measures** to prevent data loss.

As referenced earlier, ETL provides essential **decision points** for data pipeline creation. With contemporary tools, data engineers can minimize development duration, concentrating on business logic and data quality checks using SQL, `Python`, `R`, and similar. Achievable actions include:

- **intent-focused declarative development**, clarifying the problem, and simplifying the solution;
- automatic generation of **detailed data lineage** and handling table dependencies **within the pipeline**;
- auto-checking for **missing components**, **syntax anomalies**, and ensuring **data pipeline recovery**.

To enhance **data reliability**, we can:

- establish **data quality and integrity measures** within the pipeline;
- address **data discrepancies** using **pre-set policies** such as alerts, quarantine, or dropping faulty data;
- use **metrics** that continuously monitor, log, and report on data quality throughout the pipeline.

Strategies for Automated Pipeline Deployment Traditional software deployment often involved:

- initiating a build;
- manually transferring the build to a production server, and
- an ad-hoc "test" to verify application functionality.

This approach is neither scalable nor efficient, with manual steps increasing risk. The ultimate aim of automated pipeline design and deployment is to prototype and validate scalable components **before their full deployment**, while supporting an ongoing development cycle. Agile methodologies align well here.[14]

14: See What Is Agile? And When to Use It for an overview.

Effective Testing Practices Deploying in a live environment without exhaustive testing can expose end-users to unresolved bugs or issues. Optimal **code promotion** practices involve automated verification processes checking code functionality across varied scenarios:

- **unit tests** examine code segments, verifying if, given specific inputs, they yield expected outputs without depending on external code;[15]
- **integration testing** verifies that multiple code segments cohesively function and produce anticipated results.[16]

Combining both testing methods with modern strategies like **blue-green deployments** drastically reduces potential disruptions when introducing new code.[17]

Disaster Recovery Protocols Before advancing changes to a system, it's imperative to pass them through rigorous testing. Furthermore, a contingency plan for **system failure** is vital. Systems should be robust against catastrophic failures. Typical metrics in data engineering for disaster recovery include **Recovery Time Objective** (RTO) and **Recovery Point Objective** (RPO).[18]

During disaster recovery, it's essential to gauge the impact on consumers and system downtime. Data engineers are tasked with ensuring that data pipelines and storage solutions comply with acceptable recovery benchmarks.

Guidelines for Effective Pipeline Development With the influx of data into platforms, adopting **development best practices** is crucial to guarantee reliability, especially with the dynamic nature of data platforms. Standard practices encompass:

- leveraging **Source Code Management** (SCM) utilities;
- using **Continuous Integration** (CI)/**Continuous Delivery** (CD);
- designing using **diverse deployment settings**;
- prioritizing testing and data quality;
- embracing **Infrastructure as Code** (IaC);[19]
- using **database change control**;
- formulating effective **rollback strategies**; and
- constant monitoring and alert mechanisms.

The guiding principles revolve around **automation**, **testing**, and **monitoring**:

- streamlining the construction and validation of digital components;
- forming deployment pipelines to distribute these components;
- evaluating deployments and advancing components through different stages; and
- incorporating data quality assessments in pipelines, and raising flags on inconsistencies.

When tests detect issues, automated rollback procedures should initiate. Given the constantly shifting landscape of data governance standards, tooling, best practices, security measures, and business requirements, deployments must be both **automated** and **verifiable**.

15: This not only validates the logic but ensures the code operates as anticipated.

16: Ensuring seamless interplay between systems is the primary objective of this test layer.

17: "Blue-green deployment" is a strategy used for releasing applications by having two separate environments – a "blue" one and a "green" one:

- the **blue environment** is the currently running production environment, serving all the user traffic;
- the **green environment** is a clone of the blue environment, to which updates or new versions are deployed.

Once the green environment is ready and fully tested, the traffic is switched from the blue environment to the green environment, making the transition seamless to users. This approach is popular because it allows deployments with no downtime/service interruption, and because if something goes wrong in the green environment after the switch, traffic can be quickly rerouted back to the blue environment, ensuring high availability and minimizing disruptions. Note that there may be issues with data synchronization, especially for databases. Additionally, since two environments are running concurrently (at least during deployment), the infrastructure costs can double (albeit temporarily).

18: RTO represents how long the system or application can be down before there's a significant impact on the organization; RPO represents how much data an organization can afford to lose in the event of an incident.

19: Defining and managing servers, databases, networks, and other infrastructure components through code, rather than manually.

17.2.3 Scheduled Pipelines and Workflows

There are three primary **data pipeline architectures** that cater to the automated scheduling of tasks and workflows:

- **batch data pipelines** transfer vast data amounts at designated intervals;[20] ;
- **streaming data pipelines** transfer data from its origin to its destination immediately upon being generated,[21] and
- **change capture data pipelines**, whose role is to renew large datasets and uphold data uniformity across platforms.[22]

20: Prevalent in scenarios where tables require daily or weekly updates for reporting or dashboard functionality.

21: They often fill data lakes, serve data warehouse integration, or disseminate data for real-time uses, such as stock price updates or on-the-spot fraud detection.

22: Pivotal when datasets are shared among multiple systems.

Blueprint for Efficient Pipelines Conceptually, constructing an **efficient** data pipeline is a systematic six-phase endeavour:

1. **cataloging and overseeing data** entails facilitating enterprise-wide access to trustworthy and compliant data;
2. **proficient data ingestion** involves drawing data from myriad sources – on-premises databases, SaaS applications, IoT devices, streaming apps – and channeling it into a cloud-centric data lake;
3. **data fusion** cleans, enriches, and remodels the data – the creation of specific zones, such as landing areas, enrichment hubs, and enterprise territories, is integral here;
4. **implementation of data quality protocols** ensures data purity and organizational distribution, bolstering DataOps;[23]
5. **data refinement** prepares the sanitized data to be migrated to a cloud data warehouse, thus allowing for self-driven analytics and data science scenarios, and
6. **real-time data processing** ensures that insights are gleaned from real-time data sources, like Kafka, and subsequently channeled to a cloud data warehouse for analytical use.

23: DataOps, starting as best practices, has evolved into a comprehensive data analytics methodology. It addresses the entire data life cycle, fostering collaboration between data analytics groups and IT functionalities. [13]

To support ML/AI and process big data at reasonable service level objectives, an efficient pipeline should also:

- **seamlessly deploy and process** any data on any cloud ecosystem, such as Amazon Web Services (AWS), Microsoft Azure, Google Cloud, and Snowflake for both batch and real-time processing;
- **efficiently ingest data** from any source,[24] into any target, such as cloud data warehouses and data lakes;
- **detect schema drift** in relational data base management systems (RDBMS) schema [11] in the source database or a modification to a table,[25] and **automatically replicate** the target changes in real time for data synchronization and real-time analytics use cases;
- **provide a simple wizard-based interface** with no hand coding for a unified experience;
- **incorporate automation and intelligence capabilities** such as auto-tuning, auto-provisioning, and auto-scaling to design time and run-time, and
- **deploy in a fully managed advanced server-less environment** for improving productivity and operational efficiency.

24: Such as legacy on-premises systems, databases, change data capture (CDC) sources, applications, or IoT sources.

25: Such as adding a column or modifying a column size.

Assessing Pipeline Performance and SLO A pivotal performance metric is the pipeline's alignment with business prerequisites. **Service level objectives** (SLOs) offer concrete performance benchmarks against set standards.

For instance, a system could have the following SLO framework:

- **data timeliness** – 90% of product advice should stem from user online activity within the last three minutes;
- **data accuracy** – fewer than 0.5% of monthly client bills should have inaccuracies;
- **data isolation and resource allocation** – within a workday, priority payments should be processed within 10 minutes of submission, with standard ones being settled by the subsequent business day.

Data freshness relates to data's relevance in regards to its age. Typical SLOs for data freshness encompass:

- x**% of data processed within** y **time units [sec, min, days]** – this is commonly used for batch pipelines that process bounded data sources; the metrics are the input and output data sizes at key processing steps relative to the elapsed pipeline run-time; we may choose a step that reads an input dataset and another step that processes each item of the input;
- **oldest data shouldn't exceed** y **time units [sec, min, days]** – this is commonly used for streaming pipelines that process data from unbounded sources; the metrics indicate how long the pipeline takes to process data, such as the age of the oldest unprocessed item,[26] or the age of the most recently processed item;
- **pipeline task completion within** y **time units [sec, min, days]** – this sets a deadline for successful completion and is commonly used for batch pipelines that process data from bounded data sources; it requires the total pipeline-elapsed time and job-completion status, in addition to other signals that indicate the success of the job.[27]

26: That is, how long an unprocessed item has been in the queue.

27: For example, the percentage of processed elements that result in errors.

Data correctness refers to data being free of errors. We can determine data correctness through different means. One method is to check whether the data is consistent by using a set of validation rules, such as rules that use regular expressions (regexps). Another method is to have a domain expert verify that the data is correct, perhaps by checking it against reference data.[28] These reference datasets can then be stored and used for different pipeline tests.

28: One challenge is that reference data for validating correctness might not always be available. Therefore, there might be a need to generate reference data using automated tools, or even manually.

With reference datasets, we can verify data correctness in the following contexts:

- **unit and integration tests**, which are automated through continuous integration;
- **end-to-end pipeline tests**, which can be executed in a pre-production environment after the pipeline has successfully passed unit and integration tests, and is automated *via* continuous delivery, and/or
- **pipelines running in production**, when using monitoring to observe metrics related to data correctness.

For running pipelines, defining a data correctness target usually involves measuring correctness over a period of time, such as:

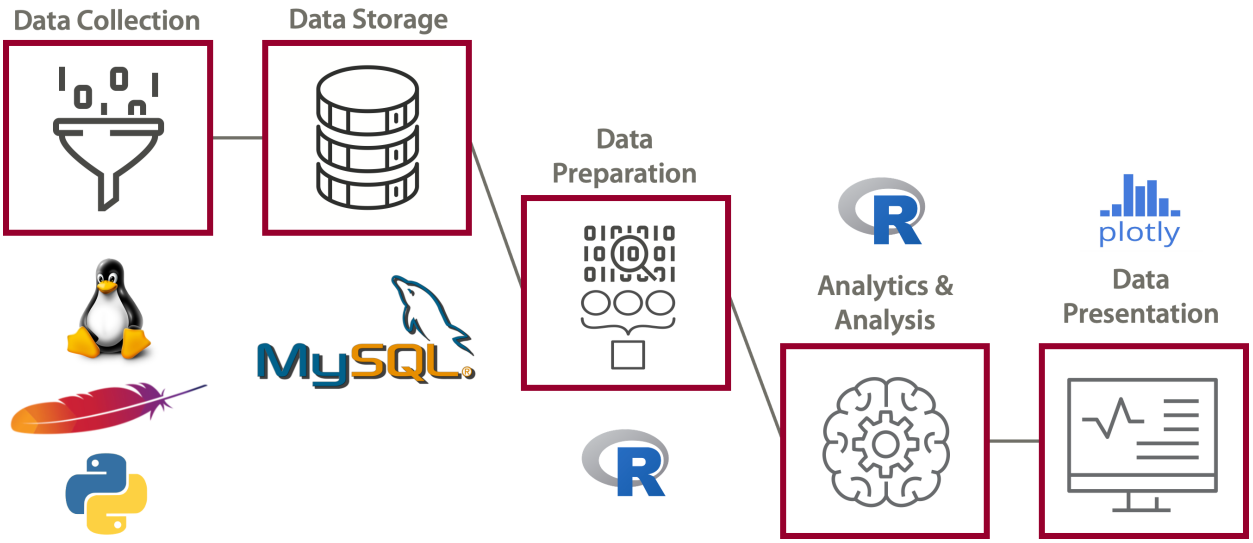

Figure 17.3: An open-source data analysis pipeline.

- **on a per-job basis, fewer than x% of input items contain data errors** – this SLO can be used to measure data correctness for batch pipelines;[29]
- **over an y-minute moving window, fewer than x% of input items contain data errors** – this SLO can be used to measure data correctness for streaming pipelines.[30]

To measure these SLO, we can use metrics over a suitable period of time to accumulate the number of errors by type, such as the data being incorrect due to a malformed schema, or the data being outside a valid range.

29: As an example, consider: "For each daily batch job to process electricity meter readings, fewer than 3% of readings contain data entry errors".

30: As an example, consider: "Fewer than 2% of electricity meter readings over the last hour contain negative values."

17.2.4 Data Engineering Tools

While it is unlikely that any one data engineer could achieve mastery over all possible data engineering tools, it would be beneficial for data teams to have competencies in a fair number of the following:[31]

- **analytical databases** (Big Query, Redshift, Synapse, etc.)
- **ETL** (Spark, Databricks, DataFlow, DataPrep, etc.)
- **scalable compute engines** (GKE, AKS, EC2, DataProc, etc.)
- **process orchestration** (AirFlow / Cloud Composer, Bat, Azure Data Factory, etc.)
- **platform deployment and scaling** (Terraform, custom tools, etc.)
- **visualization tools** (Power BI, Tableau, Google Data Studio, D3.js, ggplot2, etc.)
- **programming** (tidyverse, numpy, pandas, matplotlib, scikit-learn, scipy, Spark, Scala, Java, SQL, T-SQL, H-SQL, PL/SQL, etc.)

Here are some currently popular pipeline tools [2].

1. **Luigi** (Spotify) builds long-running pipelines (thousands of tasks stretching across days or weeks); it is a `Python` module available on an open-source license under Apache. It addresses the "plumbing" issues typically associated with long-running batch processes,

31: The content of this section is highly time-sensitive and is liable to have changed completely within 1-2 years from publication. That's life in the fast data engineering lane for you!

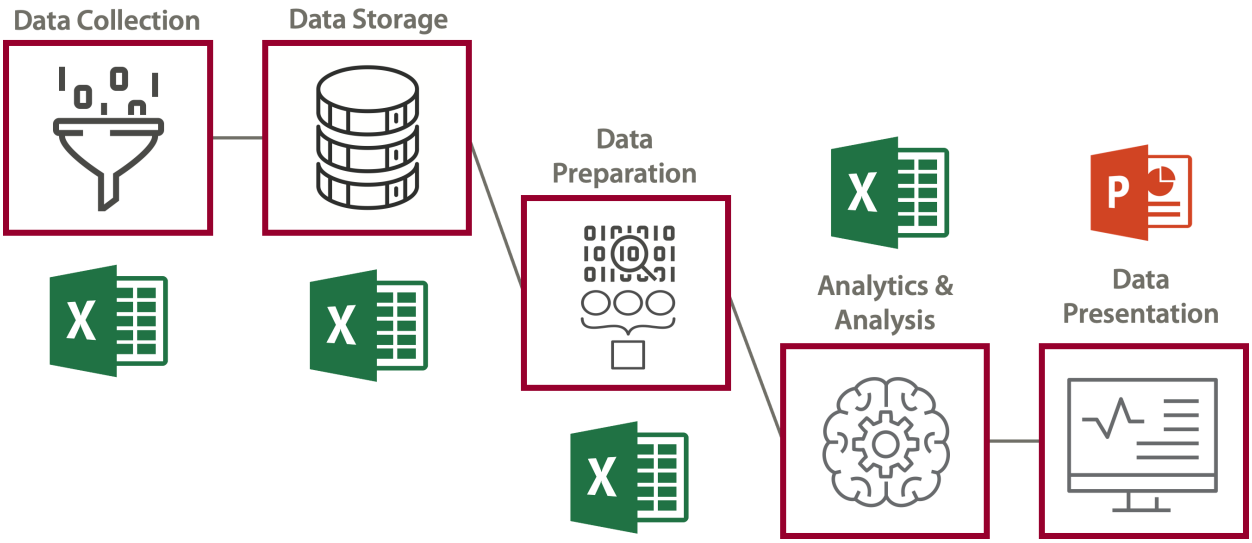

Figure 17.4: An unfortunately still far-too-common data analysis pipeline.

where many tasks need to be chained together (Hadoop jobs, dumping data to/from databases, running machine learning algorithms, etc.).[32]

2. **Airflow** (AirBnB) is used to build, monitor, and retrofit data pipelines. It is a very general system, capable of handling flows for a variety of tools and highly complex pipelines; it is good tool for pipeline orchestration and monitoring. It connects well with other systems (databases, Spark, Kubernetes, etc.).[33]
3. **scikit-learn pipelines:** scikit-learn pipelines are not used to orchestrate big tasks from different services; rather they help make code cleaner and easier to reproduce/re-use. They are found in scikit-learn, a popular `Python` data science module. The pipelines allow users to concatenate a series of **transformers**, followed by a final **estimator**; this is useful for model training and data processing, for instance. With scikit-learn pipelines, data science workflows are easy to read and understand, which also makes it easier to spot issues such as **data leakage** (unplanned or unauthorized release of data). The pipelines only work with scikit-learn transformers and estimators, however, and they must all be run within the same run-time, which makes it impossible to run different pipeline parts on different worker nodes while keeping a single control point.
4. **Pandas** (`Python`) or **Tidyverse** (`R`) **Pipes:** pandas and the tidyverse are popular data analysis and manipulation libraries. When data analysis becomes very sophisticated, the underlying code tends to become messier. Pandas and tidyverse pipes keep the code clean by allowing users to concatenate multiple tasks using a simple framework, similar to scikit-learn pipelines. These pipes have one criterion, the "data frame in, data frame out" principle: every step consists of a function with a **data frame** and other parameters as arguments, and a data frame as output. Users can add as many steps as needed to the pipe, as long as the criterion is satisfied.

32: Luigi uses 3 steps to build pipelines: `requires()` defines the dependencies between the tasks, `output()` defines the target of the task, and `run()` defines the computation performed by each task. Luigi tasks are intricately connected with the data that feeds into them, making it difficult to create, modify, and test a single task, but relatively easy to string tasks together.

33: Airflow defines workflows as Directed Acyclic Graphs (DAG), and tasks are instantiated dynamically. Airflow is built around: **hooks** (high-level interfaces for connections to external platforms), **operators** (predefined tasks that become DAG nodes), **executors** (run jobs remotely, handle message queuing, and decide which worker will execute each task), and **schedulers** (trigger scheduled workflows and submit tasks to the executors).

17.3 Data Management

As covered in the previous section, a major element of data engineering (that is, developing data pipelines) involves moving data from storage to storage as it is processed. In this sense, data storage can be viewed as the metaphorical heart of the data pipeline, while the machine learning model components of the pipeline could be thought of as the brains. In this section we will focus on this metaphorical heart, and consider management of the brains (analytics and machine learning models) in Section 17.4, *Reporting and Deployment*.

Computers have advanced significantly in their ability to store large amounts of data. In this section, we will cover **databases**, **data modeling**, and **data storage**. Readers are invited to refer to [11] (and Section 14.5, *Getting Insight From Data*) for more details.

17.3.1 Databases

Historically, computers relied on a **file-based system** (i.e., they manipulate data files). File-based systems face a number of shortcomings:

1. **data redundancy:** files and applications are created by different programmers from various departments over long periods of time. This can lead to redundancy, a situation that occurs in a database when a field needs to be updated in more than one table, inconsistencies in data format, the same info being stored in multiple files, and conflicting copies;
2. **data isolation:** it can prove difficult for new applications to retrieve the appropriate data, which might be stored in various files;
3. **data integrity:** maintenance may be required to ensure that data in a database are correct and consistent;
4. **security:** it can be difficult to enforce access constraints (if needed) when application requirements are added to the system in an *ad-hoc* manner, and
5. **concurrency:** if multiple users access the same file at the same time, there can be issues with file locking.

Spreadsheets were originally designed for a single user, which is reflected in their characteristics. They are adequate for single users or for small teams of users who have no need for complicated data manipulations.

Databases, on the other hand, hold massive amounts of information, and allow multiple concurrent users to quickly and securely access and query data using highly complex logic and language. They only need to be defined once before being accessed by various users.

Databases They consist of a representation of some aspect of the real world, in the form of a collection of **data elements** representing **real world information**.[34] They are:

- logical, coherent, and internally consistent;
- designed, built, and populated with data for a specific purpose;
- made up of data items, which are stored in fields,
- populated with tables, which are combinations of fields.

34: Most databases use a **structured query language** (SQL) for writing and querying data. SQL statements include: create/-drop/alter table; select, insert, update, delete; where, like, order by, group by, count, having; join.

A **database management system** (DBMS) is a collection of programs that enables users to create and maintain databases and control all access to them. The primary goal of a DBMS is to provide an environment for users to retrieve and store information in a convenient and efficient manner.

Data management is "simply" care-taking for the data so that it works for its users and remains useful for tasks. Managing information using a database allows data scientists to become strategic users of the data at their disposal. The processing power in a database can be used to manipulate the data it houses, namely: sort, match, link, aggregate, filter, compute contents, etc. Because of the versatility of databases, we find them powering all sorts of projects.

Database Benefits While databases might be overkill for small datasets, they have many benefits (especially for larger projects):

1. **self-describing nature of a database system:** a database contains the data and the metadata, which describes and defines relationships between tables in the database. This separation of data and information about the data makes a database system entirely different from the traditional file-based system in which the data definition is part of the application program;
2. **insulation between program and data** (also called program-data independence): in a file-based system, the structure of data files is defined in the application programs, so if a user wants to change the structure of a file, all programs that access it need to be changed as well. In a database system, the data structure is stored in the system catalogue and not in the programs. Therefore, one change (such as adding a new variable) is all that is needed to change the structure of a file;
3. **support for multiple views:** a database supports multiple views, or subsets, of the database. Each view contains data that is only of interest to the group of users subscribed to the particular view;
4. **sharing of data and multi-users:** many users can access data at the same time, through features called concurrency control strategies. The design of model multi-user database systems is a great improvement from those in the past which restricted usage to one user at a time,
5. **control of redundancy:** ideally, each data item is only found in one location, but redundancy can sometimes improve query performance (even though it should be kept to a minimum wherever possible).

Types of Databases Databases come in various flavours:

- the most common (as of 2022) are **relational databases**, in which data items are organized as a set of tables with columns and rows;
- data in **object-oriented databases** is represented in the form of objects, as in object-oriented programming (OOP);[35]
- in **distributed databases**, two or more files are located in different sites – such databases may be stored on multiple computers located in the same physical location, or scattered over different networks, etc.;

35: We discuss OOP briefly in Chapter 1, but there is a lot more to be said on the topic.

- **data warehouses** are central repository for data, designed specifically for fast query and analysis;
- **NoSQL warehouses** are non-relational databases that allow for unstructured and semi-structured data to be stored and manipulated (in contrast with relational databases which define how all the data inserted into the database must be composed) – NoSQL has grown popular as web apps have became more common and more complex,
- **graph databases** store data in terms of entities and relationships between entities – for instance, online transaction processing (OLTP) databases are speedy analytic databases designed for large numbers of transactions performed by multiple users.

Database Challenges Today's large enterprise databases often support very complex queries and are expected to deliver nearly instant responses to those queries. As a result, database administrators are constantly called upon to employ a wide variety of methods to help improve performance and overcome some common database challenges.

- **Absorbing significant increases in data volume**: the explosion of data coming in from sensors, connected machines, and dozens of other sources keeps database administrators scrambling to manage and organize their companies' data efficiently;
- **ensuring data security:** data breaches are happening at an ever-increasing rate, and hackers are getting more and more inventive – it is more important than ever to ensure that data is secure ... yet also easily accessible to users;
- **keeping up with demand:** in today's fast-moving business environment, companies need real-time access to their data to support timely decision-making and to take advantage of new opportunities;
- **managing and maintaining the database and infrastructure:** database administrators must continually watch the database for problems and perform preventative maintenance, as well as apply software upgrades and patches; as databases become more complex and data volumes grow, companies are faced with the expense of hiring additional talent to monitor and tune their databases;
- **removing limits on scalability:** some claim that businesses need to grow if they are going to survive, and so must their data management; but it is nearly impossible for database administrators to predict how much capacity the company will need, particularly with on-premises databases,
- **ensuring data residency, data sovereignty, or latency requirements:** some organizations have use cases that are better suited to run on-premises; in those cases, engineered systems that are pre-configured and pre-optimized for running the database are ideal.

Addressing all of these challenges can be time-consuming and can prevent database administrators from performing more strategic functions.

17.3.2 Database Modeling

Database modeling is a process used to define and analyze data requirements needed to support the business processes within the scope of corresponding information systems. This includes both data elements and structures/relationships between them.

1. Requirements are put into a **conceptual model** (tech independent specifications), which describes the semantics of a domain and the scope of the model. For example, a model of the interest area of an organization or industry. This consists of entity classes, representing the kinds of things of significance in the domain, and relationship assertions about associations between pairs of entity classes. A conceptual schema specifies the kinds of facts or propositions that can be expressed using the model. In that sense, it defines the allowed expressions in an artificial 'language' with a scope that is limited by the scope of the model.
2. The structure of the database data is put into a **logical model**, which describes the model semantics, as represented by a particular data manipulation technology. This consists of descriptions of tables and columns, object-oriented classes, and XML tags, among other things. The implementation of a single conceptual model may require multiple logical models. The logical models are then incorporated into a physical data model that organizes data into tables, which accounts for access, performance, and storage details.
3. The **physical data model** describes the physical means by which data is stored, including partitions, CPUs, tablespaces, and the like.

A database model, then, is a specification describing how a database is structured and used.

- The **flat (table) model** may not strictly qualify as a data model; it consists of a single, two-dimensional array of data elements, where all members of a given column are assumed to be roughly similar values, and all members of a row are assumed to be related to one another.
- The **network model** organizes data using two fundamental constructs: the records and the sets. Records contain fields, and sets define one-to-many relationships between records: one owner, many members. The network data model is an abstraction of the design concept used in the implementation of databases.
- The **hierarchical model** is similar to the network model except that links in the hierarchical model form a tree structure, while the network model allows arbitrary graphs.
- The **relational model** is a database model based on first-order predicate logic. Its core idea is to describe a database as a collection of predicates over a finite set of predicate variables, describing constraints on the possible values and combinations of values. The power of the relational data model lies in its mathematical foundations and its simple user-level paradigm.
- The **object-relational model** is similar to a relational database model, but objects, classes and inheritance are directly supported in database schemas and in the query language.

- **Object-role modeling** is an approach that has been defined as "attribute free" and "fact-based". The result is a verifiably correct system, from which other common artifacts, such as ERD, UML, and semantic models may be derived. Associations between data objects are described during the database design procedure, leading to inevitable database normalization.[36]
- The **star schema** is the simplest of the data warehouse schemas; it consists of a few "fact tables" (possibly only one, justifying the name) referencing any number of "dimension tables". The star schema is considered an important special case of the snowflake schema.

36: "Database normalization is a technique for creating database tables with suitable columns and keys by decomposing a large table into smaller logical units. The process also considers the demands of the environment in which the database resides. Normalization is an iterative process. Commonly, normalizing a database occurs through a series of tests. Each subsequent step decomposes tables into more manageable information, making the overall database logical and easier to work with." [9]

Data modeling can also be phrased as a high-level abstract design phase used to describe:

- the data contained in the database;
- the relationships between data items, and
- constraints on data.

The data items, relationships and constraints are all expressed using concepts provided by the high-level data model. Because these concepts do not include the implementation details, the result of the data modeling process is a semi-formal representation of the database structure. Database design includes logical design which is the definition of a database in a data model of a specific DBMS, and physical design which defines the internal database storage structure, file organization, and indexing techniques.

Database Design In database design, the first step is to identify **business rules** [11]. The design is then created and implemented using a DBMS.

- In an **external model,** the user's view of a database (multiple different external views) is closely related to the real world as perceived by each user.
- **Conceptual models** provide flexible data-structuring capabilities; they offer a "community view" of the entire database (logical structure). This contains the data stored in the database and it shows relationships including: constraints, semantic information (e.g., business rules), security and integrity information, etc.[37]
- **Internal models** are relational, network, and/or hierarchical data models. They consider the database as a collection of fixed-size records, closer to the physical level or the file structure. Internal models offer a representation of the database as seen by the DBMS and require the database designer to match the conceptual model's characteristics and constraints to those of the selected implementation model; this may involve mapping entities in the conceptual model to tables in the relational model, say.
- **Physical models** are physical representations of the database, its lowest level of abstraction. The focus is on how to deal with runtime, storage utilization and compression, file organization and access, and data encryption. The physical level is managed by the operating system; it provides concepts that describe how the data is stored in computer memory, in detail.

37: Conceptual models consider that a database is a collection of entities (objects) of various kinds, but they avoid detailed descriptions of the main data objects, in effect being independent of the eventual database implementation model.

Schemas We have already mentioned **schemas**, which are database descriptions represented by an **entity relationship diagram** (ERD, see *Structuring and Organizing Data* in Section 14.5). The most popular data models today are relational data models, although hierarchical and network data models are also often used on mainframe platforms. Relational data models describe the world as "a collection of inter-related relations (or tables)" [11].

Fundamental concepts include:

1. **relations** (table or file), which are subset of the Cartesian product of a list of domains characterized by a name;[38]
2. **tables** and **columns** house the basic data components, into which content can be broken down;[39]
3. a column's **domain**, the range of values found it the column, and
4. **records**, which contain related fields, and **degree**, which refers to the number of attributes.[40]

38: Within each table, the row represents a group of related data values; the row is known as a record or a tuple. Table columns are known as fields or attributes. Attributes are used to define a record, and a record contains a set of attributes.

39: Columns are combined into tables. Tables must have distinct names, no duplicate rows, and atomic entries (values that cannot be divided) in its columns.

40: Records and fields form the basis of all databases. A simple table provides the clearest picture of how records and fields work together in a database storage project.

17.3.3 Data Storage

Data storage refers to the collection and retention of digital information: the bits and bytes behind applications, network protocols, documents, media, address books, user preferences, and so on.

For computers, short term information is handled on random-access memory (RAM), and long-term information is held on storage volumes. Computers also distribute data by type. Markup languages have become popular formats for digital file storage: UML, XML, JSON, CSV, etc.

Data storage basically boils down to:

- the different ways to store different files;
- how to store them in the right kind of structures based on data type, and
- how those structures link together in a database.

It is data engineers and database analysts (**data managers**) that are responsible for storing collected and transformed data in various locations depending on the business requirements. Each combination of tool and location may store and access the data in different ways; the limitations, benefits, and use cases for each location and set of data must be taken into account as part of good data management.

For instance, let us assume a company is ingesting a million records a day from a particular data source. If the data is stored on a disk, we cannot simply append the daily updates to a singular file![41]) Any report or question needing a particular piece of information found on the disk would never be produced/answered.

41: This would be akin to looking for a needle in the world's largest haystack!

Instead, the company's data engineers would:

- know that the data needs to be **partitioned** across different files and directories within the file system to separate the data;
- evaluate the data and how it is loaded and consumed to determine the appropriate way to **split it**,
- determine how to **update** specific pieces of data as changes are applied to the data source.

At a more meta level, there are other factors to consider, such as:

- is the data **key-value based** (see *Structuring and Organizing Data*, in Section 14.5)?
- are there **complex relationships** in the data?
- does the data need to be **processed** or **joined** with other datasets?
- and so on.

Data Warehousing **Data warehousing** is the term used to refer to the storage process of structured data. Data storage is transforming rapidly, since files can be compressed to take up less memory space, and computers can hold more files locally and in RAM.

Cloud-based data warehousing solutions like *Snowflake*, *AWS Redshift*, *Azure Synapse*, and *Google BigQuery* allow for pay-per-use data warehouses too, giving seemingly infinite storage.[42]

42: This was written in August 2022; that list is liable to have changed quite a lot since then.

For **on-premise data warehousing solutions**, the investment is all up-front. The customers pay for the data warehousing solution, but do not get to see any return on investment while the hardware is set up, configured, and operationalized.[43] Initially, then, businesses are left with a severely under-utilized piece of hardware, making such a move a high-risk leap of faith for anyone but the biggest players.

43: It might take months, with millions of dollars already invested, just to be able to start to implement a solution for the first use case.

At some point in the warehouse lifetime, enough use cases exist to **eat the available hardware computer power or storage**. When this occurs, either more hardware must be acquired (at another large hit to the budget) or existing use cases that can be scaled back (and to what extent) must be identified to create the required "space". Purchasing more hardware in this stage is not as much of a leap of faith as the initial commitment was, but will once again leave the organization with an under-utilized data platform as new use cases are prioritized and solutions built for them.

In comparison, **cloud-based data warehouse solutions** use a pay-per-use cost model, where there is an opportunity to prove the value of a use case using an iterative approach. The initial step is to implement a use case solution with very light requirements to help gauge cost estimates and to understand how valuable that solution might be. Future iterations can expand on the solution, modifying the complexity of data transformation or how data flows through it, and even remove it to focus on another use case, if appropriate.

At no point is there a need to consider purchasing and installing additional hardware, as new warehouses or clusters can be created on-demand. Using a cloud-based data warehouse allows costs to scale according to the number of use cases and their complexity. However, this requires a level of expertise[44] and a lack of control over any changes to prices or policies that go with cloud tools.

44: Potentially different than the level of expertise required for on-premise warehousing, if not necessarily more sophisticated.

It is also important to consider who has access to what pieces of information that are stored (**data governance**). In practice, rules and regulations define who should have access to particular pieces of information within your organization. For a shipping company, as an example, we may need to separate the data that suppliers and customers can see at any given time, or ensure that different suppliers cannot see information about other suppliers.

This requires **data classification**, **tagging**, and **access constraints**. When gathering data from various systems, a data engineer is responsible for applying the classification and tagging rules upon collection.[45]

45: This might include adding additional data points to the collected data or storing data separately on disk.

Then, when the data is aggregated or transformed, the end result must include this same information. When setting up access constraints to the data, the data engineer also has to enforce the required policies.

As more organizations are obtaining additional data from ever-growing new sources, they are faced with new problems:

- securing the data;
- ensuring regulatory compliance, and
- general management of the data.

These are also problems that **data governance** exists to solve. [46]

46: Unfortunately, data governance is not achieved by using a specific tool or set of tools. Tools exist to support some of the aspects of data governance, but they only enhance existing data governance practices. Part of the challenge is that data governance is very much a "**people and process**" oriented discipline, intending to make data secure, usable, available, and of high quality.

17.4 Reporting and Deployment

Currently, the two main applications of data science in industry are **reporting** and **deployment** of machine learning models. In the context of data engineering, these machine learning models are **embedded** in the data pipeline.[47] As noted in a previous section, these machine learning models can be viewed metaphorically as the **brains** of the pipeline.

47: Data pipelines do not need to contain machine learning models; they may instead focus, for example, on business intelligence functionality. But we will assume that they do.

In an implemented context, managing machine learning models is known as **MLOps**. The traditional AI training cycle often involves a single pass of the following steps:

1. preparing the training data;
2. training the model,
3. evaluating the model.

These are still present in MLOps, with an increased focus on **ongoing monitoring/management** of the models embedded in the pipeline.[48]

48: For example, MLOps processes monitor models for drift in the context of the automated data stream, monitor models for performance relative to volume of data, and iteratively and automatically train and improve models over time based on the feedback received from this monitoring.

This iterative or interactive approach often includes **automated machine learning** (AutoML) capabilities; what happens outside the scope of the trained model is not included in this traditional definition.[49]

49: In modern data science contexts, MLOps may also refer to the entire data science process, from ingestion of the data to a live application that runs in a business environment and makes an impact at the business level. In this respect, MLOps overlaps with DataOps and DevOps.

17.4.1 Reports and Products

In the **research-first** approach to data science, which still dominates a lot of industry applications, machine learning models are used to generate **static** or **interactive reports** for business analysts; data science is handled as a **silo**, running batch predictions on historical data and returning the results for someone else to incorporate manually into applications.

In those conditions, there is little demand for **resiliency**, **scale**, **real-time access**, or **continuous integration and deployment** (CI/CD); the results are of limited value, in and of themselves, and are used more as proof-of-concept.

Most data science solutions and platforms today still start with a research workflow but fail to move past the proof-of-concept stage.[50]

50: CI/CD components refer to the training/re-training loop of a model, and do not extend to the full reporting and deployment pipeline. Even the concept of a CI/CD pipeline is often used to refer only to the training loop and do not extend to include the **entire operational pipeline**.

Generaly, we start an AI project with the **development of a model**:

1. data scientists **receive data**, which may be extracted manually from many sources;
2. the data is then **joined** and **cleaned** in an interactive way (using notebooks, perhaps), and
3. **training** and **experiments** are conducted while tracking results.

The model is **generated** and **tested/validated** until the results "look good",[51] at which point different teams take the results and attempt to integrate them into real-world applications.[52] In most cases, eventually, the original data science product is set aside and re-implemented in a **robust** and **scalable** way which fits production, but which may not be what the data scientist originally intended.

A **production pipeline** starts with automated data collection and preparation, continues with automated training and evaluation pipelines, and incorporates real-time application pipelines, data quality and model monitoring, feedback loops, etc.

As applications that demand real-time recommendations, prevent fraud, predict failures, and make decisions continue to be in demand, engineering efforts are required to make them feasible. Business needs have forced data science components to be **robust**, **performant**, **highly scalable**, and **aligned with agile software and DevOps practices**.[53]

Instead of this siloed, complex, and manual process, we should start by designing the ML elements of the pipeline using a **modular strategy**, where the different parts of the ML component provide a continuous, automated, and far simpler way to move from research and development to scalable production pipelines, without the need to refactor code, add glue logic, and spend significant efforts on data and ML engineering.[54]

51: That is, they meet a certain performance threshold.

52: Modern tools (such as Flask) allow data scientists to **serialize a model** into a file and then simply call the file to make predictions. However, the full process of monitoring, creating feedback loops, then retraining and updating the model still requires an underlying architecture.

53: It is all too often the case that **operationalizing** machine learning (in the sense of considering all business requirements, such as federated data sources, need for scale, critical implications of real-time data ingestion or transformation, online feature engineering, handling upgrades, monitoring, etc.) comes as an afterthought, making it all the more difficult to create real business value with AI.

54: ML production-ready pipelines have four key components:

1. **feature store**: collects, prepares, catalogues, and serves data features for development (offline) and real-time (online) usage;
2. **machine learning CI/CD pipeline:** automatically trains, tests, optimizes, and deploys or updates models using a snapshot of the production data (generated by the feature store) and code from the source control (Git);
3. **real-time/event-driven application pipeline:** includes the API handling, data preparation/enrichment, model serving, ensembles, driving and measuring actions, etc., and
4. **real-time data and model monitoring:** monitors data, models, and production components, and provides a feedback loop for exploring production data, identifying drift, alerting on anomalies or data quality issues, triggering re-training jobs, measuring business impact, etc.

17.4.2 Cloud and On-Premise Architecture

Organizations have to make decisions on how much of their data architecture to **build in-house**, and how much to build with **off-the-shelf** tools. Additionally, there are compromises and benefits to building infrastructure **on the cloud** (renting external resources) with potential to publish results for anyone in the world to see and build on, and building solutions on premise which depend heavily on local capacity and hardware.*

Developers must write new code for every data source, and may need to rewrite it if a vendor changes its API, or if the organization adopts a different data warehouse destination. Data engineers must also address **speed** and **scalability**: for time-sensitive analysis or business intelligence applications, ensuring **low latency** can be crucial to providing data that drives decisions.

* Many companies, such as Spotify, build their own pipelines from scratch to analyze data and understand user preferences, and map customers to music preferences, say. The main challenges to developing in-house pipelines are that different data sources provide **different application program interfaces** (API, see Section 16.3.6) and involve different kinds of technologies.

Data solutions need to be able to dynamically access more resources as data volume grows. Therefore, in-house pipelines can be expensive to **build** and **maintain**.

On-premise amateur-ish data pipelines ingest data in **pre-scheduled batches** (e.g., twice every hour or every night, say), and are not ideal for any real-time analytics solutions.[55]

55: Such pipelines may be all that is required in certain cases, such as establishing a proof-of-concept for business processes that require less frequent and manual decision-making. For example, a retailer can use them to make decisions about the order of recommendation of certain items in an online store, but may miss on recommending a product to an individual on a certain short-term buying spree in real-time.

ETL tools that work with in-house data warehouses do as much preparation work as possible, including transformation, prior to loading data into data warehouses. Cloud data warehouses like *Amazon Redshift*, *Google BigQuery*, *Azure SQL Data Warehouse*, and *Snowflake* can scale up and down in seconds or minutes, so developers can replicate raw data from disparate sources and define transformations in SQL and run them in the data warehouse after loading or at the time of query.

Just as there are cloud-native data warehouses, there also are **ETL services built for the cloud**. Organizations can set up a cloud-first platform for moving data in minutes, and data engineers can rely on the solution to monitor and handle unusual scenarios and failure points.[56]

56: Even without larger, more specialized tools, simple desktop tools such as *Tableau*, *Looker*, or *Microsoft's Power BI* can still be used to run queries and reports, and with a modern real-time pipeline the results will be current and immediately actionable.

Overall, cloud tools are becoming more and more popular to host data pipelines and by extension data science solutions.

Data engineering and data management are not always the most interesting aspects of the discipline for data analysts, but the long and the short of it is that it is impossible to conduct meaningful data science without the right data or without the right tools.[57] But tools do not only refer to the analytical tools; becoming familiar with the entire **data ecosystem** will pay off in the end.

57: This might perhaps be the only thing worth remembering from this chapter, especially since technology changes so quickly.

Chapter References

[1] *Introduction to Data Engineering* .

[2] Anouk Dutrée. *Data pipelines: what, why and which ones* . 2021.

[3] *What is Data Engineering? Everything You Need to Know in 2022* . phData, 2022.

[4] *Systems and software engineering - Vocabulary, ISO/IEC/IEEE std 24765:2010(E)* . 2010.

[5] M. Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly, 2017.

[6] Henryk Konsek. *Automating Data Pipelines: Types, Use Cases, Best Practices* .

[7] J. Kunigk et al. *Architecting Modern Data Platforms: A Guide to Enterprise Hadoop at Scale*. O'Reilly, 2018.

[8] T. Malaska and J. Seidman. *Foundations for Architecting Data Solutions: Managing Successful Data Projects*. O'Reilly, 2018.

[9] *What is Database Normalization?* .

[10] E. Uz. *Analysis of the data job market using "Ask HN: Who is hiring?" posts* . Aug. 2023.

[11] Adrienne Watt. *Database Design* . BCCampus, 2014.

[12] *Data Governance* .

[13] *DataOps* .