CHINMAY ARANKALLE | GARETH DWYER
BAS GEERDINK | KUNAL GERA
KEVIN LIAO | ANAND N.S.

THE ARTIFICIAL INTELLIGENCE INFRASTRUCTURE WORKSHOP

BUILD YOUR OWN HIGHLY SCALABLE AND ROBUST DATA STORAGE SYSTEMS THAT CAN SUPPORT A VARIETY OF CUTTING-EDGE AI APPLICATIONS

Packt

DATA SCIENCE & ARTIFICIAL INTELLIGENCE

THE ARTIFICIAL INTELLIGENCE INFRASTRUCTURE WORKSHOP

Build your own highly scalable and robust data storage systems that can support a variety of cutting-edge AI applications

Chinmay Arankalle, Gareth Dwyer, Bas Geerdink, Kunal Gera, Kevin Liao, and Anand N.S.

Packt

THE ARTIFICIAL INTELLIGENCE INFRASTRUCTURE WORKSHOP

Authors: Chinmay Arankalle, Gareth Dwyer, Bas Geerdink, Kunal Gera, Kevin Liao, and Anand N.S.

Reviewers: Brent Broadnax, John Wesley Doyle, Tim Hoolihan, Rochit Jain, Sasikanth Kotti, Asheesh Mehta, Arunkumar Nair, Madhav Pandya, Ashish Patel, Shovon Sengupta, and Ashish Tulsankar

Managing Editor: Ashish James

Acquisitions Editors: Manuraj Nair, Royluis Rodrigues, Kunal Sawant, Anindya Sil, Archie Vankar, and Karan Wadekar

Production Editor: Shantanu Zagade

Editorial Board: Megan Carlisle, Samuel Christa, Mahesh Dhyani, Heather Gopsill, Manasa Kumar, Alex Mazonowicz, Monesh Mirpuri, Bridget Neale, Dominic Pereira, Shiny Poojary, Abhishek Rane, Brendan Rodrigues, Erol Staveley, Ankita Thakur, Nitesh Thakur, and Jonathan Wray

First published: August 2020

Production reference: 1130820

ISBN: 978-1-80020-984-8
Published by Packt Publishing Ltd.
Livery Place, 35 Livery Street
Birmingham B3 2PB, UK

2

ARTIFICIAL INTELLIGENCE STORAGE REQUIREMENTS

OVERVIEW

In this chapter, you will learn how to differentiate between traditional data warehousing and modern AI-focused systems. You'll be able to describe the typical layers in an architecture that is suited for building AI systems, such as a data lake, and list the requirements for creating the storage layers for an AI system. Later, you will learn how to define the specific requirements per storage layer for a use case and identify the infrastructure as well as the software systems based on the requirements. By the end of this chapter, you'll be able to identify the requirements for data storage solutions for AI systems based on the data layers.

INTRODUCTION

In the previous chapter, we covered the fundamentals of data storage. In this chapter, we'll dive a little deeper into the architecture of **Artificial Intelligence** (**AI**) solutions, starting with the requirements that define them. This chapter will be a mixture of theoretical content and hands-on exercises, with real-life examples where AI is actively used.

Let's say you are a solution architect involved in the design of a new data lake. There are a lot of technology choices to be made that would have an impact on the people involved and on the long-term operations of the organization. It is great to have a set of requirements at the start of the project that each decision could be based on. Storing data essentially means writing data to disk or memory so that it is safe, secure, findable, and retrievable. There are many ways to store data: on-premise, in the cloud, on disk, in a database, in memory, and so on. Each way fulfills a set of requirements to a greater or lesser extent. Therefore, always think about your requirements before choosing a technology or launching an AI project.

When designing a solution for AI systems, it's important to start with the requirements for data storage. The storage solution (infrastructure and software) is determined by the type of data you want to store and the types of analysis you want to perform. AI-powered solutions usually require high scalability, big data stores, and high-performance access.

IT solutions tend to be either data-intensive or compute-intensive. Data-intensive solutions are "big data" systems that store large amounts of data in a distributed form but require relatively little processing power. An example of a data-intensive system is an online video website that just shows videos, but where no intelligent algorithms are being run to classify them or offer any suggestions about what to watch next to its users. Compute-intensive solutions can have smaller datasets but demand many computing resources from the hardware; for example, language translation software that is continuously being trained with neural networks.

AI projects are not your typical IT projects; they are both data-intensive and compute-intensive. Data scientists need to have access to huge amounts of data to build and train their models. Once trained, the models need to be served in production and fed through a data pipeline. It's possible that these models get their features from a data store that holds the customer data in a type of cache for quick access. Another possibility is that data is continuously loaded from source systems so that it can be stored in a historical overview and queried by real-time dashboards that contain predictive models or other forms of intensive data usage. For example, a retail organization might want to predict trends in their product sales based on previous years. This kind of data cannot be retrieved from the source systems directly since they only keep track of the current state. For each of these scenarios, a combination of data stores must be deployed, filled, and maintained in order to fulfill the business requirements.

Let's have a look at the requirements that need to be evaluated for an AI project. We'll start with a brief list and do a deep dive later in the chapter.

STORAGE REQUIREMENTS

It's crucial to keep track of the requirements of your solution in all phases of the project. Since most projects follow the agile methodology, it's not an option to just define the requirements at the start of the project and then "get to work."

The agile methodology requires team members to continuously reflect on the initial plan and requirements provided in the Deming cycle, as shown in the following figure:

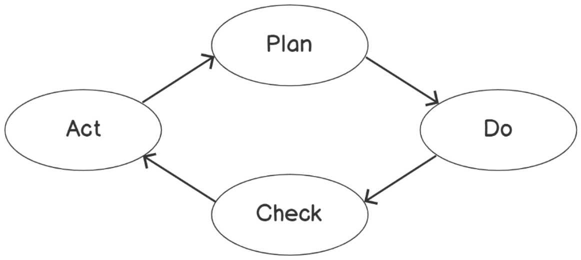
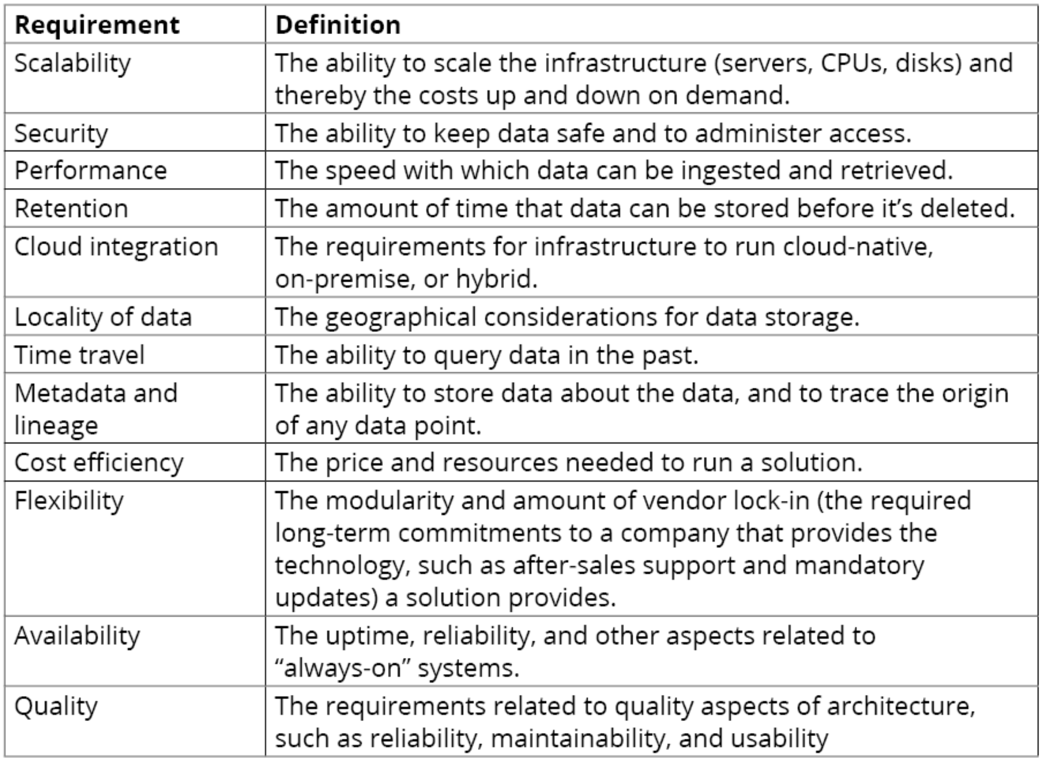


Figure 2.1: The Deming cycle

A list of requirements can be divided into functional and non-functional requirements. The **functional** requirements contain the user stories that explain how to interact with the system; these are not in the scope of this book since they are less technical and more concerned with UX design and customer journeys. The **non-functional** (or technical) requirements contain descriptions of the required workings of the system. The non-functional architecture requirements for an AI storage solution describe the technical aspects and have an impact on technology choices and their way of working. The major requirements of an AI system are as follows:



Requirement	Definition
Scalability	The ability to scale the infrastructure (servers, CPUs, disks) and thereby the costs up and down on demand.
Security	The ability to keep data safe and to administer access.
Performance	The speed with which data can be ingested and retrieved.
Retention	The amount of time that data can be stored before it's deleted.
Cloud integration	The requirements for infrastructure to run cloud-native, on-premise, or hybrid.
Locality of data	The geographical considerations for data storage.
Time travel	The ability to query data in the past.
Metadata and lineage	The ability to store data about the data, and to trace the origin of any data point.
Cost efficiency	The price and resources needed to run a solution.
Flexibility	The modularity and amount of vendor lock-in (the required long-term commitments to a company that provides the technology, such as after-sales support and mandatory updates) a solution provides.
Availability	The uptime, reliability, and other aspects related to "always-on" systems.
Quality	The requirements related to quality aspects of architecture, such as reliability, maintainability, and usability

Figure 2.2: Requirements for AI systems

Since this a very extensive list and some requirements are more important for a certain architectural layer than others, we will list the most important requirements per architecture layer. Before we start with that deep dive, we'll give a brief overview of the architecture of an AI system or data lake.

Throughout this chapter, we'll provide you with an example of a use case that helps translate the abstract concepts in the requirements for data storage to real-world, hands-on content. Although the sample is fictional, it's built on some common projects that we came across in real life. Therefore, the situation, target architecture, and requirements are quite realistic for an AI project.

A bank in the UK (let's say it's called PacktBank) wanted to upgrade its data storage systems to create a better environment for data scientists for AI-related projects. Currently, the data is spread out in various source systems, ranging from an old ERP system to on-premise Oracle databases, to a SaaS solution in the cloud. The new data environment (data lake) must be secure, accessible, high-performing, scalable, and easy to use. The target infrastructure is **Amazon Web Services** (**AWS**), but in the future, the company might switch to other cloud vendors or take a multi-cloud strategy; therefore, the software components should be vendor-agnostic if possible.

The Three Stages of Digital Data

It's important to realize that data storage comes in three stages:

- **At rest**: Data that is stored on a disk or in memory for long-term storage; for example, data on a hard disk or data in a database.
- **In motion**: Data that is transferred across a network from one system to another. Sometimes, this is also called *in transit*; for example, HTTP traffic on the internet, or data that comes from a database and is "on its way" to an application.
- **In use**: Data that is loaded in the RAM of an application for short-term usage. This data is only available in the context of the software that is loaded. It can be seen as a cache that is temporarily needed by the software that performs tasks on the data. The data is usually a copy of data at rest; for example, a piece of customer information (let's say, a changed home address) that has been pushed from a website to the server where an API processes the update.

These stages are important to keep in mind when reasoning about technology, security, scalability, and so on. We'll bring them up in this book in several places, so make sure that you understand the differences.

DATA LAYERS

An AI system consists of multiple data storage layers that are connected with **Extract**, **Transform**, and **Load** (**ETL**) or **Extract**, **Load**, and **Transform** (**ELT**) pipelines. Each separate storage solution has its own requirements, depending on the type of data that is stored and the usage pattern. The following figure shows this concept:

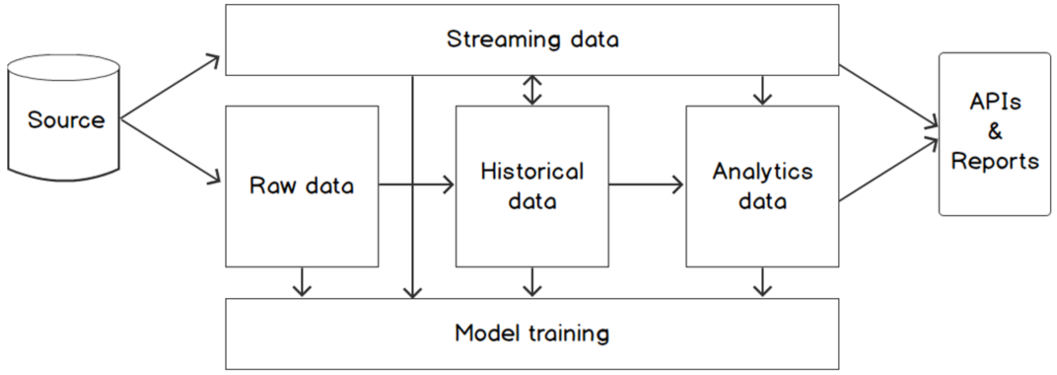


Figure 2.3: Conceptual overview of the data layers in a typical AI solution

From a high-level viewpoint, the backend (and thus, the storage systems) of an AI solution is split up into three parts or layers:

- **Raw data layer**: Contains copies of files from source systems. Also known as the staging area.
- **Historical data layer**: The core of a data-driven system, containing an overview of data from multiple source systems that have been gathered over time. By stacking the data rather than replacing or updating old values, history is preserved and time travel (being able to make queries over a data state in the past) is made possible in the data tables.
- **Analytics data layer**: A set of tools that are used to get access to the data in the historical data layer. This includes cache tables, views (virtual or materialized), queries, and so on.

These three layers contain the data in production. For model development and training, data can be offloaded into a special model development environment such as a DataBricks cluster or SageMaker instance. In that case, an extra layer can be added:

- **Model training layer**: A set of tools (databases, file stores, machine learning frameworks, and so on) that allows data scientists to build models and train them with massive amounts of data.

For scenarios where data is not being ingested by the system per batch but rather streamed in continuously, such as a system that processes sensory machine data from a factory, we must set up specific infrastructure and software. In those cases, we will use a new layer that takes the role of the raw data layer:

- **Streaming data layer**: An event bus that can store large amounts of continuously inflowing data streams, combined with a streaming data engine that is able to get data from the event bus in real time and analyze it. The streaming data engine can also read and write data to data stores in other layers, for example, to combine the real-time data from the event bus with historical data about customers from a historical data view.

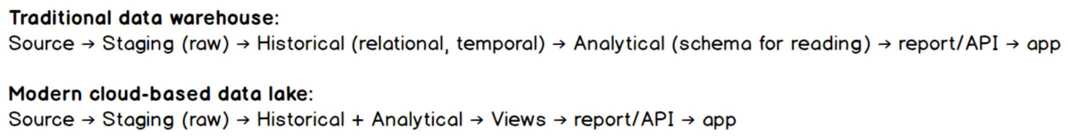
Depending on the requirements for data storage and analysis, for each layer, a different technology set can be picked. The data stores don't have to be physical file stores or databases. An in-memory database, graph database, or even a virtual view (just queries) can be considered as a proper data storage mechanism. Working with large datasets in complex machine learning algorithms requires special attention since the models to be trained require the storage and usage of big datasets, but for a relatively short period of time.

To summarize, the data layers of AI solutions are like many other data-driven architecture layers, with the addition of the model training layer and possibly the streaming data layer. But there is a bigger shift happening, namely the one from data warehouses to data lakes. We'll explore that paradigm shift in the next part of this chapter.

From Data Warehouse to Data Lake

The common architecture of a data processing system is shifting from traditional data warehouses that run on-premise toward modern data lakes in the cloud. This shift is being made by a new technology wave that started in the "big data" era with Hadoop in around 2006. Since then, more and more technology has arrived that makes it easier to store and process large datasets and to build models efficiently, making use of advanced concepts such as distributed data, in-memory storage, and **graphical processing units** (**GPU**s). Many organizations saw the opportunities of these new technologies and started migrating their report-driven data warehouses toward data lakes with the aim of becoming more predictive and to get more value out of their data.

Next to a technology shift, what we see is a move toward more virtual data layers. Since computing power has become cheap and parallel processing is now a common practice, it's possible to virtualize the analytics layer instead of storing the data on disk. The following figure illustrates this; while the patterns and layers are almost the same, the technology and approach differ:



Traditional data warehouse:
Source → Staging (raw) → Historical (relational, temporal) → Analytical (schema for reading) → report/API → app

Modern cloud-based data lake:
Source → Staging (raw) → Historical + Analytical → Views → report/API → app

Figure 2.4: Data pipelines in a data warehouse and a data lake

The following table highlights the similarities and differences between the data platforms:

	Data Warehouse	**Data Lake**
Infrastructure	On-premise	Cloud-native
Data pipeline	ETL	ELT
Raw data layer	Staging area	Staging area
Historical data layer	Relational data vault	Relational data vault
Analytics data layer	Analytical database	Virtual views on top of historical data
Model training layer	-	Machine learning cluster
Streaming data layer	-	Streaming data cluster and stream processing engine

Figure 2.5 Comparison between data warehouses and data lakes

NOTE

The preceding table has been written to highlight the differences between data warehouses and date lakes, but there is a "gray zone." There are many data solutions that don't fall into either of the extremes; for example, ETL pipelines that run in the cloud.

To understand data layers better, let's revisit our bank example. The architects of PacktBank started by defining a new data lake architecture that should be the data source for all AI projects. Since the bank did not foresee any streaming project on short notice, the focus was on batch; every source system had to upload a daily export to the data lake. This data was then transformed into a relational data vault with time-traveling possibilities. Finally, a set of views allowed quick access to the historical data for a set of use cases. The data scientists were building models on top of the data from these views, which was temporarily stored in a model development environment (Amazon SageMaker). Where needed, the data scientists could also get access to the raw data. All these access points were regulated with role-based access and were monitored extensively. The following figure shows the final architecture of the solution:

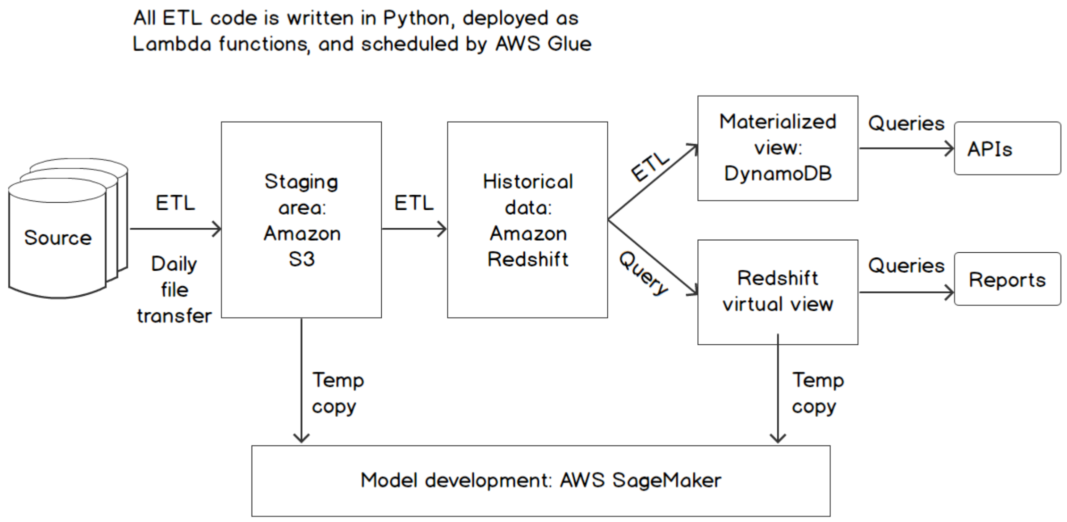

All ETL code is written in Python, deployed as Lambda functions, and scheduled by AWS Glue
Source
ETL
Daily file transfer
Staging area: Amazon S3
ETL
Historical data: Amazon Redshift
ETL
Query
Materialized view: DynamoDB
Redshift virtual view
Queries
Queries
APIs
Reports
Temp copy
Temp copy
Model development: AWS SageMaker

Figure 2.6: Sample data lake architecture

EXERCISE 2.01: DESIGNING A LAYERED ARCHITECTURE FOR AN AI SYSTEM

The purpose of this exercise is to get acquainted with the layered architecture that is common for large AI systems.

For this exercise, imagine that you must design a new system for a telecom organization; let's call it PacktTelecom. The company wants to analyze internet traffic, call data, and text messages on a daily basis and perform predictive analysis on the dataset to forecast the load on the network. The data itself is produced by the clients of the company, who are using their smartphones on the network. The company is not interested in the content of the traffic itself, but most of the information at the meta-level is interesting. The AI system is considered part of a new data lake, which will be created with the aim of supporting many similar use cases in the future. Data from multiple sources should be combined and analyzed in reports and made available to websites and mobile apps through a set of APIs.

Now, answer the following questions for this use case:

1. What is the data source of the use case? Which systems are producing data, and in which way are they sending data to the data lake?

   The prime data source is the smartphones of the clients. The smartphones are continuously connected to the network of the company and send their metadata to the core systems (for example, internet traffic and text messages). These core systems will send a daily batch of data to the data lake; if required, this can be made real-time streaming at a later stage.

2. Which data layers should you use for the use case? Is it a streaming infrastructure or a more traditional data warehousing scenario?

   A raw data layer stores the daily batches that are sent from the core systems. A historical data layer is used to build a historical overview per customer. An analytics layer is used to query the data in an efficient way. At a later stage, a streaming infrastructure can be realized to replace the daily batches.

3. What data preparation steps need to be done in the ETL process to get the raw data in shape so that it's useful to work with?

   The raw data needs to be cleaned; the content must be removed. Some metadata might have to be added from other data sources, for example, client information or sales data.

4. Are there any models that need to be trained? If so, which layer are they getting their data from?

   To forecast the network load, a machine learning model needs to be created that is trained on the daily data. This data is gathered from the historical data layer.

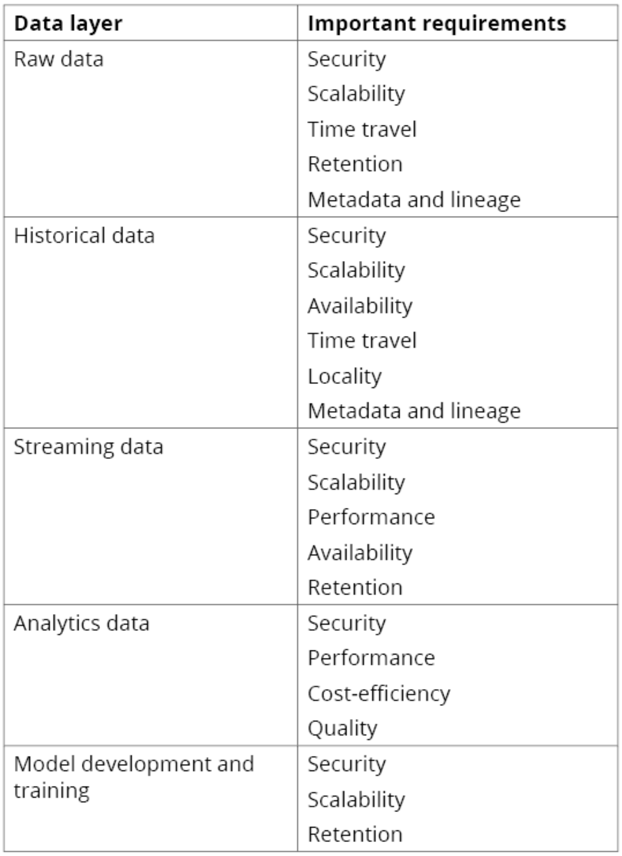
By completing this exercise, you have reasoned about the layers in an AI system and (partly) designed an architecture.

Requirements per Infrastructure Layer

Let's dive into the requirements for each part of a data solution. Depending on the actual requirements per layer, an architect can choose the technology options for the layer. We'll list the most important requirements per category here.

Some requirements apply to all data layers and can, therefore, be considered generic. For example, scalability and security are always important for a data-driven system. However, we've chosen to list them for each layer separately because each layer has many specific attention points for these generic requirements as well.

The following table highlights the most important requirements per data layer:



Data layer	Important requirements
Raw data	Security
	Scalability
	Time travel
	Retention
	Metadata and lineage
Historical data	Security
	Scalability
	Availability
	Time travel
	Locality
	Metadata and lineage
Streaming data	Security
	Scalability
	Performance
	Availability
	Retention
Analytics data	Security
	Performance
	Cost-efficiency
	Quality
Model development and training	Security
	Scalability
	Retention

Figure 2.7: Important requirements per data layer

Raw Data

The raw data layer contains the one-to-one copies of files from the source systems. The copies are stored to make sure that any data that arrives is preserved in its original form. After storing the raw data, some checks can be done to make sure that the data can be processed by the rest of the ETL pipeline, such as a checksum.

Security

We'll look at data security first. All modern software and data systems must be secure. By security requirements, we mean all aspects related to ensuring that the data in a system cannot be viewed or deleted by unauthorized people or systems. It entails identity and access management, role-based access, and data encryption.

Basic Protection

In any data project, security is a key requirement. The basic level of data protection is to require a username-password combination for anyone who can access the data: customers, developers, analysts, and so on. In all cases, the passwords should be evaluated against a strong password policy and must be changed on a regular basis. Passwords should never be stored in plain text; instead, they should always be in a salted and hashed form so that even system administrators cannot retrieve the actual passwords. The security levels themselves depend on the **Availability**, **Integrity**, and **Confidentiality** (**AIC**) rating, which we'll explain in the following paragraph. Suffice to say that highly secure data should not only be protected with a username and password. Multi-factor authentication is a way of adding a security layer by making use of a second or even third authentication method alongside password protection, such as an SMS message on your phone, a fingerprint ID, or a dedicated security token generator.

THE AIC RATING

Data security in organizations can be classified with the AIC rating (sometimes the CIA rating is also used, but this causes confusion with the abbreviation for the Central Intelligence Agency). Each data source and application should be categorized into three dimensions:

- **A = availability**: The level of protection against data loss in cases of system failure or upgrades. If data loss must be prevented at all times, for example, for payment transactions, the level is high. If data loss is annoying but not terrible, for example, spam emails, the level is low.
- **I = integrity**: The level of consistency and accuracy that the dataset must uphold during its life cycle. Some data might be removed or updated without any consequences; those datasets will receive a low integrity rating. However, if it's crucial that data records are preserved for a long time, for example, tax records, the rating will go up.
- **C = confidentiality**: The level of personal details of a person or company that is included in the dataset. If personal details such as names or addresses are stored, it's likely that the rating is high. The highest level of confidentiality is reserved for datasets that contain very private data, such as credit card numbers or passwords.

Each of these dimensions gets a rating from 1 to 3. Thus, a dataset with an AIC rating of 111 is considered to be less risky and vulnerable compared to a dataset with a rating of 333. You could argue that when data from a 111 system falls into the wrong hands (a hacker or competing organization), it's no big deal. However, when data from a 333 system ends up "on the street," you're in serious legal trouble and might even be out of business. When it comes to securing data, each category should imply certain measurements within your company. For example, any dataset with a C rating of 3 can only be accessed with multi-factor authentication.

ROLE-BASED ACCESS

In the raw data layer, all data files must be governed with **role-based access control** (**RBAC**) to ensure that no data falls into the wrong hands. Every principal (human or machine account) has one or more roles. Each role has one or more permissions to a database, file share, table, or other pieces of data. With the right permissions, files can be read by humans (for example, data scientists that require access to the raw data); write access is only available for software that imports the data from the source systems. Every file must be secured. In many cases, the actual security is inherited from a higher directory. The file structure proposed in *Figure 2.6: Sample data lake architecture*, allows security to be configured per source system or per period. It should also be possible to give data scientists access to a part of the data store temporarily, for example, to copy data to a machine learning environment. Modern security frameworks and identity and access management systems such as Microsoft Active Directory and Amazon AWS **Identity and Access Management** (**IAM**) have these options available. The following figure shows these:

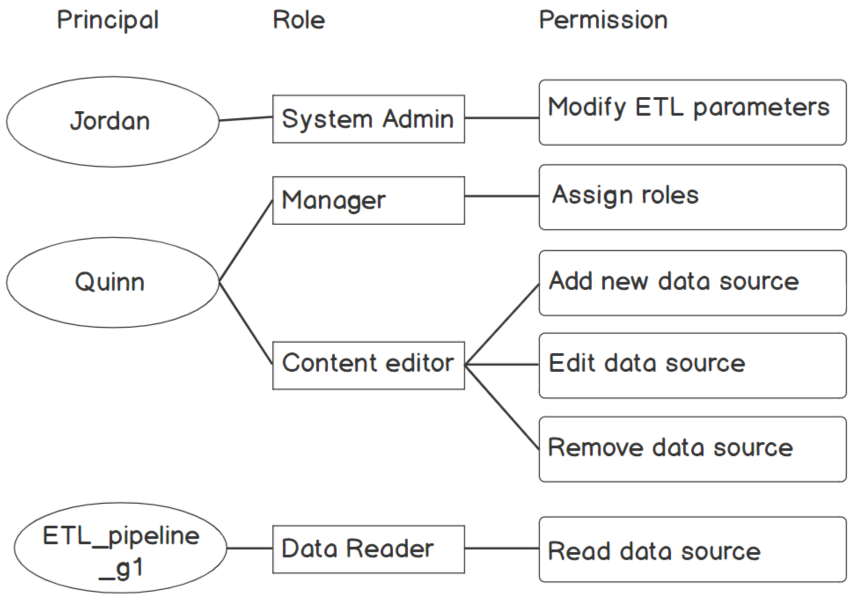


Figure 2.8: Simplified sample of roles and permissions of a data importing system

The preceding figure is a highly simplified role-based access diagram that has been sketched to illustrate the relations between principals, roles, and permissions. In this example, **`Jordan`** has the role of **`System Admin`** and is therefore allowed to modify the parameters of an ETL pipeline. However, he cannot read the actual contents of the data sources since he has no permissions. **`Quinn`** has two roles: **`Manager`** and **`Content Editor`**. She can assign roles to other principals and add, edit, and remove data sources. **`ETL_pipeline_g1`** is a non-human principal, namely an account that a piece of software uses to access data and execute tasks. It has the role of **`Data Reader`** and therefore has permission to read the data from the data sources.

Encryption

Data at rest (stored on disk) and in motion (transferred across a network) should be encrypted to prevent third parties and hackers from accessing it. By encrypting data, a hacker who intercepts the data or reads it straight from disk still cannot get to its content; it will just be a scrambled array of characters. To read the data, you must be in possession of a private key that has a relation to the public key that the data was encrypted with (with asymmetric encryption, which is by far the most popular mechanism). These private keys must, therefore, be kept secure at all times, for example, in a special purpose key store. All modern cloud infrastructure providers have such a key store as a service in their offerings.

There are many types of encryption possible. **Advanced Encryption Standard** (**AES**) and **Rivest-Shamir-Adleman** (**RSA**) are two examples of popular asymmetric encryption algorithms. These can both be used to encrypt data at rest and in motion, although some performance considerations might apply; RSA is a bit slower than AES. What's more important is to choose the size of the keys; the more bits, the harder it is to break the encryption with a brute-force attack. Currently, 256 bits is considered to be a safe key size.

Keys should never be stored in places with more or less open access, for example, sticky notes or Git repositories. It's good practice to rotate your keys, which means that they alter after a certain period (say, a month). This makes sure that even if keys are accidentally stored in a public place, they can only be used for a limited period of time.

At the core of data security are four basic principles:

- Security starts with basic protection, such as strong and rotating passwords. All users are registered in a central identity and access management system.
- All access to data is regulated with permissions. Permissions can be attached to roles, and roles can be assigned to users. This is called role-based access.
- The data security measurements are related to the AIC rating of a dataset. A higher rating indicates that more security controls should be put into place.
- Data at rest and in motion can be encrypted to protect it from intruders.

Let's understand this better by going through the next exercise.

Exercise 2.02: Defining the Security Requirements for Storing Raw Data

For this exercise, imagine that you are creating a new data environment for an ambulance control room. The goal of the system is to gather as much useful information as possible from government and open data sources in order to direct ambulances on their way to a 911 call once it arrives. The core data sources that must be stored are the 911 calls; this is combined with maps data, traffic information, local news from the internet, and other sources. It's apparent that such systems are prone to hacking and a wrong/fake call could lead to medical mistakes or the late arrival of emergency personnel.

In this exercise, you will create a security plan for the ambulance control room. The aim of this exercise is to become familiar with the security requirements of a system where data protection plays an important role.

Now, answer the following questions for this use case:

1. Consider the data source of your application. Who is the owner of the data? Where is the data coming from?

   The prime data source is the 911 calls that come from the people who need help. The call data is owned by the person who makes the call.

2. What is a potential security threat? A hacker on the internet? A malicious employee of your company? A physical attack on your data center?

   Potential security threats are hackers on the internet, fake phone callers, employees who might turn against the company, physical attackers on the data center, terrorists, and many more.

3. Try to define the AIC rating of the dataset. What are the levels (from 1 to 3) for the availability, integrity, and consistency of the data?

   The phone calls have an AIC rating of 233. The availability is reasonably high but retrieving the data, in retrospect, is not as important as being able to respond to the calls once they arrive; thus, the overall availability is 2. The infrastructure for making the calls has an availability rating of 3. The integrity rating is 3 since the ambulance control room must be able to rely on the data; the location, time, and call quality are all very important aspects of the data. The confidentiality rating is also 3 since the calls themselves will contain many privacy-related details and confidential information.

4. Regarding the AIC rating, which measurements should you take to secure the data? What kind of identity and access management should you put in place? What kind of data encryption will you use? Consider the roles and permissions for accessing the data, as well as password regulations and multi-factor authentication.

   Considering the high integrity and confidentiality ratings, the security around the call data must be very good. The data should only be accessed by registered and authorized personnel of the control room, who have been given access by a senior manager. The access controls must be audited on a regular basis. Two-factor authentication, a strong password policy, and encryption of all data at rest and in motion must be put in place in order to minimize the risk of security breaches and hacks from outside.

By completing this exercise, you have created a security plan for a demanding system. This helps when setting the requirements for systems in your own organization.

SCALABILITY

Scalability is the ability of a data store to increase in size. Elasticity is the ability of a system to grow and shrink on demand, depending on the need at hand. For the sake of simplicity, we address both scalability and elasticity under one requirement: scalability.

In traditional data warehousing projects, a retention policy was very important in the raw data layer since it prevented the disks from getting full. In modern AI projects, what we see is that it's best to keep the raw data for as long as possible since (file) storage is cheap, scalable, and available in the cloud. Moreover, the raw data often provides the best source for model training in a machine learning environment, so it's valuable to give data scientists access to many historical data files.

To cater for storing this much data on such a large scale, a modern file store such as Amazon S3 or Microsoft Azure ADLS Gen2 is ideally suited. These cloud-based services can be seen as the next generation of Hadoop file stores, where massive parallel file storage is made easily available to its consumers. For an on-premise solution, Hadoop HDFS is still a good solution.

Using the same example of PacktBank, the new data lake for AI must start small but soon scale to incorporate many data sources of the bank.

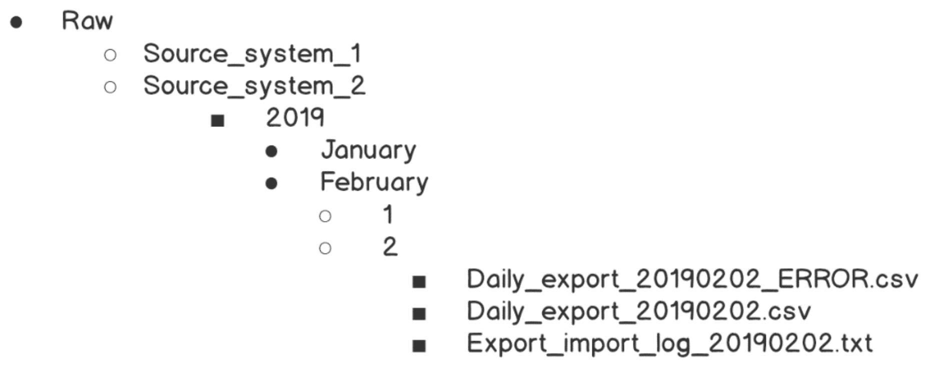
The architects of PacktBank defined the following set of requirements for the new system:

- The data store should start very small since we will first do a proof-of-concept with only test data. The initial dataset is about 100 MB in size.
- The data store should be able to expand rapidly toward a size where all the data from hundreds of core systems will be stored. The expected target size is 20 TB.
- There will be a retention policy forced on some parts of the data since privacy regulations enforce that certain sensitive data be removed after 7 years. The data store should be able to shrink back to a smaller size (~15 TB) if needed, and the costs associated with the data store should follow proportionally.

Time Travel

For many organizations, it is important to be able to query data in the past. It's very valuable and is often required by laws or regulations to be able to answer questions such as "how many customers were in possession of product 3019 one month ago?" or "which employees had access to document 9201 on 14 March 2018?". This ability is called **time travel**, and it can be embedded in data storage systems.

Raw data must be stored in a way so that its origins and time of storage are apparent. Many companies choose to create a directory structure that reflects the daily or hourly import schedule, like so:



- Raw
    - Source_system_1
    - Source_system_2
        - 2019
            - January
            - February
                - 1
                - 2
                    - Daily_export_20190202_ERROR.csv
                    - Daily_export_20190202.csv
                    - Export_import_log_20190202.txt

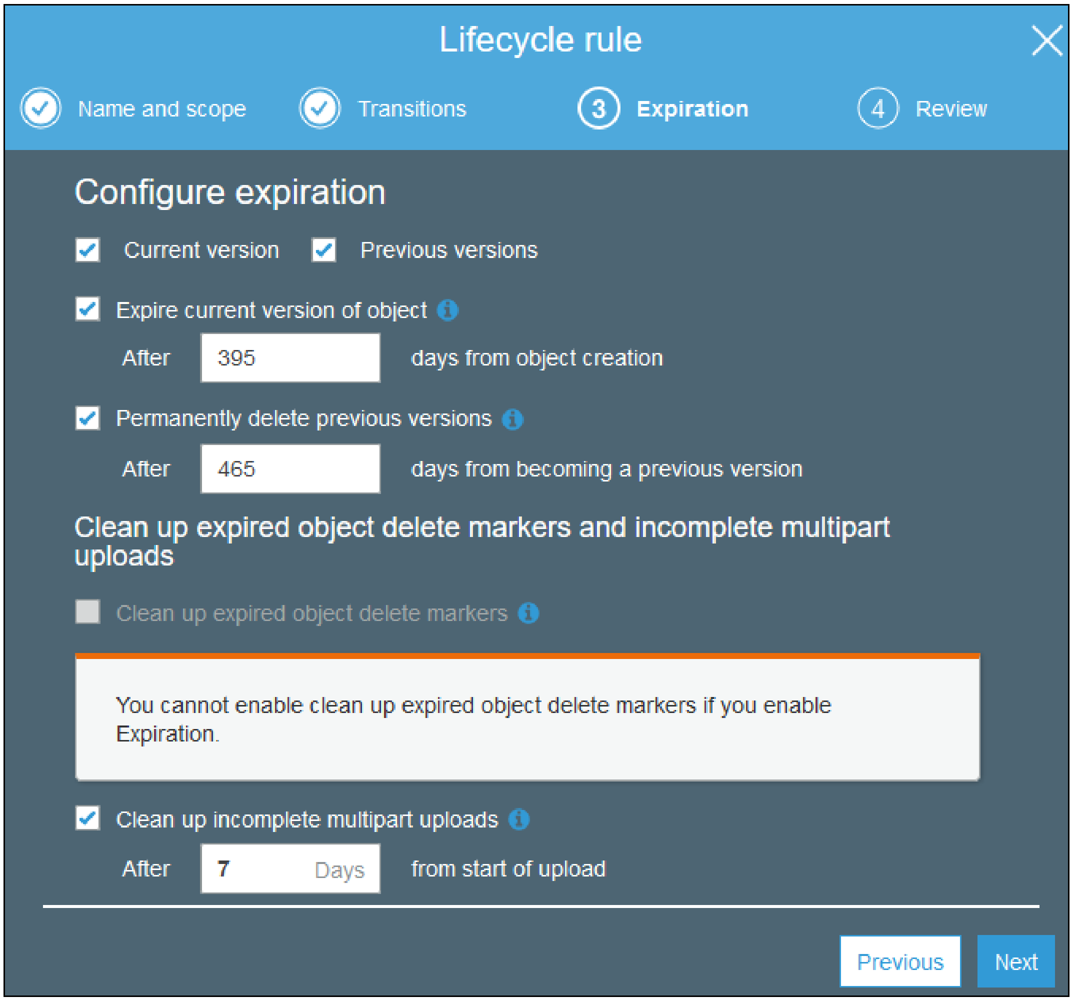
Figure 2.9: Example of a directory structure for raw data files

In the preceding figure, a file that arrives on a certain date and time gets placed in a directory that reflects its arrival date. We can see that on February 2, 2019, a daily export was stored. There is also a file containing `ERROR` which possibly indicates a failed or incomplete import. The full import log is stored in a text file in the same folder. By storing the raw data in this structure, it's very easy for an administrator to ask questions about the source data in the past; all they must do is browse to the right directory on the filesystem.

Retention

Data retention requirements define in what way data is stored to meet laws and regulations or to preserve disk space by deleting old files and data records. Since it's often convenient and useful in a modern AI system to keep storing all the data (after all, data scientists are data-hungry), a retention policy is not always necessary from a scalability perspective. As we saw when exploring the scalability requirement, many data stores can store massive amounts of data cheaply. However, a retention requirement (and therefore, a policy) might be needed because of laws and regulations. For example, in the EU's GDPR regulations, it's stated that data must be stored "for the shortest time possible." Some laws and regulations are specific for industries, for example, call data and metadata in a telecom system may only be stored for 7 years by a telecom provider.

To cater to retention requirements, the infrastructure and software of your data lake should have a means of removing and/or scrambling/anonymizing data periodically. In modern file stores and databases, policies can be defined in the tool itself and the tool automatically implements the retention mechanisms. For example, Amazon S3 supports lifecycle policies in which data owners can specify what should happen with the data over time, as shown in the following figure:



Configure expiration

☑ Current version ☑ Previous versions

☑ Expire current version of object

After 395 days from object creation

☑ Permanently delete previous versions

After 465 days from becoming a previous version

Clean up expired object delete markers and incomplete multipart uploads

☐ Clean up expired object delete markers

You cannot enable clean up expired object delete markers if you enable Expiration.

☑ Clean up incomplete multipart uploads

After 7 Days from start of upload

Figure 2.10: Retention rules in Amazon S3

When storing data in an on-premise data store that does not support retention (for example, MySQL or a regular file share), you'll have to write code yourself that periodically runs through all your data and deletes it, depending on the parameters that have been defined for your policy.

Metadata and Lineage

Metadata is data about data. If we store a file on a disk, the actual data is the contents of the file. The metadata is its name, location, size, owner, history, origin, and many other properties. If metadata is correctly and consistently registered for each dataset (a file or database record), this can be used to track and trace data across an organization. This tracking and tracing is called **lineage**, and it can be mostly automated with modern tooling.

If the requirements for metadata management and lineage requirements are in place, every data point in the pipeline must be traced back to its source if required. This can be requested in audits or for internal data checks. For the raw data layer, it implies that for every data source file, we store a set of metadata, including the following:

- **Filename**: The name of the file
- **Origin**: The location or source system where the data comes from
- **Date**: The timestamp when the data entered the data layer
- **Owner**: The data owner who is responsible for the file's contents
- **Size**: The file size

For data streams, we store the same attributes as metadata but they are translated to the streaming world; for example, the stream name instead of a filename. In *Chapter 3*, *Data Preparation*, we'll discuss lineage in detail and provide an exercise for building lineage into ETL jobs.

A further extension to the security model and metadata-driven lineage, once in place, is **consent management**. Based on the metadata of the raw data files, a data owner can select which roles or individuals have access to its files. In that way, it's possible to create a data-sharing environment where each data owner can take responsibility for the availability of their own data.

Now, we have discussed the main requirements for the raw data layer: security, scalability, time travel, retention, and metadata. Keep in mind that these topics are important for any data storage layer and technology. In the next section, we will explore these requirements for the historical data layer and add availability as an important requirement that is most apparent for the technology and data in that layer.

Historical Data

The historical data layer contains data stores that hold all data from a certain point in the past (for example, the start of the company) up until now. In most cases, this data is considered to be important to run a business, and in some cases, even vital for its existence. For example, the historical data layer of a newspaper agency contains sources, reference material, interviews, media footage, and so on, all of which were used to publish news articles. Data is stored in blobs, file shares, and relational tables, often with primary and foreign keys (enforced by the infrastructure or in software). The data can be modeled to a standard such as a data vault to preserve historical information. This data layer is responsible for keeping the truth, which means it is highly regulated and governed. Any data that is inserted into one of the tables in this layer has gone through several checks, and metadata is stored next to the actual data to keep track of the history and manage security.

Security

In general, the same requirements that apply to the raw data store also apply to the historical data layer. Whereas the raw data layer dealt with files, the historical data layer has tables to protect. But that is often not enough granularity since a lot of data can be combined in tables. For example, a company that provides consultancy services to multiple customers could have a table with address information that contains records from these companies. But for the sake of privacy and secrecy, not all account managers in the consultancy organization may have access to each client; they should only see the information of the clients that they are working for directly. For these kinds of cases, it must be possible to apply row-level or column-level security.

Row-Level Security

When setting up tables in a multi-tenant way, containing data from multiple owners, it's necessary to administer the owner per record. Modern databases and data warehouse systems such as Azure Synapse can then assign role-based access security controls to the tables per row; for example, "people that have role A have read-only access to all records where the data owner is O."

Column-Level Security

In a similar way, security can be arranged per column in modern columnar NoSQL databases. It might be beneficial to add columns to a table for a specific client or data owner. In those cases, access to the columns can be arranged with similar role-based access; for example, "people that have role B have read and write access to all data in columns Y and Z."

Scalability

The amount of data in the historical data layer will keep on growing since fresh data will arrive every day and not all data will be part of a retention plan. Business users will also regard the historical data as highly valuable since the information there can be used to train models and generally compare situations of the organization. Therefore, it's crucial to pick technology that can scale. Modern data warehouses in the cloud all cater to scalability and elasticity, for example, Amazon Redshift and Snowflake. The scalability of data stores on-premise is more limited, constrained by your own data center. However, for many organizations, the requirements to scale might be perfectly met by an on-premise infrastructure. For example, local government organizations often deal with complex data from many sources, but the total size of a data lake usually does not surpass the 1 TB mark. In these kinds of cases, setting up a solid infrastructure that can hold this data is perhaps a better choice than to put all the data in the public cloud.

Availability

A system or datastore is considered to be reliable and highly available if there is a guarantee that a system keeps on running and no data is lost in the lifespan of the system, even during outages or failures. Usually, this problem is solved by distributing data across an infrastructure cluster (separated in servers/nodes or geographical regions) as soon as it enters the system. In the case of a crash or other malfunction, the data is backed up in multiple locations that seamlessly take over from the main database.

The historical data layer is the heart of the modern data lake and as such, it is primarily concerned with storing data for an (in principle) indefinite duration. Therefore, reliability and robustness are key to selecting the technology components for this layer. Technology components are selected based on the maximum downtime of a system. So, first, let's look into calculating the availability percentage and from there, decide on the technology.

The availability of a system is usually expressed in a percentage, which denotes time (in hours) that the system is up and running as a function of its lifespan. For example, a system with an availability of 90% is expected to be online 168 * 0.9 = 151 hours of a full week of 168 hours, which means that there are 168 - 151 = 17 hours in which the system can be taken offline for maintenance. Unfortunately, an availability of 90% is very poor nowadays.

The following table gives an overview of availability as a percentage, and the amount of downtime related to the percentage:

Availability	**Downtime per week**	**Downtime per year**
90%	17 hours	37 days
95%	8 hours	18 days
99%	2 hours	4 days
99.9%	10 minutes	9 hours
99.99%	1 minute	53 minutes
99.999%	26 seconds	5 minutes
99.9999%	0.5 seconds	32 seconds

Figure 2.11: Availability percentages explained in downtime

It's possible to calculate the availability of a system using the following formula:

$$\text{Availability} = \frac{\text{Available hours} - \text{Downtime}}{\text{Available hours}} \times 100\%$$

Figure 2.12: Availability formula

For example, if a data store was down for maintenance for 3 hours with available hours being 5040, the availability of that system was as follows:

$$\frac{5040 - 3}{5040} \times 100\% = 99.94\%$$

Figure 2.13: Availability calculation

Let's understand this better with an exercise.

Exercise 2.03: Analyzing the Availability of a Data Store

For this exercise, imagine that you work as an operations engineer for TV broadcasting company PacktNet. The core system of the company must be available so that clients can binge-watch their favorite series at all times. The system, therefore, has received an availability rating of 99.99%.

The aim of this exercise is to become familiar with the availability formula and to practice it in a real-world scenario.

Now, answer the following questions for this use case:

1. Is it allowed to bring the system down for 5 minutes per month for maintenance?

   An availability of 99.99% means that the system can be down for 1 minute per week or about 4 minutes per month. So, a downtime of 5 minutes per month is *not* allowed.

2. In the previous year, there were only a few minor incidents and almost no scheduled maintenance. The system was offline for 1 hour in total. What was the availability during that year?

   There are 365 x 24 = 8760 hours in a year. The availability in the previous year was $\frac{8760 - 1}{8760} \times 100\% = 99.99\%$ (rounded to two decimals).

By completing this exercise, you have successfully calculated the availability of a data store. This is an important requirement for any system.

AVAILABILITY CONSEQUENCES

Once the availability requirements of a system have been determined, a matching infrastructure should be chosen. There are many different data stores (file shares and databases) that all have their own availability percentage. The following table contains the percentages for a selection of popular cloud-based data stores. Note that some services offer a stepped approach, where more uptime costs more. Also, note that the availability can be drastically increased if the data is parallelized across different data centers and different regions:

Service	Availability
Amazon S3	99.99%
Amazon EC2	99.99%
Azure Cosmos DB	99.99%
Azure Databricks	99.95%

Figure 2.14: Availability of a few popular data stores in the cloud

Some cloud services offer high availability but don't express this in a percentage. For example, the documentation of Amazon SageMaker states that it is designed for high availability and runs on Amazon's scalable and robust infrastructure (with availability zones, data replication, and so on), but does not give a guaranteed maximum downtime percentage.

When working with data on-premise, the calculation differs a bit. The correct way to calculate the entire availability of a system is to multiply the availability of the infrastructure (servers) by the availability of the software.

Some considerations when writing down the availability requirements for data storage are as follows:

- There is planned downtime versus unplanned downtime; planned downtime is for scheduled upgrades and other maintenance on the system that must be done on a regular basis. Planned downtime counts as non-available, but obviously, it can be controlled, managed, and communicated better (for example, an email to all users of the system) than unplanned downtime, which is when there are unexpected crashes of the system.
- Once availability becomes very high, it's often described in the "number of nines." This indicates the number of nines in the percentage; 99.99% is four nines availability.
- A system that is 100% available might still be unusable, for example, if the performance of the interface is very slow. So, keep in mind that the availability percentage does not express the entire scope of the reliability and availability of the system; it's just a helpful measurement.
- Next to measuring availability, it can be even more important to measure data loss. If a system is down for 1 hour, but all the data that was entered into the system during that period is lost, there could be major implications. Therefore, it's good practice to focus on the distribution (and thereby redundancy) of data; as soon as data is stored, it should be replicated across multiple nodes. Furthermore, there should be backups in place for emergency scenarios where all the nodes fail.
- High availability always comes at a cost. There is a trade-off between the "number of nines" and the cost of development, infrastructure, and maintenance. This is ultimately a business decision; how mission-critical is the data and the system, and at what price?

TIME TRAVEL

One of the key requirements of the historical data layer is the ability to "time travel." That means it should be possible to retrieve the status of a record or table from any moment in the past, providing the data was there. Tables that have this ability are called **temporal tables**. This can be achieved by applying a data model that allows time travel, for example, a data vault. Essentially, these data models are append-only; no updates or deletes are allowed. With every new record that enters the system, a "valid from" and optionally "valid to" timestamp is set. Let's understand this better with an example.

The following table contains an example of a table with company addresses. The data format is according to the "facts and dimensions" model, where a relatively static dimension (for example, company, person, or object) is surrounded by changing facts (for example, address, purchase, call). The company with ID 51 recently changed address from Dallas to Seattle, so a new record was added (record ID: 4). The old address of the same company is still preserved in the table (record ID: 3). So, now, we have two address rows for the same company, which is fine since only one can be valid at any given moment:

ID	CompanyId	Address	PostalCode	City	ValidFrom
2	45	Main St. 34	10982	New York	2013-10-15 12:18
3	51	Old St. 41	92028	Dallas	2011-02-18
4	51	6th Avenue 1002	87108	Seattle	2019-11-01

Figure 2.15: Example of a table that preserves historical data and allows time travel

Suppose the government needs to have a report with a list of offices that have shut down in Dallas and their new locations. In such cases, time travel is a very important requirement. A query that retrieves all these addresses (current and historical) of the company is as follows:

```
SELECT * FROM Addresses WHERE CompanyId = 51;
```

A query that retrieves the current address of a company is as follows:

```
SELECT TOP 1 * FROM Addresses WHERE CompanyId = 51 ORDER BY ValidFrom
DESC;
```

The same requirement can also be fulfilled by using a **`ValidTo`** column; if that is empty (**`NULL`**), the record is the most actual one. The downside of this approach is that it requires updates to a table, not just inserts, so the ETL code can become more complex. Using both **`ValidFrom`** and **`ValidTo`** is also possible and provides better querying options but adds some complexity and requires the **`insert`** and **`update`** statements to be in sync.

If time travel is a key feature for your use case, for example, a healthcare system that needs to keep track of all the medicine that was provided to patients, you might consider a database where these kinds of timestamps are a native element for all data entry; for example, Snowflake.

Another way to achieve the possibility to time-travel your data is with a mechanism called **event sourcing**. This is a relatively new method of storing data, where each change that's made to the data is stored separately rather than as a result of the change. For example, an **`UPDATE`** statement in a traditional database results in the overwriting of a record. With event sourcing, an **`UPDATE`** statement would not alter the record itself but rather add a new record to the table, along with information about the change. In this way, there is automatically a trail of events that leads from the original record to the latest one. This trail can be quite long and is therefore mostly used in the historical data layer, not in the analytics layer of an AI application. To get the latest record, all events must be replayed and calculated over the original event. This can include events that cancel each other out; for example, the combination of an **`INSERT`** and **`DELETE`** statement.

The data of PacktBank has great value if it can be queried from a historical perspective. Since many source systems only store the "present" situation, it's important that the new data lake preserves the data's history. To that extent, the bank chooses to create a historical data warehouse that AI systems can benefit from. On a daily basis, the current state of the source systems is appended to the database tables, which are arranged in a data vault model. Data scientists and analysts can now request access to perform time-series analysis, for example, of the earnings and spending of a customer to forecast their ability to afford a loan.

Locality of Data

When data is stored in an international organization, it's important to think about the physical location of data storage. We are used to systems that respond instantaneously and smoothly; a lag of 1 second when visiting a web page is already considered to be annoying. When data is stored in one continent, it can take some time (up to a few seconds) to reach another continent, due to the time it takes on the network. This kind of delay (latency) is not acceptable to clients who are working with the data, for example, the visitors of websites. Therefore, data must be stored close to the end users to minimize the amount of network distance. Furthermore, there might be laws and regulations that constrain the possible physical locations of data storage. For example, a government organization might require that all data is stored in its own country.

Most cloud-based data storage services offer the option to store data in specific regions or locations. Amazon and Microsoft provide geographical regions across the globe for their cloud offerings (AWS and Azure), in which customers can choose to put their data. If needed, there is a guarantee that the data will not leave the chosen region. For the sake of availability and robustness, it's best to distribute data across regions.

Metadata and Lineage

Metadata management in the historical data layer is important but quite difficult to realize. For every table and record (row), there should be a metadata entry that lists the origin, timestamp, owner, transformations, and so on. Usually, this metadata is stored in a separate table or database, or in a dedicated metadata management system. Since data is usually entered into the database via an ETL process, there is a big responsibility for the ETL tools to keep updating the metadata. Once in place, the metadata repository will be a valuable asset in the data lake, since it allows questions such as the following to be answered:

- What are the sources of the aggregated calculation in my report?
- At what date and time were the records from source system X last updated?
- How often on average is the data in the employee table refreshed?

In this section, we have looked at the most important requirements for the historical data layer of an AI system. We have looked at security, scalability, availability, time travel, and the locality of data. In the next section, we'll look at the typical requirements that should be considered when working with data streams.

Streaming Data

The requirements for a streaming data layer are different from a batch-oriented data lake. Firstly, the time dimension plays a crucial role. Any event data that enters the message bus as a stream must be timestamped. Secondly, performance and latency are more important since it must be certain that data can be processed in due time. Thirdly, the way that analytics and machine learning are applied differs; while the data is being streamed in, the system must analyze it in near-real-time. In general, streaming data software relies more on computing power than storage space; processing speed, low latency, and high throughput are key. Nevertheless, the storage requirements that are in place for a streaming data system are worth considering and are a bit different from "static" batch-driven applications.

Security

A typical streaming datastore is separated into **topics**. A topic is named as such in the popular streaming data store Kafka. These can be considered as being like tables in a traditional database. In other frameworks, they might have different names; for example, in the cloud-based Amazon Kinesis data store, the topics are called shards. For the sake of clarity, we will continue to call them topics in the remainder of this chapter. Topics can be secured by administering role-based access. For example, in a bank, there are many kinds of streaming data: financial transactions, page visits of clients, stock market traders, and others. Each of these kinds of data gets its own topic in a streaming data store, with its own data format, security, access role, and availability rating.

Data at rest (stored in an event bus) and in motion (incoming and outgoing traffic) should be encrypted with the mechanisms we explained in the *Security* subsection within the *Raw Data* section.

Performance

When analyzing data streams, it's crucial to select technology and write code that can handle thousands or even millions of records per second. For example, systems that work with **Internet of Things** (**IoT**) or sensory data must handle massive amounts of events. There are two important performance requirements for these systems:

- The amount of data (the number of events per second and bytes per second) that the event bus and stream processing engine is able to handle; this is the base figure that tells us whether there is a risk of overloading the system. In frameworks such as Kafka, Spark, and Flink, this is scalable; roughly speaking, just add more hardware to process more events.
- The amount of data that the software runs as jobs on the stream processing engine that it is able to handle. The software should run fast enough to be able to process all events per time window before a new calculation is required. Therefore, the software that performs the aggregations and eventually more complex event processing, such as machine learning, must be optimized and carefully tested.

Availability

When a streaming engine crashes or must be taken offline for maintenance, it should only temporarily stop processing the never-ending stream of data and reprocess any events it missed. Also, there must be a guarantee that no data is lost; even when the system is down temporarily, data should be replayed into the streaming engine to make sure that all the events go through the system. To that extent, modern streaming engines such as Spark and Flink offer **savepoints** and **checkpoints**. These are backups of the in-memory state of the streaming engine to disk. If there is a crash or scheduled maintenance, the latest checkpoint or `savepoint` is reloaded into memory from disk, and the data isn't lost. In combination with Kafka offsets (the latest point that is read from a data source topic by a consumer), it's clear that all data is replayed if necessary.

There are three main semantics when configuring availability and preventing the data loss of a streaming system:

- **At-least-once**: The guarantee that any event is processed at least once, but it's possible that one event goes through the system multiple times in the event of failures.
- **At-most-once**: The guarantee that any event is never processed more than once.
- **Exactly-once**: The guarantee that any event is processed exactly once by the streaming engine; this can only be accomplished with tight integration between the event bus (using offsets) and the streaming engine (using checkpoints and **`savepoints`**).

For example, one of the requirements of PacktBank is to analyze the financial transactions and online user activity of its customers in real time. Use cases that should be supported include real-time fraud detection and customer support (based on clickstreams and actions in the mobile app). A streaming engine was designed and developed with state-of-the-art technology, including Apache Kafka and Apache Flink. The requirement for the availability of the system was clear: in the case of maintenance or bugs, the system should not lose any data, since all transactions have to be processed. It's better to have a little delay and to keep customers waiting for a few minutes more than to miss fraudulent transactions altogether. Therefore, the architecture of the system was designed with an at-least-once guarantee of data availability. For every streaming job that handles the customer event data, an offset in Kafka keeps track of the latest data that has been read. Once data has entered a job in Flink, it's backed up in **`savepoints`** and checkpoints to make sure that no data has been lost.

Retention

In a streaming data system, the data is used when it's "fresh." Old data is only used for training models and generating historical (aggregated) reports. Therefore, the retention of a streaming data topic in an event bus can usually be set to a few days or a few weeks at the most. This saves storage space and other resources. When setting this requirement, think carefully about the aggregation step; perhaps it's useful to store the averages per hour or the results of the window calculations in your stream. As a typical example, the clickstream data of an online news website is only valuable when it's less than 1 day old. After all, news that's 1 day old is not very relevant anymore and the customers who have read the articles have already moved on!.

Retention is also related to the amount of data that is expected. Sometimes, the number of events is just too many to be stored for a long period of time. When reasoning about data retention, it's advised to estimate the average and peak load of a system first. This can be done by multiplying the number of concurrent data sources that produce event data with the number of events that are being produced. For example, a payment processing engine at PacktBank has 1 million users in total, which all make 3 payments per day on average with a peak of 20 per day. The average load of the system is 1 million x 3 = 3 million payments per day, which is about 2,000 payments per minute or 35 per second. At peak times, this can rise to 250 or so per second. A streaming data store that handles these events should be able to store these amounts of data and set a retention period in such a way that the disks will not become full.

Exercise 2.04: Setting the Requirements for Data Retention

For this exercise, imagine that you are building a real-time marketing engine for an online clothing distribution company. Based on the online behavior of (potential) customers, you want to create advertisements and personalized offerings to increase your sales. You will get real-time clickstreams (page visits) as your prime event data source. On average, 200,000 individuals visit your website per day. They spend about 20 minutes on your site and usually visit the home page, their favorites, about 75 clothing items, and their shopping basket.

The aim of this exercise is to become familiar with the concept of data retention.

Now, answer the following questions for this use case:

1. What is a reasonable number of events that the system should be able to handle per minute? Are there peak times, and how would you handle them?

   A quick estimation: 200,000 visits per day is 8,333 per hour on average. But we expect the evenings to be much busier than the mornings and nights, so we aim for a load of 25,000 concurrent users. They visit at least 75 items plus some other pages, so a reasonable clickstream size is 100-page visits per 20 minutes, which is 5 visits per minute. So, the total load is 5 x 25,000 = 125, 000 events per minute.

2. What retention policy would you attach to the event data? How long would the events be useful for in their raw form? Would you still require a report or other form of historical insight into the old data?

> The page visits will probably be valuable data for a week or so. After a week, other items will have sparked the interest of the clients, so the real-time information about the old events won't be as valuable anymore. Of course, this depends on the frequency of visits; if someone only logs in once per month, the historical data might be valuable over a longer period of time.

In this section, we discussed the typical requirements for streaming data storage: security, performance, availability, and retention. In the next section, we'll explore the requirements for the analytics layer, where data is stored for quick access in APIs and reports.

Analytics Data

The responsibility of the analytics layer of an AI system is to make data fast and available for machine learning models, queries, and so on. This can be achieved by caching data efficiently or by virtualizing views and queries where needed to materialize these views.

Performance

The data needs to be quickly available for ad hoc queries, reports, machine learning models, and so on. Therefore, the data schema that is chosen should reflect a "schema-on-read" pattern rather than a "schema-on-write" one. When caching data, it can be very efficient to store the data in a columnar **`NoSQL`** database for fast access. This would mean the duplication of data in many cases, but that's all right since the analytics layer is not responsible for maintaining "one version of the truth." We call these caches **data marts**. They are usually specific for one goal, for example, retrieving the sales data of the last month.

In modern data lakes, the entire analytics layer can be virtualized so that it just consists of queries and source code. When doing so, regular performance testing should be done to make sure that these queries are still delivering data as quickly as expected. Also, monitoring is crucial since queries may be being used in inappropriate ways, for example, setting parameters to such values (dates in **`WHERE`** clauses that span too many days) that the entire system becomes slow to respond. It's possible to set maximum durations for queries.

Cost-Efficiency

The queries that run in the analytics layer can become very resource-intensive and keep running for hours. Any compute action in a cloud-based environment costs money, so it's crucial to keep the queries under control and to limit the amount of resources that are spent. A few ways to make the environment more cost-effective are as follows:

- Limit the maximum duration of queries to (for example) 10 minutes.
- Develop more specific queries for reports, APIs, and so on, rather than having a few parameterized queries that are "one size fits all." Large, complicated queries and views are more difficult to maintain, debug, and tune.
- Apply good database practices to the tables where possible: indexes, partitioning, and so on.
- Analyze the usage of the data and create caches and/or materialized views for the most commonly used queries.

Quality

Maintaining the data and software in an analytics cluster is difficult but necessary. The quality of the environment becomes higher when traditional software practices are being applied to the assets in the environment, which are as follows:

- DTAP environments (development → test → acceptance → production)
- Software development principles (SOLID, KISS, YAGNI, clean code, and so on)
- Testing (unit tests, regression tests, integration tests, security tests)
- Continuous integration (version control, code reviews)
- Continuous delivery (controlled releases)
- Monitoring and alerting
- Proper and up-to-date documentation

For example, PacktBank stores its data about products, customers, sales, and employees in the new data lake. The analytics layer of the data lake provides business users and data analysts access to the historical data in a secure and controlled way. Since the results of queries and views must be trusted by management, any updates to the software must go through an extensive review and testing pipeline before they're deployed to the production environment. The models and code are part of a continuous integration and delivery cycle, where the release pipeline and an enforced "4-eyes principle" (the mechanism that ensures that development and operations are separated) makes sure that no software goes to production before a set of automatic and manual checks. When writing code, the engineers often engage in pair programming to keep the code quality high and to learn from each other. Models are documented and explained as carefully as possible and reviewed by a central risk management team.

In this section, we have discussed some important requirements for the analytics layer: performance, cost-efficiency, and quality. Keep in mind that other requirements for data storage that were described in the other layers, such as scalability, metadata, and retention, also play an important role. In the next and final section, we will dive into the specific requirements for the model development and training layer.

Model Development and Training

Data that is used for developing and training machine learning models is temporarily stored in a model development environment. The data store itself can be physical (a file share or database) or in memory. The data is a copy of one or more sources in the other data layers. Once the data has been used, it should be removed to free up space and to prevent security breaches. When developing reinforcement learning systems, it's necessary to merge this environment with the production environment; for example, by training the models directly on the data in the historical data layer.

In our example of PacktBank, the model development environment of the new data lake is used by data scientists to build and train new risk models. Whereas the old way of forecasting whether clients could afford a loan was purely based on rules, the new management wants to become more data-driven and rely on algorithms that have been trained on historical data. The historical data in this example is the combination of clients' history with the number of missed payments (defaults) for each client. This information can be used in the model development environment to create machine learning models.

Security

The model development environment must be secured with a very strict access policy since it can contain a lot of production data. A common practice is to make the environment only available to a select set of data scientists, who only have access to a few datasets that are temporarily available.

Datasets that are copied from a production environment into the model development environment fall under the responsibility of the data scientist who is developing and training the model. They should govern access to the datasets that are temporarily acquired in this way. The level of security is set is on the dataset level and not a row or table level. Data scientists usually require a full dataset and it would make no sense to have fine-grained access once the data has been copied into the secure model environment where only data scientists have access.

Availability

The required availability of data in a model development environment is usually not very high. Since the data is only a copy of the actual production data, the data doesn't have to be available at all times. If the environment becomes unavailable due to a crash or regular maintenance, the data scientists working with the data will just have a productivity loss (for example, they will not be able to work on new models for a few days) but not a production incident (for example, a mission-critical system not working for a few days).

Retention

The data in a model development environment is essentially copied here temporarily, for data scientists to work on. This data should be treated as a developer cache. This means the data should be automatically or manually deleted as soon as the models have been trained. For any subsequent training that's required, the data can be loaded as a copy of the production environment again. It's considered to be good practice to only keep the data in the model development environment for a short period in order to prevent data leaks and to keep resources available. For example, once the data scientists of PacktBank have created a new model for the acceptance of a credit card, the data that was used to train the model can be removed from the model development environment. It's important to store the metadata of the model so that the data can be loaded from the historical archive again if needed.

ACTIVITY 2.01: REQUIREMENTS ENGINEERING FOR A DATA-DRIVEN APPLICATION

Now, we have covered the major requirements that should be considered when building AI systems. In this activity, you will combine the concepts you've learned and defined the requirements of so far for a new set of data-driven applications for a national taxi organization. The customers, taxi rides, drivers, financial information, and other data must all be captured in one data lake. Historical data such as rides in the past should be combined with real-time data, such as the current location of the taxis. The aim of the data lake is to serve as the new data source for financial reports, client investigations, marketing and sales, real-time updates, and more. Algorithms and machine learning models should be trained on the data to provide advice to taxi drivers and planners about staffing and routes. Imagine that you're the architect who is responsible for setting the requirements for the data lake and choosing the technology.

The aim of this activity is to combine all the requirements that you've learned into one set that makes it possible to select the technology for a data-driven solution.

Answer the following questions to complete this activity:

1. Write down a list of data sources that you will need to import data from, either via batch or streaming.
2. With the business aim and the data sources in mind, select the layers of the solution that you will require.
3. What would your ETL data pipelines look like? How often would they be required to run? Are there any streaming data pipelines?
4. What metadata will be captured when importing and processing the data? To what extent can the metadata be used in the extension to the raw and transformed data?
5. What are the security, scalability, and availability requirements of the solution (per layer)?
6. How important are time travel, retention, and the locality of the data?
7. When selecting technology, how will you judge the cost-efficiency and quality (maintainability, operability, and so on) of the solution?
8. What are the requirements for an environment that will serve data scientists who are building forecasting models on top of the historical and real-time data?

Now that you've completed this activity, you have mastered the skill of designing a high-level architecture and can reason about the requirements for AI solutions.

> **NOTE**
>
> The solution for this activity can be found via this link.

SUMMARY

In this chapter, we discussed the non-functional requirements for data storage solutions. It has become clear that a data lake, which is an evolution of a data warehouse, consists of multiple layers that have their own requirements and thus technology. We have discussed the key requirements for a raw data store where primarily flat files need to be stored in a robust way, for a historical database where temporal information is saved, and for analytics data stores where fast querying is necessary. Furthermore, we have explained the requirements for a streaming data engine and for a model development environment. In all cases, requirements management is an ongoing process in an AI project. Rather than setting all the requirements in stone at the start of the project, architects and developers should be agile, revisiting and revising the requirements after every iteration.

In the next chapter, we will connect the layers of the architecture we have explored in this chapter by creating a data processing pipeline that transforms data from the raw data layer to the historical data layer and to the analytics layer. We will do this to ensure that all the data has been prepared for use in machine learning models. We will also cover data preparation for streaming data scenarios.