Course Book

DATA ENGINEERING

DLMDSEDE01

LEARNING OBJECTIVES

To begin this course on **Data Engineering**, we will learn about data systems and data-intensive applications—applications whose performance rely primarily on data. We will learn about the design criteria of data-intensive applications: reliability, scalability, and maintainability, discussing each of these criteria in detail.

Following this, we will explore two different approaches to data processing: batch processing and stream processing. After a short review of batch processing and the MapReduce algorithm as an approach to the batch processing, we will discuss stream processing with Apache Kafka as an example of a typical stream processing platform.

We will then learn about microservice architecture, its advantages, and its disadvantages. We will also learn how microservice architecture is implemented into applications.

The focus then shifts to two important aspects of every data system: data security, and data governance. To do this, we will discuss the data protection act as a concept and, as an example, take a brief look at the General Data Protection Regulation (GDPR). We will also learn how to develop GDPR-compliant software. We will discuss the system security, security requirement engineering, and data system security patterns. Finally, we will delve into the topic of data governance: a set of principles and practices to support efficient performance in evaluating, generating, storing, processing, and deleting corporate data.

The subject of the next unit is the three most common cloud platforms: Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. After a brief introduction into cloud computing, we will learn about each of these cloud platforms and some of their most common services.

Finally, we will learn about DataOps, exploring how best to operationalize data analysis and machine learning applications. We will first discuss the containerization approach and then learn how to build data pipelines for machine learning applications.

UNIT 1

FOUNDATION OF DATA SYSTEMS

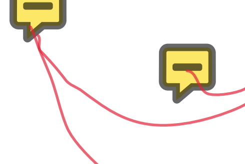
STUDY GOALS

On completion of this unit, you will have learned …

– what data systems and data-intensive applications are.
– what the main features of well-designed data systems are.
– how to quantitatively measure the reliability of an application.
– about the different approaches to application scalability.
– about some of the principles used in software design to increase software maintainability.

1. FOUNDATION OF DATA SYSTEMS

Introduction



“The world’s most valuable resource is no longer oil, but data” (*Economist*, 2017, title), and the platforms to extract this “new oil” are data-intensive applications, which are developed and maintained by data system engineers and data scientists.

In this unit, after a general introduction to data systems and data-intensive applications, we will discuss the key criteria for efficient data-intensive applications: reliability, scalability, and maintainability.

To begin, it is worth briefly discussing the differences between data engineers and data scientists, particularly regarding their responsibilities and the tasks they perform. Consider the data workflow of a data-driven application, as shown in the following figure. The main task of a data engineer is delivering data in the right form to the right people as efficiently as possible. All tasks related to this could be summarized in the first step of a typical data workflow: “data collection and storage.” The following three steps (data preparation, exploration and visualization, experimentation and prediction) are, in most cases, the responsibility of a data scientist (although there is not a sharp boundary) and beyond the scope of this course.

Figure 1: Data Workflow

Data Collection and Storage → Data Preparation → Exploration and Visualization → Experimentation and Prediction

Source: Alavirad, 2020.

Some of the responsibilities of data engineers are as follows:

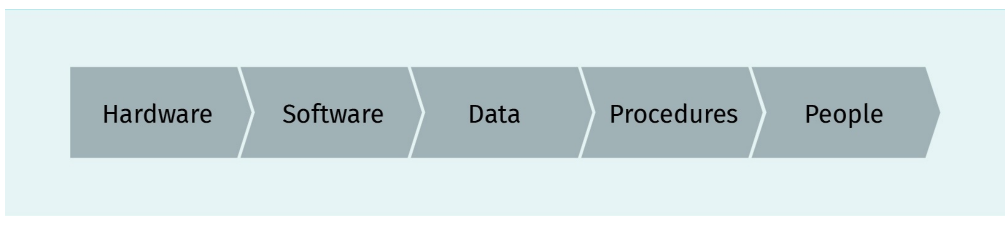
- ingest data from different sources
- optimize databases for analysis
- remove corrupted data
- develop, construct, test, and maintain data system architectures

Speaking generally, a data system is a framework that manages the storage and querying of data with a defined lifetime and consists of data storage, a database management system, queues, and caches.

In more technical terms, a data system could be considered part of a wider information system, which is defined as a group of components that interact to produce information and extract value from that information. The main components of a typical information

system are hardware, software, data, procedures, and people. In this book, when we talk about data systems, we will mostly be referring to the software component of the information system.

Figure 2: Five Components of an Information System



Source: Alavirad, 2020.

Today, raw CPU power or the amount of RAM (the hardware) is rarely a limiting factor for most applications. This is because, nowadays, many applications are data-intensive as opposed to compute-intensive. An application is called data-intensive if data are the primary challenge to its services and performance. Users demand real-time interactions powered by millions of endpoints and large amounts of data. Such applications are now a regular part of or daily applications: social media (Facebook), online commerce (Amazon), and transportation (Uber). In this book, we will focus primarily on data-intensive applications.

During the design phase of a data-intensive application, certain criteria, such the following, should be considered:

- the volume of data that feeds into the applications
- the variety or complexity of the data that is being processed by the application
- the velocity with which the data is being produced or changing

These criteria may seem familiar to you if you already know the four V's of big data: volume, variety, velocity, and veracity.

Like civil engineers, who use off-the-shelf products like girders and prefabricated-wall panels to construct a building, data engineers use pre-programmed building blocks to build an application. For example, when a team is developing an application to analyze stored data, they will not develop a data storage engine from scratch, because database management systems such as MySQL have already been designed perfectly for this job. The same logic applies when an application needs to access the results of an expensive operation in order to speed up the reading of data, search in the database using keywords (search indexes), or analyze the data as they are being produced or received (stream processing).

There are many products (data systems) on the market well-suited to the aforementioned tasks, which can be stitched together to develop a data-intensive application. In this example, the application code is responsible for synchronizing caches and indices with the main database (all systems here are off-the-shelf data systems). However, this, in reality, is not such a simple and straightforward task. Indeed, it is very challenging to find the

right building blocks and tools to achieve the desired solution. There are hundreds of database management systems and various methods for caching and building search indices. For example, some data stores can be used as message queues (Redis), and there are also message queue applications that guarantee database-like durability (Apache Kafka).

The main task of a data system engineer is having the creativity and skills to find the right tools and, more importantly, put them together in the most effective architecture regarding the functionality and the performance of the application. In this scenario, you as a data system engineer hide the implementation details from the client of the end product (i.e., the data-intensive application) using the application programming interface (API) or the service. The final product of this procedure is a new, tailored, and special-purpose data system made from smaller and more general-purpose building block data systems. The execution of this whole procedure defines you as a data engineer.

During the design process of a data system, the system engineer should also address some key questions: How do I ensure that data remains complete and correct during an internal error? How will my application perform when the data flow or volume increases drastically?

The international standard ISO/IEC 25010 (2011) defines the main metrics of software quality with eight characteristics: maintainability, functional suitability, performance efficiency, compatibility, usability, reliability, security, and portability.

In this unit, we will focus on three significant metrics of a typical data-intensive application which should be considered during the design phase:

1. Reliability. The system should perform the defined and expected functional, even in the case of hardware, software, or human error.
2. Scalability. As the system expands (the data volume increases, the rate of data generation becomes quicker, or the variety of data grows), it should be possible (with as little effort as possible) to scale up the system to perform reliably under new circumstances.
3. Maintainability. The people who are tasked with the operation and future development of the system should be able to do their job productively.

1.1 Reliability

Software reliability is defined as the probability of software operating failure-free for a specific period of time in a specific environment" (Lyu, 1995). Although software reliability is described as a probabilistic function and it is a time-dependent event, it should be noted that it is different from traditional hardware reliability. For example, electronic and mechanical parts may become “old” and wear out during their lifetime. Software products do not wear out during their lifetime and, generally, do not change over time. However, software reliability does depend on hardware reliability, as we will see in this section.

Software reliability is one of the most significant characteristics of software quality. Software Reliability Engineering (SRE) is the quantitative study of the operational behavior of software-based systems concerning the reliability of the system (Lyu, 2007). How, then, is it possible to make software reliability a measurable parameter? The answer is reliability metrics.

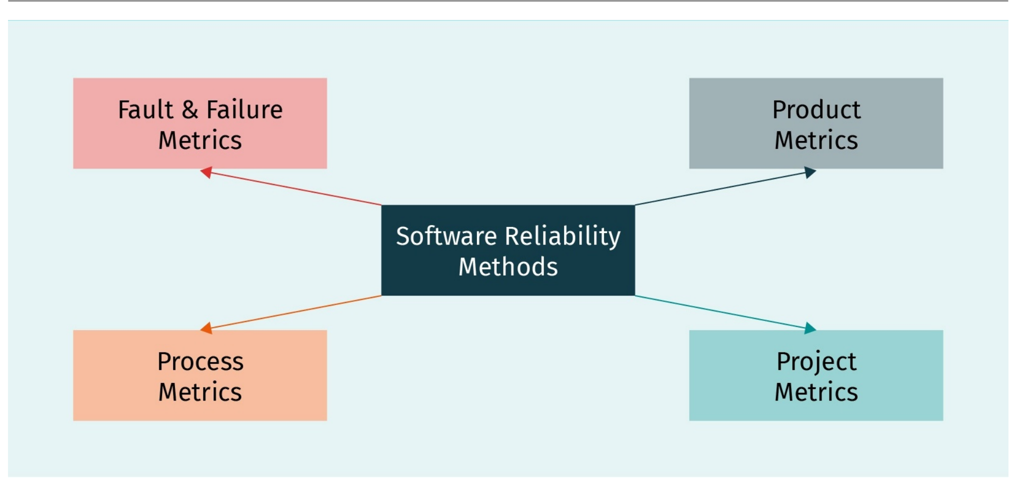
Reliability Metrics

Quantifying software reliability has been a major goal for some time. However, estimating software reliability remains a challenge because we still don't have a solid understanding of the nature of software (Javatpoint, 2020). A proper way to measure software reliability (as we do, for example, electric current), and most of the aspects related to software reliability directly, has not yet been found.

If we cannot measure reliability directly, then, we can measure it indirectly via software reliability metrics. The current methods of software reliability measurement can be divided into four categories (Reliability Analysis Center, 1997):

1. product metrics
2. project metrics
3. process metrics
4. fault and failure metrics

Figure 3: Software Reliability Methods



Source: Alavirad, 2020, based on Reliability Analysis Center, 1997.

Product metrics

The following features could be considered product metrics:

- Software size is a measure of complexity, development effort, and reliability. Lines of Code (LOC), or LOC in thousands (KLOC), is one method of measuring software size. However, there is no standard way of counting. Moreover, this method cannot compare

software that is written in different languages with certainty. The emergence of new technologies for code reuse and code generation techniques also cast doubt on this simple method.
- Using function point metrics are a technique for measuring the functionality of software development based on the number of inputs, outputs, master files, inquires, and interfaces. This method can be applied to determine the size of a software system once these functions are identified. It concerns the functionality delivered to the user and is independent of the programming language.
- Complexity-oriented metrics are also used as a technique for determining the complexity of a program's control structure by visualizing the code with a graphical representation.
- Test coverage metrics are a method of determining fault and reliability. This is done by performing tests on software products based on the hypothesis that software reliability is a function of successfully verified or tested software parts.

Project management metrics

When a development team manages a software project well, it is easier to deliver better and more reliable products. There is a relationship between the development process and the capability to accomplish projects on time and within the desired quality objectives. More reliable software products can be delivered by using a reasonable development process and a risk management process, for example. Keeping documentation of the project is always an important part of software project management.

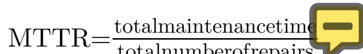
Process metrics

The quality of your software product also depends directly on the process. Process metrics can be applied to evaluate, monitor, and enhance the reliability and quality of a software product. The ISO-9000 certification, a type of quality management standard, is the relevant family of standards for process metrics. Some of these metrics are time to develop the software, the number of bugs found during the testing phase, and the maturity of the process.

Fault and failure metrics

These metrics are used to evaluate the failure-free execution of your software. To accomplish this, the faults discovered during the testing phase, along with those reported by users, will be summarized and analyzed by the development team. Some fault and failure metrics which can be used to quantify the reliability of the software products are as follows (Javatpoint, 2020):

Mean Time to Repair (MTTR) is the time required to fix the failure and is defined as



$$\mathrm{MTTR} = \frac{\mathrm{totalmaintenancetime}}{\mathrm{totalnumberofrepairs}}$$

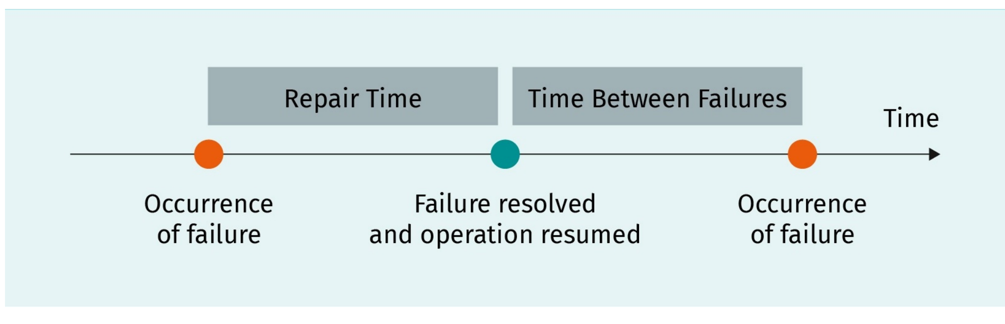
For example, assume a software module fails three times over the span of a working day (8 hours) and the time spent fixing all three bugs is one hour. In that case, MTTR would be 1 hour / 3 = 20 minutes.

Mean Time Between Failure (MTBF) is defined as

$$MTBF = \frac{totaloperationaltime}{totalnumberoffailures}$$

Taking the same software example used with MTTR, the software ran for seven hours of the expected eight and failed three times, with a total downtime of one hour. Here, the MTBF = 7 hours / 3 = 2.33 hours.

Figure 4: Mean Time Between Failure (MTBF)



Source: Alavirad, 2020.

Rate of Occurrence of Failure (ROCOF) measures the number of failures that happen per time interval for a predefined unit. For example, a ROCOF of 0.03 means that, statistically, you should expect three failures every 100 minutes or seconds.

Probability of Failure on Demand (POFOD) is defined as the probability that the system fails when a service is requested. For example, a PODOF of 0.08 means that for every 100 requests made, 8 requests will fail.

Availability is the probability that the system is usable at a given time. This also considers the maintenance time of the system. A 0.995 availability means that, in every 1000 time units, the system is usable for 995. We will discuss availability in more detail later in this unit.

By applying reliability metrics, a software engineer can assess and enhance software reliability. Before releasing the software, testing, verification, and validation are necessary steps. Software testing is heavily used to discover and resolve software defects.

After releasing a software product, field data can be collected and analyzed to investigate the presence of software defects. Fault tolerance or fault/failure prediction methods are helpful techniques for decreasing fault events or the impact of faults on the system.

So far, you have learned that an application is referred to as reliable if it continues to perform its defined and expected functionality correctly even when something goes wrong. When we talk about performing correctly, a user can expect, for example, that

- the application tolerates mistakes made by users such as entering the wrong input,
- the application blocks any unauthorized or unauthenticated access, and

- in the case of a hardware problem, the application continues to perform the defined and expected functionality or halts/exits “gracefully,” i.e., without doing further damage and preserving as much of the state before failure as possible.

The things that could go wrong with software are called faults. Faults can prevent an entire system from providing its defined functionality (Kleppmann, 2017). Software failures may be caused by errors, ambiguities, oversights, misinterpretation of the specification that a software is supposed to provide, negligence, inadequate coding, inadequate testing, or inaccurate or unexpected usage of the software. A data system engineer should design a system to be as fault tolerant as possible. Of course, it is not realistic to expect a system to be 100 percent fault-tolerant, but it should be fault tolerant to an acceptable degrees.

In principle, system engineers should examine the effect of all possible faults on a data system during the test phase of the software development life cycle by deliberately producing faults and observing their impacts on the systems (Kleppmann, 2017).

There is, however, not a lot of analogy between software reliability and hardware reliability, as software and hardware are essentially different, which means different failure mechanisms exist. Hardware faults are generally physical faults, while software errors are design faults. These are harder to detect, classify, and correct (Lyu, 1995). Design errors are closely associated with human factors and the design process. With hardware, design faults do exist but physical faults are a much more common source of problems.

Data system faults can be classified into three main categories: hardware faults, software errors, and human error, each of which we will discuss separately.

Hardware Faults

Hardware faults are one of the main sources of system failure: be it a power system blackout, network connection disruption, hard-disk crash, or a case of overloaded RAM. When the number of hardware components in a data system increases, like in data centers with thousands of hard disks, there is a higher chance of having a faulty component. For example, the mean time between failure (MTBF) of a hard disk is 10 to 50 years: on a cluster of 10,000 hard disks, we expect one faulty component per day.

Hardware failures over time can be displayed as a curve, known as the bathtub curve. Phases A, B, and C on this curve stand for the stand-up phase (burn-in), normal operation phase (useful life), and end-of-life phase (wear-out). The first phase shows a decreasing failure rate, the second phase a constant and low failure rate, and the third an increasing failure rate (ReliaSoft Corporation, 2002).

Figure 5: Bathtub Curve for Hardware Reliability

Source: Alavirad, 2020.

The traditional approach to minimize such hardware faults is redundancy: providing batteries for data centers, **RAID** disks, hot-swappable CPUs, and so on. Therefore, in the case of a component fault, the redundant twin will take over the operation until the faulty component can be repaired and return to normal functionality. However, this approach is not the most economic approach to prevent system failure, as purchasing and maintaining the extra redundant components causes a linear increase in the system cost. When cost is not the most important factor in designing an application, like with air traffic control system software, extra redundancy costs are acceptable.

RAID
This stands for redundant array of independent disks and is a way of storing the same data in different places on multiple hard disks or solid-state drives in order to protect data in the case of a drive failure (Rouse, 2020).

Today, however, most data-intensive applications analyze a big volume of data running on several nodes and machines. Therefore, even for noncrucial applications, a hardware fault could cause notable performance problems. In such cases, it is necessary to decrease the impact of hardware fault by improving software reliability. One example of this approach is distributed computing. In distributed computing, the compute task in question is split into subtasks and distributed across separate remote nodes. An example of such a distributed computation approach can be seen with the MapReduce approach. MapReduce was developed as a technique for writing algorithms to process large amounts of data (Dean & Ghemawat, 2008). The volume of data in an application can be so big that it cannot be stored on a single machine (e.g., massive amounts of human genome data) and must be distributed over many machines to be processed in a reasonable time. With this approach, one single faulty hardware component does not result in a total system failure.

Software Errors

Software errors are probably the most visible and tangible faults experienced by users. When discussing software faults, we can consider the following scenarios:

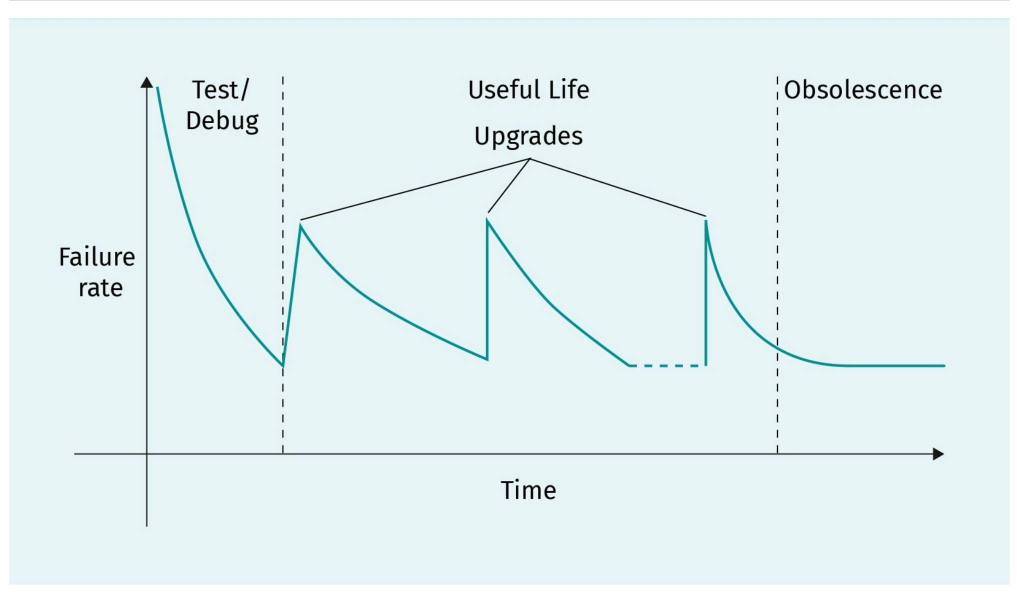
- A process in an application suffers from memory leakage, i.e., uses up memory without freeing it.
- A service of the system stops working.
- A cascading failure, where a small fault in the system spreads to other parts of the system and causes the whole system failure (e.g. a server overload).

Software is a pure cognitive product (Détienne, 2002) and, as mentioned before, software errors are different from hardware failures. Software products do not suffer physical problems like corrosion or aging.

Still, an application that is running faultlessly now cannot be guaranteed to run seamlessly in even the next few seconds. How a software system performed in the last second tells you nothing about whether the system will fail in the next; the consequences of a software failure cannot be anticipated until it happens.

However, the bathtub curve for software reliability, shown below, can help us to predict software errors to some degree. There are two primary differences between hardware and software bathtub curves. First, software does not have an increasing fault rate in the final phase as hardware does. In this phase, software approaches obsolescence, so there are no motives for any upgrades or modifications to be made. Consequently, the fault rate will not vary; the software simply won't be used as much as before. The second difference is that, in the useful-life phase, software will undergo a drastic increase in fault rate each time an upgrade is made. The fault rate levels off gradually, partly due to bugs being found and fixed after each upgrade (Naylor & Joyner, 2014).

Figure 6: Revised Bathtub Curve for Software Errors



Source: Alavirad, 2020, based on Naylor & Joyner, 2014.

The best approach to minimizing the number of such faults is planning a concrete software testing phase, which involves deliberately producing possible errors and checking whether they cause system failure. For example, in an integer input field, entering a string and seeing whether the system response to faulty input.

Human Errors

Human error is defined as an action which fails to produce an expected result, leading to an unwanted consequence (Hollnagel, 1993). Human error is one of the main sources of the errors that cause a system failure.

As the complexity of data systems grow to assist people in their daily routines, the operation of such complex systems has become more complicated, resulting in more space for operator (human) mishandling and errors.

As software and hardware systems and procedures are becoming more and more reliable, human errors are now likely one of the most dominant error types that cause system failures, especially, for example, in security systems. System failures are regularly caused by human errors, such as British Airways' IT chaos (BBC, 2017). BakerHostetler's report shows that 32 percent of all security incidents are caused by employees (BakerHostetler, 2017).

During the designing phase of a data system, an engineer should consider humans (operators) unreliable, as they can, for example, incorrectly configure a system. There are some approaches to minimize the effect of human errors on a system's failure, such as

- providing tools to make recovery from human errors quick and easy, like the ability to easily revert back from a faulty configuration
- designing systems with minimum opportunity for human error, for example, by limiting user input through a user interface instead of the command line
- trying to find all possible sources of human errors and their impact on the whole system during the testing phase of the application

1.2 Scalability

When you design an application, sooner or later it will accrue more and more users who also expect a reliable functionality. A well-designed application should be able to cope with growing demand over time—this is known as scalability, which is the ability to remain reliable in the future, even as the load increases.

In other words, application scalability is an application's ability to grow with time, being able to efficiently handle more and more load. The load is an architecture-dependent parameter. For example, the load could be the read-to-write ratio in a database, or the number of online users in a chatroom.

A system engineer may face different obstacles to scalability, such as limited physical resources (such as physical memory), memory management, an inefficient database engine, a complicated database schema, bad indexing, application server limitations, general spaghetti code, inefficient caching, lack of monitoring tools, too many external dependencies, or bad background job design (Kleppmann, 2017).

A system's performance regarding a growing load may be examined and quantified by answering the following questions:

- When critical resources like CPU and memory stay the same, how will the system perform when you increase the load?
- When the load is increased, how much must critical resources such as CPU and memory be improved to keep the performance of the system constant?

In defining the scalability of the system, we have mentioned two concepts: system load and performance. Let's start by defining these two concepts in more detail.

System load: We use measurable quantities to describe system load. These measurable quantities are called load parameters. The most suitable selection of load parameters depends on the system's design: it may be requests per second to a web server, the ratio of reads to writes in a database, or the cache hit rate. In some cases, the average value is important, while for other cases, other parameters like a hardware bottleneck could be relevant.

System performance: To be able to quantitatively measure the system performance, we must define measurable parameters as we did for software reliability. First, we should note that the system performance is defined for a given functionality. For example, the performance of the Hadoop system could be defined using the runtime of the read/write operations, **throughput**, or the I/O rate. On the other hand, in an online service system, such as Amazon, the response time—the time between a client request and receiving the response from the system—is a performance criterion.

Throughput
This is the amount of work done in a unit of time.

Let's explain one of these performance parameters in more detail. In the case of response time, we can define the percentile response time as a measurable value to evaluate the performance of the system.

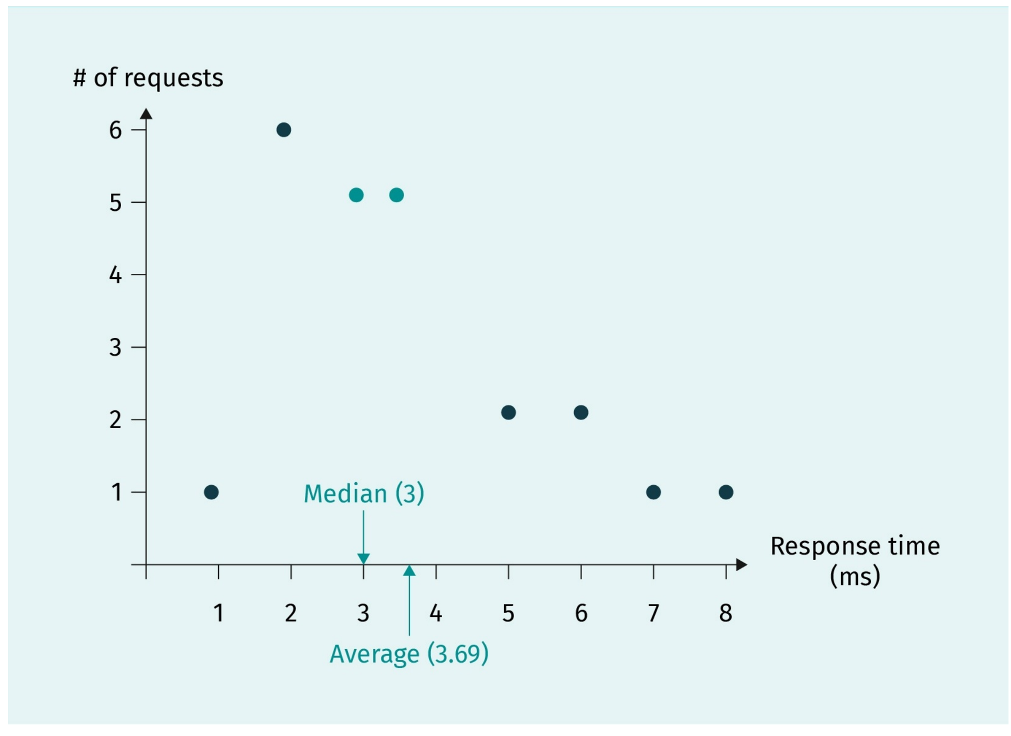
To understand percentiles, we start with a more familiar concept from statistics: the **median**. In a set of values (e.g., a set of measured response times), the median is the response time in the middle of the sorted values. For example, this would be the number 13 in the following set:

Median
In a set with an odd number of elements, the median is the (n+1)/2-th element. In a set with even number of elements the median is the average of n/2-th and (n+1)/2-th elements.

7 9 10 12 **13** 14 17 18 19

In most cases, the median is not the same as average (mean) which is the sum of all response time divided by the number of the recorded response times.

Figure 7: Median vs. Average



Source: Alavirad, 2020.

One interpretation of the median response time is as follows: in fifty percent of cases, the response time is less than the median response time and in the other fifty percent of cases it is higher. We call median the fiftieth percentile, or p50. By following this definition, we could also define for example the twentieth percentile value, or p20, where twenty percent of the measured values are less than p20 and eighty percent are higher than p20. In the figure below, the ninetieth percentile of the HTTP response time is depicted. In this case, ninety percent of the responses take less than 100 milliseconds and ten percent of the responses take more than 100 milliseconds.

Therefore, the percentile is a practical measure to calculate system performance.

Higher percentiles (or tail latencies) such as the 95th, 99th or 99.9th affect the user-experience directly. Here, the 99.9th percentile only affects 0.1 percent of requests. The users who are affected by such high percentiles (and, thus, run the slowest requests) are commonly the most valuable customers as they are the ones who have the most data on your system. Therefore, it is important to keep those valuable customers satisfied. For example, Amazon uses the 99.9th percentile as the benchmark response time of its internal services in the **service level agreement (SLA)**.

Service level agreement
A service-level agreement (SLA) specifies the level of service you anticipate from a vendor, defining the metrics by which

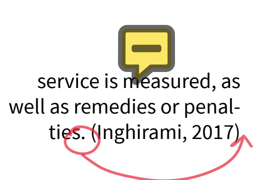
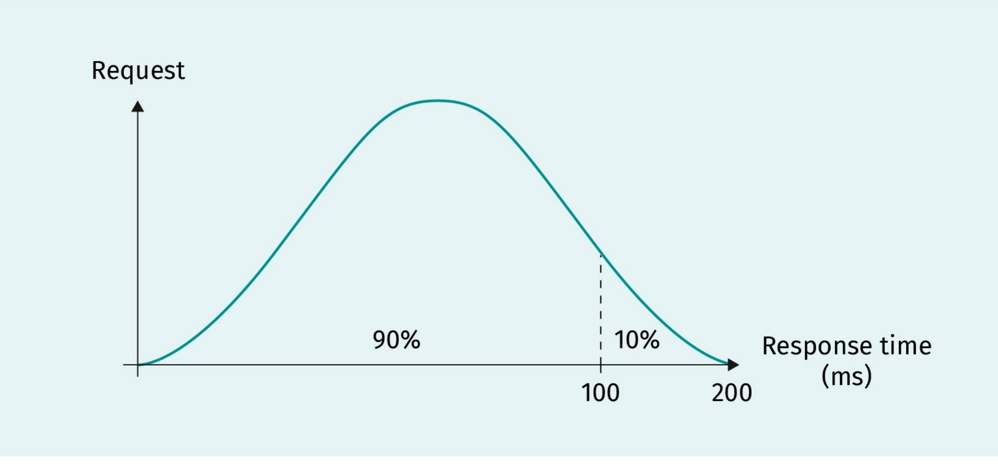


Figure 8: 90th Percentile



Source: Alavirad, 2020.

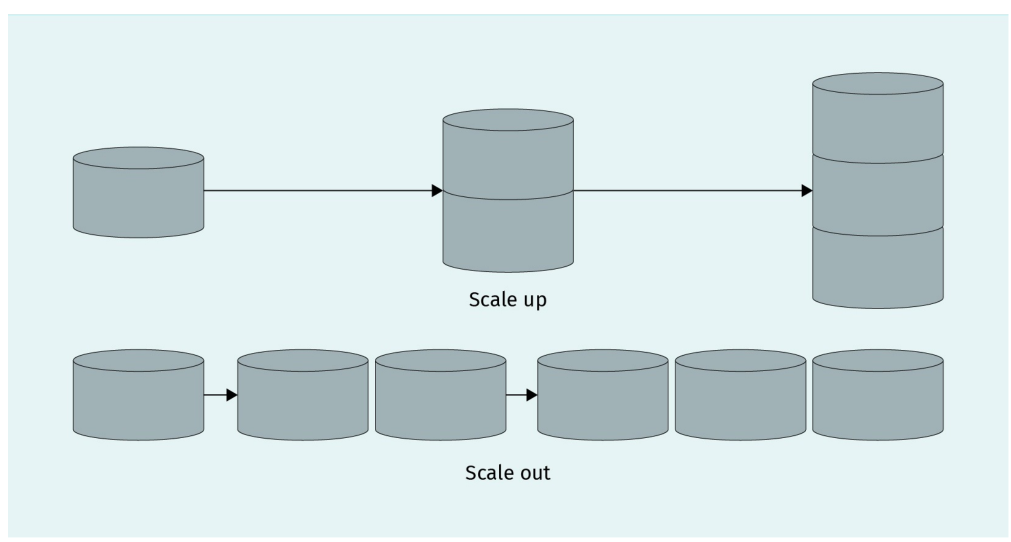
Having defined the load and some key measures to evaluate performance, we should answer the question of how we can design a system to perform the defined functionality when the load increases beyond the primary defined load parameters.

There are, in principle, two approaches to designing a scalable system: scaling up and scaling out.

In scaling up, or using a vertical scaling approach, we increase the supply of resources such as CPUs, memory, network bandwidth, or storage to the existing system in order to reach the state of performance defined in the service level agreement. For cloud-based applications and systems where the system is going to be scaled up, we may move the job onto a more powerful instance or may even move to another host. In the case of on premise applications, scaling up can also be accomplished through adding more resources such as CPUs, connections, or cache sizes.

With the scaling out (or horizontal) approach, the increasing load will be distributed between similar machines or nodes (distributed architectures). The scaling out approach makes it easier for providers to offer “pay-as-you-grow” services to its customers. Service providers normally use both scaling up and scaling out approaches to respond to customer demands and ensure maximum performance.

Figure 9: Scale Up vs. Scale Out



Source: Alavirad, 2020.

For example, the dynamic scaling capabilities of Amazon's EC2 Auto Scaling automatically increases or decreases capacity based on load and other metrics. AWS applies the Auto Scaling approach to monitor applications continually to make sure that they are operating at the desired performance levels. When demand spikes, AWS Auto Scaling automatically increases the capacity of constrained resources, so users maintain a high quality of service.

It should be emphasized that there is no such thing as a generic scaling architecture that can scale for all purposes. Scalability should always be defined for specific performances, such as increasing the volume of reads, the volume of writes, the volume of data to store, and the complexity of the data. When designing a scalable architecture for a particular application, it should be considered, for example, which operations will be more common and which will be less common. Some load parameters to consider might be the number of reads, the size of writes, the volume of data to store, the complexity of the data, the response time specifications, the access models, or any combination of these. For example, a system that has been developed to manage 10,000,000 requests per hour, each 1 kB in size, seems very different from a system that is designed for five requests per hour, each 2 GB in size—despite both systems having the same data throughput.

1.3 Maintainability

The ease or difficulty with which a software system can be modified is known as maintainability. This is mostly determined by the software's source code and architecture.

Software maintenance accounts for the majority of cost and effort during the software's lifetime. We can categorize the following four types of software maintenance (Kleppmann, 2017):

1. Corrective maintenance: some bugs have been discovered and have to be fixed.
2. Adaptive maintenance: some changes to the system are required to adapt it to the environment in which it operates. For example, some modifications must be made to the software due to a major update to the operating system.
3. Perfective maintenance: changes are required when the system has a new group of users or the existing users have new requirements.
4. Preventive maintenance: changes are made to the system to improve its quality or to prevent future bugs from happening.

After a while, the maintenance of a system can be seen as keeping a legacy system alive, a job which may not seem very interesting to software engineers and developers. System engineers should therefore try to design a system in a way that minimizes discomfort during the maintenance phase. For this purpose, we should consider the following three principles of software design: operability, simplicity, and evolvability. It is possible to postpone the legacy phase of a software system when these three principles are taken into account during the design phase. Just like with reliability and scalability, there are no generic and predefined solutions to achieving these goals.

Operability

"Good operations can often work around the limitations of bad (or incomplete) software, but good software cannot run reliably with bad operations" (Krebs, 2012, para. 12). This statement emphasizes the role of operators in the maintenance and operation of a software (even though software is an automated product). Therefore, a good operation team is a crucial part of the reliable performance of a software system.

The tasks of a software operation team are as follows (Hamilton, 2007):

- monitoring system health and quickly restoring service in cases of a bad state
- finding out the causes of the system problems, such as degraded system performance and system failures
- keeping the system up-to-date, especially regarding security patches
- monitoring the interconnectivity of different systems modules to recognize problematic changes before they cause system failures
- predicting potential problems and resolving them in advance (e.g., memory planning)
- defining roadmaps to make operations more predictable
- ensuring the production environment remains stable
- preserving the organization's knowledge of the system

Maintainable data systems can make the operation team's tasks easier by (Hamilton, 2007)

- bringing transparent visibility to the runtime behavior and internals of the system through reliable monitoring
- supporting the automation and integration with standard tools
- ensuring individual machine independence. For example, in the event of a single machine requiring maintenance, the system as a whole continues running reliably
- offering logical documentation of the software and an easy-to-understand operational manual

- providing a reasonable default setup, while also giving the system's administrators enough freedom to override defaults when it is required
- self-healing when appropriate, while also giving administrators manual control over the system state when needed

Simplicity

Small software projects can be written in simple and expressive code but as projects grow, they often are converted into more complex and difficult codes that are hard to understand. This complexity slows down further development of the system, therefore increasing the maintenance costs.

There are many potential symptoms of complexity: an explosion of the state space, strong coupling between modules, confused dependencies, and inconsistent naming and terminology, to name a few. Complexity makes maintenance difficult, which in turn leads to budgets and schedules being overrun. When a software has a more complex code and structure, there is a greater chance that more bugs appear during the further development phase. Simplicity does not mean less functionality or services; it means avoiding accidental complexity. Complexity is defined as accidental if it is not an inherent part of the problem that the application solves, but exists due to poor implementation (Moseley & Marks, 2006).

One of the best methods to avoid accidental complexity is abstraction. A reasonable abstraction can hide a great deal of implementation detail behind an expressive, simple-to-understand software. Multiple implementations of the same source code are not the only benefit of the abstraction—they also result in a higher-quality software, as the quality of each module can be more easily improved. There are many examples of the abstraction approach in the IT world: high-level programming languages are indeed abstractions that hide machine code, CPU registers, and so on.

Evolvability

Evolvability means making change easier. Changes are an inevitable part of the software life cycle: new things are learned, unanticipated use cases emerge, business preferences change, new features are requested, and regulations change.

The agile method provides a framework for adapting changes and the agile community provides technical tools and patterns for frequently changing adaption in software development, such as test-driven development (TDD) and refactoring.

There is a close link between the ease of implementing changes into a data system, its simplicity, and its abstractions: it is always easier to modify simple and easy to understand systems. As this is an essential idea, it is more appropriate to use a different term to refer to agility on a data system level: evolvability.

Why is Maintainability Important?

We may answer this question from two different perspectives:

- Business impact: The maintenance phase of a software system can span ten years or more. During this phase, many issues should be resolved (corrective and adaptive maintenance) and improvement or enhancement requests should be implemented (perfective maintenance). When issue resolution and improvement can be performed more efficiently, both the maintenance efforts and the maintenance costs decrease. Thanks to this, staff have enough time to invest in other tasks like building new functionalities. A shorter enhancement time also means shorter time-to-market for new products and services. However, slow issue resolution and enhancements processes impact the business negatively and can lead to problems, such as missed deadlines or unusable systems.
- Quality enabler: Maintenance enables other quality characteristics such as reliability and scalability. When a system has a reasonable maintainability, making improvements in other quality areas, such as fixing a security bug, becomes easier. In such highly maintainable systems, the modifications and resolutions are much easier to apply, allowing the system engineer to implement quality optimizations faster and more effectively.

SUMMARY

In this unit, we have learned about data systems. We defined a data-intensive application as an application where data are the primary challenge. Three main criteria of a high-quality data-intensive application are reliability, scalability, and maintainability:

- Reliability: the system should perform the defined functionality, even in the case of hardware, software, or human error.
- Scalability: the system should continue to work under increased load.
- Maintainability: the people who are dealing with the operation and future development of the system should be able to do their job productively.

We discussed different metrics to quantitatively measure the reliability of a data system: product metrics, process metrics, project metrics, and fault and failure metrics. We then discussed three faults and errors that could lead to system failure: hardware faults, human errors, and software errors.

There are two approaches to make a system reliable with regard to increasing load: scaling up and scaling out. We also defined the load and the performance of the system in this section.

Finally, we introduced the methods by which a system designer can make the maintenance of a data system easier for the system operators and developers.

UNIT 2

DATA PROCESSING AT SCALE

STUDY GOALS

On completion of this unit, you will have learned …

- about different types of data processing systems: real-time, batch, and stream.
- about the applications and fundamentals of a batch processing system.
- how to use the MapReduce method as a batch processing system.
- what the applications and fundamentals of stream processing systems are.
- about Apache Kafka and Spark Streaming as streaming platforms.

2. DATA PROCESSING AT SCALE

Introduction

Hadoop
"The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models" (Hadoop, 2020, para. 1).

For every data system, we define performance in terms of the expected functionality of the system. For example, the performance of a **Hadoop** system could be defined using the runtime of the read/write operations, throughput, or the I/O rate. For an online service system, such as Amazon, the response time—the time between the client request and receiving the response from the system—is a performance criterion. In such online systems, a request is commonly triggered by the user, who then waits for a response to their request. Considering user satisfaction, the response time should occur within a reasonable time interval. The response time is therefore an important factor in designing data systems.

Regarding the processing mechanism of input data, we can distinguish between three different types of data systems:

1. Real-time processing system (or online-system). In these systems, the response time is within a tight, real-world deadline like milliseconds. In a real-time processing system, the server waits for a request from the client in order to handle it as quickly as possible and sends a response within seconds. One example of a real-time system are stock market applications, where a stock quote is expected from the network within 10 milliseconds.
2. Batch processing system. In a batch processing system, the application processes a large volume of data all at once. The data is collected over a few days and then processed in a non-stop, sequential order. An example of a batch processing system application is the processing of weather data (Anusha & Rani, 2017).
3. Stream processing system (SPS). In a stream processing system (or near-real-time system), the input data, which are the output of another system, are processed almost instantly. This system, by processing the input data, produces output data like a batch processing system. However, more similarly to the online system, a stream processing system's response time is almost real-time. Therefore, the latency of a stream processing system is lower than a batch processing system.

In this unit, we will first discuss batch processing systems. After looking at some batch processing applications and some legacy batch processing systems, we will explain the steps, decisions, and statuses of a typical batch processing job. We will then discuss one of the most important batch processing systems: MapReduce. We conclude this section with an example of a simple MapReduce implementation using Python.

Following this, we look at stream processing systems. We will introduce four legacy information flow processing systems: active databases, continuous queries, publish-subscribe systems, and complex event processing. We will then discuss stream processing systems in detail by looking at data (model), processing and the system architecture of a typical SPS system. Finally, we will introduce Apache Kafka as a modern distributed streaming platform.

2.1 Batch Processing

Consider the following information processing applications:

- Bank Investment Firm X processes the transactions of all international money transfers at the end of each working day.
- Collected weather data are processed to analyze patterns and predict the weather for the coming week.
- Steel Manufacture Inc. produces an operations report at the end of each day to deliver to the operations managers on the next day.
- Tele-D, a telecommunication company, processes monthly call data records that include the information of millions of calls in order to calculate costs and issue bills.

These applications have one thing in common: processing jobs are executed periodically and often process large amounts of information such as log files and database records without direct human interaction. This type of processing is called a batch processing job, which is the topic of this section.

Batch systems have a long history in the computational world, starting with very early operating systems such as the IBM 7094. In this operating system, after recording the information from punch cards on tapes using a card reader and the IBM 1401, the output tapes of the 1401 were used as the input for IBM 7094. This early computer read and processed all data in a single batch without user intervention (of course, after the user puts the tapes inside the machine). It then produced the output again on tapes (to be used as input for another IBM 1401).

To complete a batch job, a user interference is not required. As an example, let's consider a telecom company that generates monthly phone bills. The IT department of this company uses an application that receives the phone records as input and produces a monthly bill for each customer as output. As this process can be executed with a large, singular input file and without any user interaction, it can be run as a batch job. This process consists of two phases: associating each call from the registry with a customer bill and calculating the total amount for each bill. Each phase is a separate step in a batch processing job.

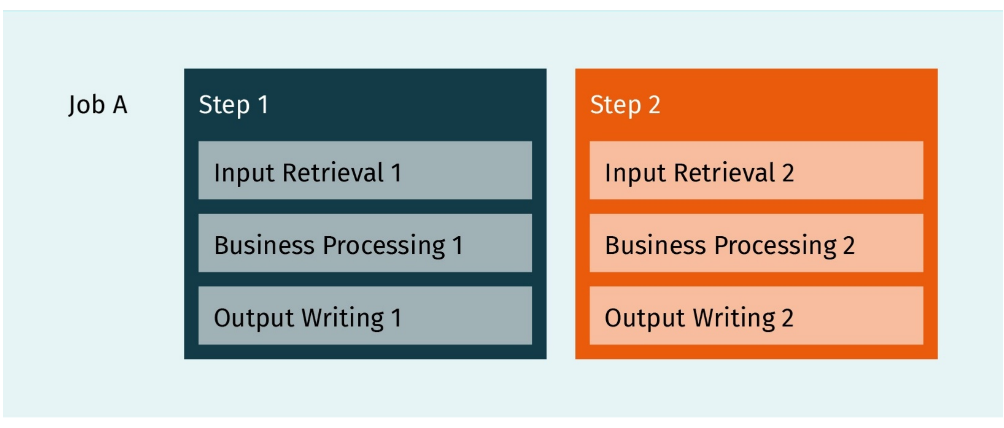
Steps of a Batch Job

We will consider a general set of steps for a typical batch processing job. First, we should define a step in batch processing. A step is an independent and sequential phase, classified into two groups: chunk-oriented and task-oriented steps (Oracle, 2017).

In a chunk-oriented step, the data are imported from a data source like a database, the defined process is applied to them, and, finally, they are stored (in memory or on a disk). In this approach, one item is read and processed at a time and at the end the results are grouped into a chunk of data. The limit of the chunk size is defined via configuration. A chunk-oriented step has three parts (Oracle, 2017):

1. Input retrieval: the chunk step reads one item at a time from a data source.
2. Business processing: in this phase, the defined business process will be applied to data. This process could be sorting, filtering, or a mathematical operation.
3. Output storing: in the final stage, the results of this business process are stored in a chunk of data.

Figure 10: Parts of a Chunk Step



Source: Alavirad, 2020, based on Oracle, 2017.

To understand the chunk steps in a batch processing job, let’s consider the phone bill batch process application again. The batch job in this example has two chunk steps:

1. In the first chunk step, the “input retrieval” reads the call data from the databank and the “business processing” part matches each call with a bill and, if the bill does not already exist, creates a new bill. Finally, the “output storing” part stores each bill in a database.
2. In the second chunk step, the “input retrieval” part reads the bills (the output of the first chunk step) from the databank, the “business processing” part calculates the total amount for each bill, and the “output storing” part stores a printable version of each bill in another database.

As chunk steps process large amounts of data, they are often long-running jobs. In some batch frameworks, it is possible to “checkpoint” (or bookmark) the process to restart the batch job from the bookmark point in case of interruption to save time and resources. As you may have noticed, it is necessary for the chunk step batch framework to bookmark the position of the input retrieval and output storing parts after the processing of each chunk to recover it when the step is restarted.

In a task-oriented step, a single task will be executed, like removing a folder or file from the system. The task-oriented steps are short-running jobs. In the bill batch processing application, we also need a task-oriented step to clean last month's bill files from the database.

After discussing the two most common steps in batch processing, let's also briefly discuss parallelism in batch processing. As batch processing applications process large amounts of data, parallel processing could be very helpful to reduce the response time. There are two approaches to boost batch applications via parallel processing:

- using independent steps that can run on separate threads
- using chunk-oriented steps where the processing of each item is independent of the processing result of the previous items and can therefore run on separate threads

Statuses and Decisions in a Batch Job

In addition to these steps, batch frameworks also provide a status for each individual job. The status of a job could be one of the following:

- running
- successful
- interrupted
- error

Batch jobs also contain decision elements. A decision element uses the exit status of the previous step to decide an action for the next step (continue or exit). This element also sets the status of the job in case of a termination.

We can define the following functionality for a batch processing framework:

- defining jobs, steps, statuses, decision elements, and the relationships between different elements
- providing parallel processing
- maintaining states of each job
- handling errors
- launching jobs and resuming interrupted jobs

Now, let us discuss one of the most powerful parallel batch processing systems: MapReduce.

MapReduce

Batch processing is an important building block for developing reliable, scalable, and maintainable applications. In the rest of this section, we will consider one of the most successful batch processing algorithms: MapReduce, which was developed by Google (Dean & Ghemawat, 2008). This algorithm is a fairly low-level programming framework that enabled the scaling out of computations on commodity hardware. MapReduce has been implemented in many data processing solutions such as Hadoop and MongoDB. MapReduce is the foundational building block of most big data batch processing systems. It should, however, be emphasized that MapReduce is coming close to its "retirement" and, in recent years, has been replaced with more modern technologies (e.g. Apache Spark). It is nonetheless beneficial to study its fundamentals as a computing model, as it laid the foundations for scalable and reliable data processing.

On a UNIX system, jobs use standard input (`stdin`) and standard output (`stdout`) as input and output methods. For example, when you run a program and do not specify stdin and stdout, the program reads the input from the keyboard and writes the output on the screen. In MapReduce, jobs read and write files from a distributed file system. In the case of Hadoop implementation of MapReduce, for example, the distributed file system is called HDFS (Hadoop Distributed File System).

Shared-nothing
A shared-nothing architecture generally does not require any specific hardware. Computers are connected by a conventional data center network.

HDFS is based on a **shared-nothing** architecture. In this architecture, each machine or virtual machine running the database management system is called a node, and each node has its resources like memory, CPUs, and so on. The orchestration between nodes is carried out by a piece of software (the MapReduce engine).

Daemon process
This is a process that runs in the background. The user does not have direct control over the daemon.

HDFS has **daemon processes** which run on each machine. HDFS has five main daemons: NameNode, the secondary NameNode, DataNode, JobTracker, and TaskTracker. We could consider the HDFS as a big file system that uses the resources of each node to run the daemons (here, we are concerning HADOOP2 specifically).

The NameNode daemon is the master node in HDFS. This node is responsible for storing the metadata regarding all files and records. It holds information about the file locations on HDFS. The secondary NameNode is not used as a backup of the NameNode as the name might imply. It is rather a helper element which checkpoints NameNode's file system namespace. The DataNodes are the slave components of the architecture and only responsible for data storage. On startup, a DataNode connects to the NameNode, waiting until the NameNode comes up. A DataNode responds to requests from the NameNode for file system operations. As a master component, the JobTracker manages all jobs and resources in the Hadoop cluster. The TaskTrackers are the slave components that run on each server in the cluster and they are responsible for the execution of MapReduce jobs, including the periodic delivery of status messages to the JobTracker.

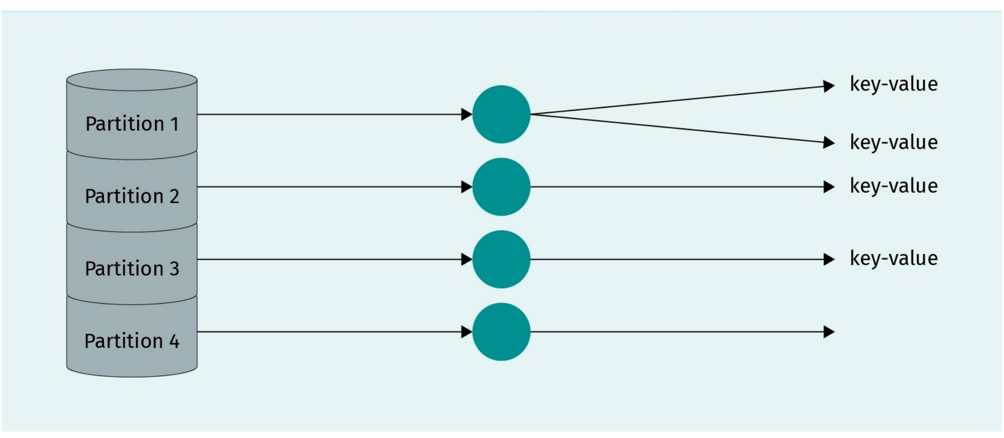
The main advantage of the implementation of MapReduce like Hadoop is automated parallelism: MapReduce distributes the computation between nodes to execute the jobs in parallel without the user having to write code to manage the parallelism.

The MapReduce implementation consists of two main functions: Map and Reduce. These two functions are distributed to nodes by the MapReduce framework. The Map function produces key-value pairs from the input data (partitions). Then the Reduce function uses the produced key-value pairs to sort the key-value data and further process them, i.e., by aggregating values with the same key. In the following, we will discuss these two cornerstones of the MapReduce framework in more detail.

Map

In Hadoop implementation, the input to a MapReduce job is a directory on HDFS. Each directory is made up of smaller units called partitions. Each partition must be processed separately by a map task (a map task is a process that runs the map function).

Figure 11: Map Side of MapReduce



Source: Alavirad, 2020.

Typically, each input file has a size to the order of hundreds of megabytes. The MapReduce scheduler runs each mapper (map task) on a machine that has a copy of the input file. This procedure is known as putting computation near the data. The map task is called once for every input partition to extract the key-value pair from the input partition. The number of mappers is therefore determined by the number of partitions.

System engineers should only define the code inside the mapper (for a simple mapper, as in the following Python example) that takes input partition data and produces key-value pairs. In the following example, each key is a word with value equal to 1.

Code

```
def mapper(key, value):
  # Split the text into words and yield (word,1)
  # as a pair
for word in value.split():
normalized_word = word.lower()
yield normalized_word, 1
```

Reduce

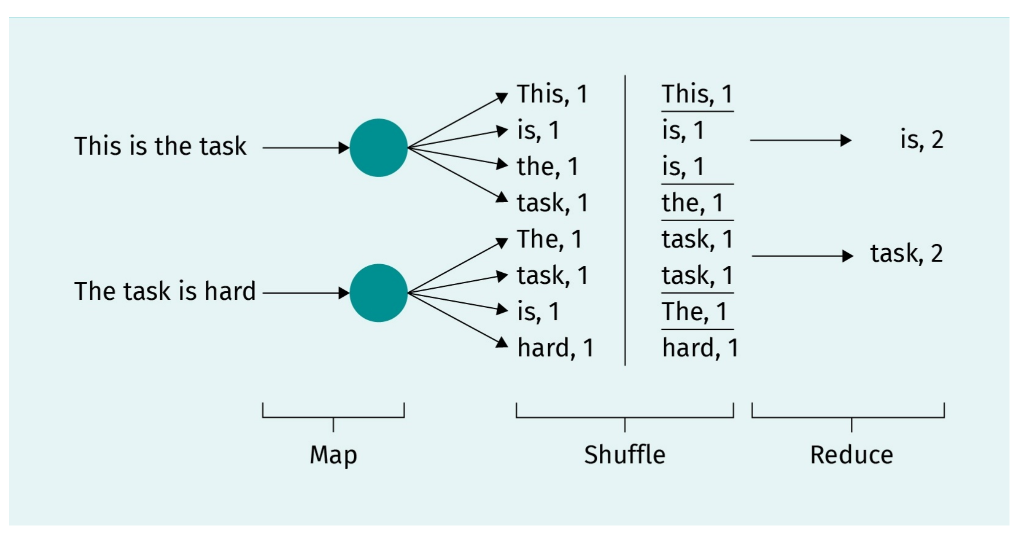
When the mappers finish their job and write the output key-value pairs in sorted output files, the MapReduce scheduler calls reducers to fetch the output files from mappers. The process of reading output files from mappers, sorting, and copying them to the reducers partitions is called shuffling. The reduce task reads the sorted data from the mappers and keeps the sort order. The reducer is called with a key to scan all records with that key. Here is an example of a simple reducer code in Python:

Code

```
# Reduce function, applied to a group of values with the same key
def reducer(key, values):
 # Sum all the values with the same key
 result = sum(values)
 return result
```

In the above example, the mapper returns a (key, value) in the form of a (word,1) and the reducer is called over all values with the same keys to create a distributed word counting pipeline (see figure below).

Figure 12: Reduce Side of MapReduce



Source: Alavirad, 2020.

Here, the number of reducer partitions is fewer than the number of mapper partitions.

A single MapReduce job is limited in the problems it can solve, so it is always more practical to chain multiple MapReduce jobs together in a workflow. In this case, the output of one MapReduce job will be the input of another MapReduce job. Using the above example, if we want to find the top five most common words in a text, we need a second MapReduce job to sort and select the top five words. There are some workflow schedulers (e.g., Airflow) that can handle the dependencies between jobs.

Hands-on

We conclude this section with a simple lightweight MapReduce implementation in Python (Fairley, 2010).

First, run the following example file from the command line as:

Code

```
Python example.py

#example.py
#!/usr/bin/env Python
import mincemeat

#Here we define the data for the mapper input
data = ["Humpty Dumpty sat on a wall",
   "Humpty Dumpty had a great fall",
   "All the King's horses and all the King's men",
   "Couldn't put Humpty together again",
   ]
#Mapper
def mapfn(k, v):
 for w in v.split():
   yield w, 1

#Reducer

def reducefn(k, vs):
 result = 0
 for v in vs:
   result += v
 return result

s = mincemeat.Server()

# The data source can be any dictionary-like object
s.datasource = dict(enumerate(data))
s.mapfn = mapfn
s.reducefn = reducefn

results = s.run_server(password="changeme")
print(results){'a': 2, 'on': 1, 'great': 1, 'Humpty': 3, 'again': 1, 'wall': 1,
'Dumpty': 2, 'men': 1, 'had': 1, 'all': 1, 'together': 1, "King's": 2,
'horses': 1, 'All': 1, "Couldn't": 1, 'fall': 1, 'and': 1, 'the': 2,
'put': 1, 'sat': 1}
```

This was a very simple example of MapReduce implementation using Python. However, with some minor modifications to the source code, and by running the client on multiple machines, it becomes possible to process even larger amounts of data (gigabytes).

2.2 Stream Processing System

In batch processing, the input and the output of a batch job are files. The output file (derived data) can be used as the input of another batch job through a piping mechanism. In batch processing, the size of the input file is also bounded or is known before running a job. For example, for the merging and sorting of map outputs in MapReduce, the exact size of the input file should be known. The reducer should wait until the mapper has produced all key-value pairs.

What about cases where the size of the input file is not constant, but changes with time, such as the number of interactions in social media, or the temperature measured by a smart thermometer? One solution could be processing the aggregated data periodically (e.g., daily or hourly). This, however, would not meet the business requirements of many impatient users, because any change in the input data will only be reflected after an hour or a day. For instance, consider ordering a laptop from an online shop. The online shop would like to recommend more articles related to your current order and your purchasing history, for example, accessories for the laptop, such as a docking station. If the processing system of the online shop uses an hour-based batch processing approach, it will be far too slow for any accessory to be suggested, as by the time an hour has passed, you will have already completed your purchase.

Therefore, it is usually more beneficial to reduce the processing period to, for example, a few seconds, or to even make this process continuous by ignoring fixed time slices and instead processing data as they are received (events). This is the main idea behind stream processing.

Before starting to discuss stream processing in more detail, it would be beneficial to mention some other use cases of stream processing:

- monitoring a production line
- geospatial data processing for applications geared toward topics like climate change and natural hazard prediction and mitigation (Li et al., 2020)
- algorithmic trading and stock market surveillance
- smart device applications
- smart grid
- predictive maintenance
- fraud detection

Information Flow Processing Technologies

In batch processing systems, since the input and the output of the processing jobs are files, the first step in the processing is to parse the chosen file in a set of records. In a stream processing system, a record is known as an event: a small, self-contained object containing detailed information of what is happening at a given time.

An event could be a user clicking, the temperature of a sensor, or a CPU utilization metric. The events are encoded as text strings, JSON files, or binary forms, and the encoded information can be appended to a file or inserted into a database.

Historically, there were several systems to process information flow, which can be categorized into four classes: active databases, continuous query (CQ) systems, publish-subscribe ("pub-sub") systems, and complex event processing (CEP) systems. These systems are the ancestors of modern continuous data stream processing systems (SPSs).

Active databases

In an active database, an extension to the database reacts to continuous changes in the stored data. A common characteristic of all active databases is their reliance on "event-condition-action" rules: capturing the events, testing the condition of these events, and performing the action which should be taken on receipt of the events.

An event represents a time-dependent change in the database, such as the insertion of a new row into the database. A condition describes the state in which the database should be in in order to execute an action. The action is the response to an event and is executed when the database is in the relevant condition.

Some examples of active databases are Ode (Lieuwen et al., 1996), HiPac (Dayal et al., 1998), Samos (Gatziu & Dittrich, 1994), and Snoop (Chakravarthy & Mishra, 1994). It is also possible to implement ECA rules into traditional database management systems (DBMS) using SQL triggers.

In some active databases, the ECA rules only react to the data stored on that database (closed active databases), while in others, the ECA rules also react to internal and external events (open active databases). Open active databases are more appropriate for stream processing of continuous data as the source of a data stream—in most cases—is an external one (e.g., streaming data from a sensor).

Continuous queries

A continuous query (CQ) system uses standing queries (unlike the normal snapshot queries) to continuously monitor information change and, in the case of a user-defined event, return the results. A CQ consists of three blocks: a query, a trigger, and a stop condition. Whenever the trigger condition meets, the query will be executed and returns the result. The query will be terminated as soon as the stop condition is met. An example of a CQ system is NiagaraCQ (Chen et al., 2000). In modern SPSs, the data are continuously run through the queries, unlike in CQs, where the queries are being continuously run through the data.

Publish-subscribe systems

A publish-subscribe system (pub-sub system) is an event-driven system. An event is generated by producers (also called publishers or senders), processed by a broker, and received and processed by consumers (subscribers or recipients). The events in the streaming system are grouped into topics or streams.

The pub-sub system decouples publisher and subscriber, i.e., publisher and subscriber do not know each other, and the delivery of the events is performed through a broker network—a set of broker nodes connected on a WAN.

There are two types of pub-sub systems: topic-based and content-based. In topic-based systems, each publication can have one or more topics, and a subscription can subscribe to these topics. In a content-based pub-sub system, a publication contains a set of properties, and a subscription is defined based on the conditions of these properties. For example, a subscription may look for all publications containing the "price" property with a value of less than 500 Euro with the "product type" property as "mobile phone".

Most content-based pub-sub systems provide atomic subscriptions. That is, subscriptions that are defined in individual publications. An example of such a system is IBM Research's Gryphon (Strom et al., 1997). Some pub-sub systems also support composite subscriptions, meaning subscriptions are defined on a series of publications using methods like sequences and repetitions. Such pub-sub systems are more complicated to implement as the broker's router algorithm must filter publications as fast as possible to ensure a reasonable response time. In the rest of this section, when we discuss CEP (Complex Event Processing) systems, we will see some similarities to composite subscription pub-sub systems.

Complex event processing systems

The main reason to develop complex event processing systems is to collect, filter, aggregate, and combine the data from different sources on a central computing platform and deliver them to analytic applications.

CEP systems provide detection of complex events, which are a composite of simpler events created as a result of additional analysis and processing like temporal sequences (e.g., event A follows event B), or negation (e.g., event A does not appear in the sequence). Most CEP systems, like IBM WebSphere Business Events (IBM, 2020c), use rule-based detection tasks. These tasks are expressed in terms of complex patterns as a function of logic conditions, uncertainty models, and so on.

After a historical review of information flow processing systems, we will continue by discussing the structure of typical modern stream processing systems.

Modern Stream Processing Systems

The information flow processing systems discussed in the previous subsection can provide, to some extent, continuous data processing. Most of them are limited to simple and small-scale streaming data applications. For most streaming processing applications (SPAs), however, we need flexible, scalable, and fault-tolerant applications that can process large amounts of data in motion in real-time. These requirements led to the emergence of stream processing systems (SPSs). In the following, we will briefly discuss the stream processing paradigm, starting with data.

Data

We will begin with the type of data that are processed by SPAs. A streaming data source is the producer of the data stream and can be broken into distinct data items known as tuples. There are many sources of tuples around us such as sensors, medical devices, and smart devices. The source of stream data can be also a data repository such as a relational database, a data warehouse, or even a plain text file. The sink of the data stream also has tuples as the end products of processing applications like visualization and monitoring tools. We will now take a look at the three main characteristics of a stream processing data model.

The data tuple is the fundamental data item embedded in a data stream and can be processed by an application. A tuple is similar to a row in a relational database with a set of attributes. A data schema defines the structure of data tuple. The data stream is a potentially infinite sequence of tuples sharing the same schema.

In an stream processing system, data tuples are processed by analytic components of applications. In the following section, we will explain some of the basic concepts of stream processing.

Operator

An operator receives the input tuples from the incoming data stream, processes them, and outputs them as an outgoing stream. We could consider the following tasks as suitable for an operator:

- Edge adaption. Converting the incoming tuple into an appropriate format that can be consumed by downstream operators
- Aggregation. Collecting a subset of tuples based on the defined conditions
- Splitting. partitioning a stream into multiple streams based on defined conditions for more efficient processing by downstream operators
- Logical and mathematical operations. Performing logical or mathematical operations on multiple tuples from one or more data streams
- Sequence manipulation. Reordering, delaying, or altering the temporal properties of a stream

Data mining, machine learning, or **unstructured data** processing techniques can also be applied.

Unstructured data
Any data that have no structure by pre-defined models and schemas, such as audio and video files, are called unstructured data.

Stream connection

Stream data are transported through a stream connection between the output port of an upstream operator and the input port of a downstream operator.

Stream processing application

A set of operators and stream connections that work in harmony to process the streaming of data is called a stream processing application (SPA).

There are fundamental differences between a SPA and a regular procedural application:

- SPAs should be designed to cope with the concept of data-in-motion—when and how fast the data arrives and also the order of data.
- SPAs should process data-in-motion, which requires a buffering mechanism for random access to data.
- SPAs should perform **one-pass** processing on the incoming data as multiple passes in order to avoid performance degradation.

One-pass algorithm
A one-pass algorithm is a streaming algorithm which reads its input exactly once.

System architecture: A stream processing system (SPS) is a middleware, for which an SPA digests, analyzes, and outputs the incoming stream data. We can consider two main components for a typical SPS: an application development environment and an application runtime environment.

Application development environment

This environment provides the framework and tools to implement stream processing applications and contains two components:

- a programming model that provides a query or programming language to develop the application's internal processing logic and structure
- a development environment that provides a platform for implementing, testing, and debugging the applications

Application runtime environment

The application runtime environment provides the infrastructure (i.e., support and services) for running one or more SPAs. This environment normally consists of a software layer that is distributed over a multi-host cluster for running instances of the SPAs. There are also components that manage the runtime environment itself. The SPS runtime environment should also provide the administrators with enough tools to manage SPAs.

Runtime management interfaces should also provide tools for application life cycle management, access and management of computational resources, data transport, fault tolerance, and authentication and security management.

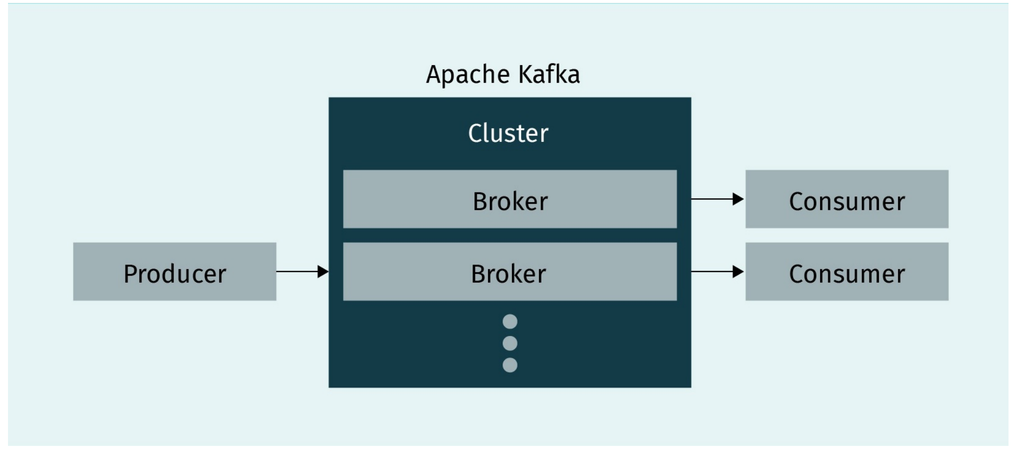
Kafka—A Distributed Streaming Platform

There are several implementations of SPSs by different research institutes and companies. In this part, we will focus on one of the state-of-art SPSs: Apache Kafka.

Apache Kafka is a distributed event streaming platform capable of managing trillions of events per day. Initially conceptualized as a messaging queue, Kafka is based on the idea of a distributed commit log (which will be explained in greater detail shortly). Kafka was developed by LinkedIn in 2011 (written in Scala and Java), and has quickly evolved from a messaging queue to a fully-fledged event streaming platform (Apache Kafka, 2017a).

The main building block of Kafka is an immutable commit log, to which any number of systems and real-time applications can subscribe and publish data. Unlike most messaging queues, Kafka is a highly scalable, fault-tolerant distributed system, allowing it to be implemented for purposes such as the managing of passenger and driver matching at Uber. Put simply, Kafka is a platform that provides a durable publish-subscribe messaging system (Apache Kafka, 2017a).

Figure 13: Apache Kafka Components



Source: Apache Kafka, 2020a.

A complete Kafka system requires four main components: the broker (cluster), Apache ZooKeeper (a cluster coordination tool), producers, and consumers.

A broker handles client requests (producers, consumers) and stores replicated data on the Kafka cluster, which is a cluster of brokers. A producer publishes the records to the broker. The consumer uses the data on the broker and ZooKeeper then stores the state of the cluster (brokers, users, topics, configurations, etc.) and coordinates the system.

Before explaining these four components in more detail, we should define the core abstractions in Kafka for a stream of records: topics and partitions.

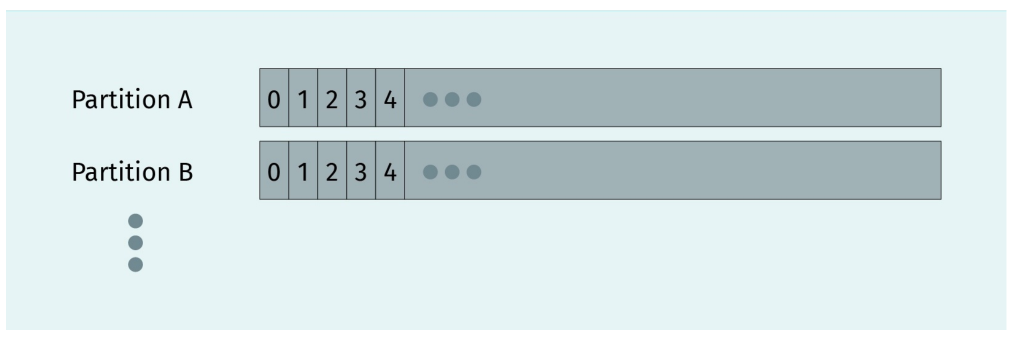
Topics

A topic is a category or feed name on the Kafka cluster to which messages are published. In other words, the records are organized by topic. Producers publish data into topics and consumers subscribe to data stored on topics. Topics in Kafka are multi-subscriber; that is, a topic can have zero, one, or many consumers. Records stay in the cluster for a configurable retention period. For example, for a two-day retention policy, a published record is available for two days, after which it will be removed to free up space. By this time, downstream databases for data persistence should have caught up with the two-day-old data.

Partitions

In Kafka, topics are split into partitions, which contain records in an immutable sequence that is continually appended to a structured commit log. To identify each record inside a partition, they are assigned a sequential ID called the "offset."

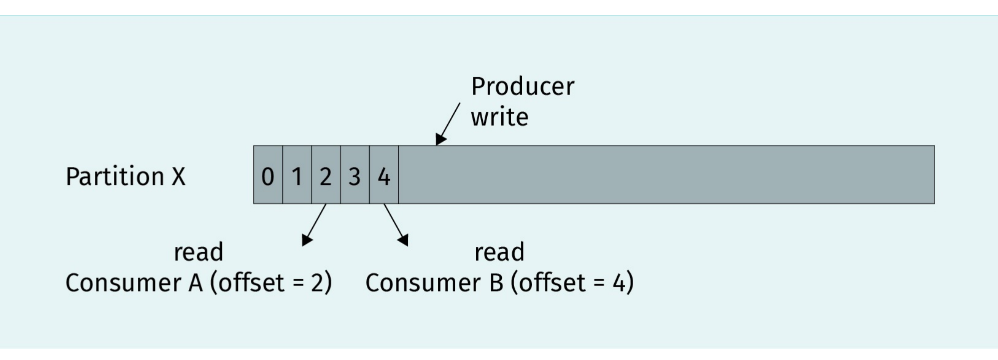
Figure 14: Anatomy of a Topic and Partition



Source: Alavirad, 2020.

Usually, a consumer will advance its offset in a linear fashion, but, as the consumer can control this position, it can also consume records in an arbitrary order. For example, a consumer can consume an older offset to reprocess data, or jump to the most recent record and start consuming from "now."

Figure 15: Kafka Offsets



Source: Alavirad, 2020.

Partitioning topics has its advantages, including:

- It makes it possible to have logs larger than can be stored on a single server. Although every single partition should fit the size of the host server, a topic may have several partitions distributed over a cluster of servers.
- Each partition can act as a parallel unit.

Having introduced the core abstractions of the data model in Kafka, we will now look at Kafka's main components.

Broker

As mentioned before, Kafka is a distributed streaming system that provides the basic functionalities of a publish-subscribe system. Kafka is designed to run on multiple hosts. Each host in the Kafka cluster contains a server called the broker. The broker stores messages published to the topics and serves consumer requests.

When one of the hosts is unavailable, other hosts continue running. This feature is what enables the "no downtime" and "unlimited scaling" characteristics of publish-subscribe systems.

Kafka assigns each partition one "leader server" and zero or more "follower servers." The main job of the leader is to manage read and write requests for that partition. The follower servers passively mirror the leader. When the leader goes offline, one of the followers takes over the leadership. A server may play the role of the leader for partition A and the role of a follower server for partition B. With this method, loads are well distributed among servers.

There is another broker for each partition called the controller. The main task of this broker is to maintain the leader/follower relationship for all partitions.

ZooKeeper

This module is not technically a part of the Kafka framework, but Kafka does not work without it. ZooKeeper has the following tasks:

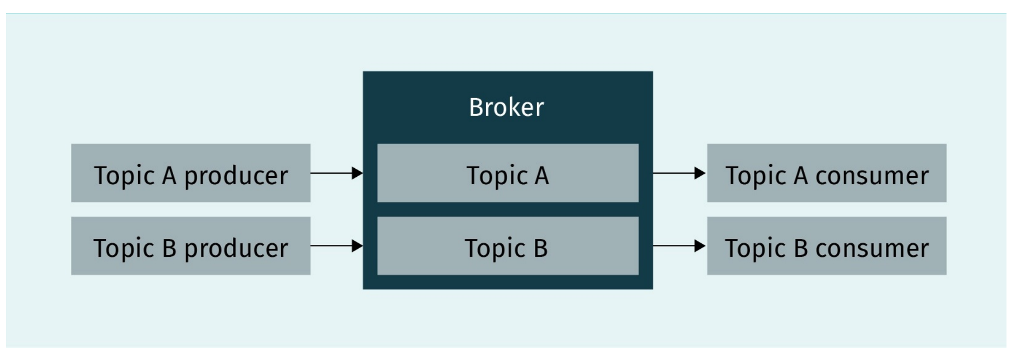
- Electing a controller. ZooKeeper elects a controller for each partition, makes sure there is only one, and then elects a new one if the original controller goes offline.
- Cluster membership. ZooKeeper manages which brokers are alive and part of the cluster.
- Topic configuration. ZooKeeper also controls the existing topics, number of partitions per topic, who is the preferred leader, etc.
- Metadata storing. the location of partitions, the configuration of topics, and other metadata are stored outside of Kafka, in a separate ZooKeeper cluster.

Producer

In Kafka, a producer publishes (sends) data to the topics. It is the responsibility of the producer to choose which record should be assigned to which specific partition. The procedure can be managed in a **round robin** method to distribute the load over partitions, or it can be done using a semantic partition function.

Round robin (RR)
In a round robin method, the available resources are distributed between a partition equally and in a circular order without priority.

Figure 16: Brokers in a Pub-Sub System



Source: Alavirad, 2020.

Consumer

The consumer is an application that reads messages from a Kafka topic, runs validations against them, and writes the results to another data store. For this purpose, Kafka provides the Consumer API (KafkaConsumer) to create a consumer object. Kafka’s consumers read messages from the partition, starting from a specific offset. In Kafka, consumers can choose to read which offset a message is from, enabling them to join the cluster at any point in time.

In Kafka, we label consumers with a consumer group name. The idea behind this label is to deliver each data record of a topic into just one consumer instance within a subscribing consumer group. In the figure below, you can see the anatomy of Kafka clusters and consumers. In this example, we have two brokers (Server1 and Server2), four partitions (P0—P3), and two consumer groups (group A and group B). Group A has four consumer instances (C1—C4) and Group B has two consumer instances (C5 and C6).

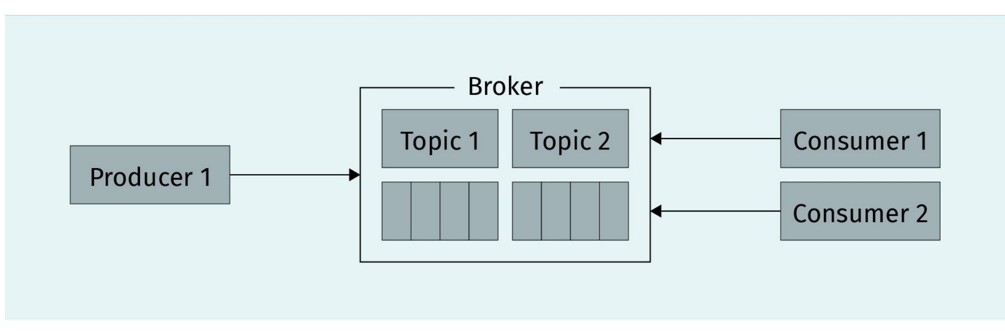
Figure 17: Most Common System Encryption Algorithms

Source: Alavirad, 2020.

Record flow in Apache Kafka

To understand the mechanism of the data stream in Kafka, we will take as an example a system with a single broker and two topics, where each topic has four partitions.

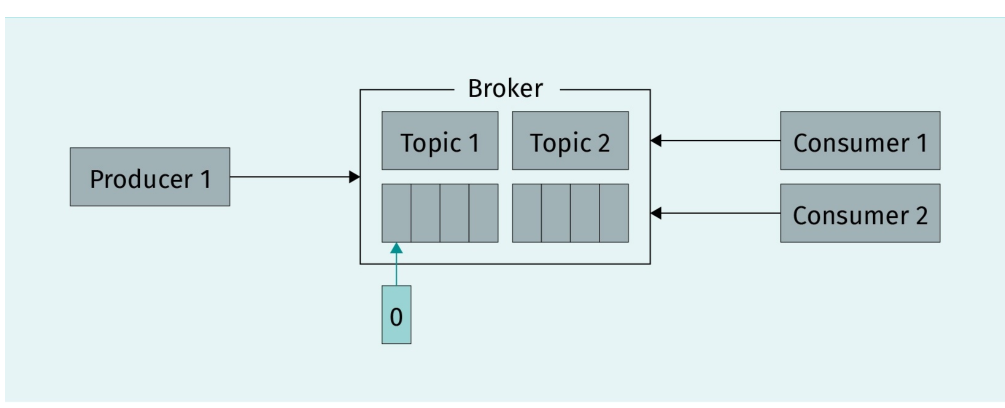
Figure 18: Apache Kafka Record Flow Step 1



Source: Alavirad, 2020.

In the first step, the producer sends a record to the first partition in Topic 1. As this partition is empty, the offset takes the value 0.

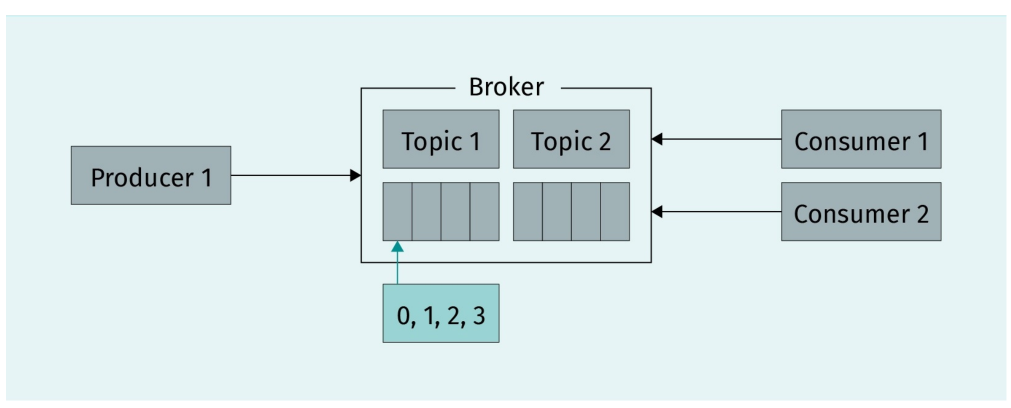
Figure 19: Apache Kafka Record Flow Step 2



Source: Alavirad, 2020.

When the producer sends more records to this partition, the offset takes values 1, 2, etc. This is called a commit log, as it is impossible to change the existing record in the log.

Figure 20: Apache Kafka Record Flow Step 3



Source: Alavirad, 2020.

Kafka streams API

Now let's have a look at the real-time stream processing using Kafka. In Kafka, a stream processor receives continual streams of data from input topics, processes them, and produces continual streams of data to the output topics.

When we want to do simple, real-time processing directly, it is possible to perform it with only producer and consumer APIs. However, for more complex transformations, Kafka offers a fully-integrated Streams API (Apache Kafka, 2017b). This API facilitates solving more complex problems like managing out-of-order data, reprocessing input as code changes, and performing stateful computations.

Use Cases of Kafka Streams API

Here are some use cases of Kafka's Streams API (Apache Kafka, 2020b):

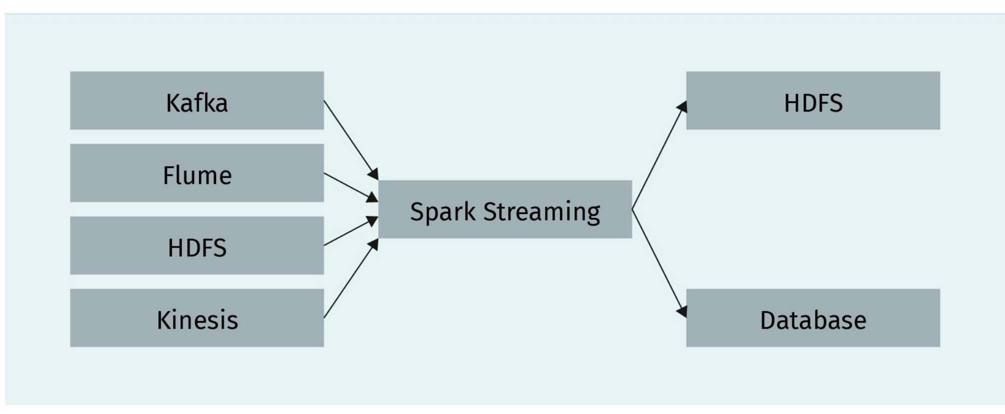
- *The New York Times* uses Apache Kafka and Kafka Streams to store and distribute the published content in real-time.
- Zalando uses Kafka as an ESB (enterprise service bus) for the transition from a monolithic to a microservice architecture. Using Kafka for processing event streams enables Zalando to perform near real-time business intelligence.
- Trivago uses Kafka Streams to enable its developers to access data within the company. Kafka Streams empowers the analytics pipeline and provides endless options to explore and operate on data sources.

Spark Streaming

Let us have a brief look into another streaming processing tool from Apache: Spark Streaming. Spark Streaming, which is an extension for the core Spark API, provides scalable, high-throughput, fault-tolerant stream processing of live data streams. The source data can be delivered to Spark Streaming from different sources like Kafka or TCP sockets.

These data can be processed using algorithms expressed with high-level functions like map, reduce or join. The processed data can be written to file systems, databases, and live dashboards.

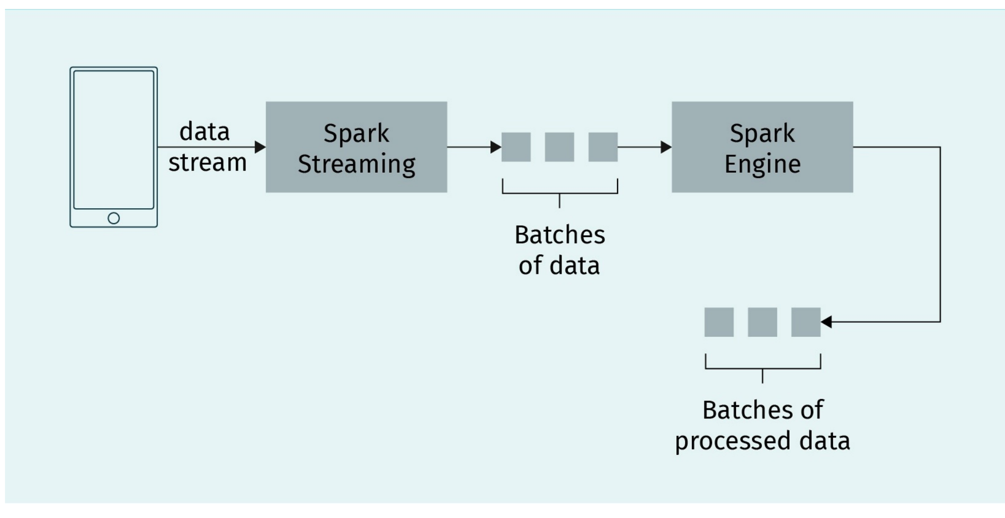
Figure 21: Spark Streaming



Source: Alavirad, 2020.

The data processing flow with Spark Streaming is as follows: Spark Streaming receives data streams from input sources and breaks the data into batches. Then, the Spark engine processes the batch data and generates a stream of batch-processed data.

Figure 22: Spark Streaming Workflow



Source: Alavirad, 2020.

The latency of Spark Streaming ranges from milliseconds to a few seconds, compared to Kafka Streams, whose latency is less than milliseconds. However, Spark Streaming is more flexible regarding data sources. Spark Streaming can, for instance, be run on Amazon EC2 with Hadoop YARN. The upstream source of Spark Streaming could be HDFS, Apache Cassandra, or Apache Hive, for example.

SUMMARY

We have now learned about data processing systems and discussed two such systems in detail: batch and stream processing systems.

When delivering the processing results in real time is not a critical requirement, it is possible to process data periodically, like at the end of each working day. In this case, we collect data during a specified time period in a batch and then process the data batch in a single processing job without the direct interaction of the user. This is what we call batch processing. One application of such a processing approach is processing financial transactions (at a financial institute) at the end of each working day.

However, there are some business use cases where the client cannot wait for the delivery of process results, so the results should be delivered in near-real-time. Stream processing systems were developed for this purpose. Some applications of stream processing are monitoring a production line, geospatial data processing, and Smart Grid.

We have discussed four ancestors of modern stream processing systems, known as information flow processing systems: active databases, continuous queries, publish-subscribe systems, and complex event processing. We then discussed the paradigm of typical streaming processing systems by discussing data model, processing, and system architecture. Finally, we reviewed two modern implementations of streaming systems: Apache Kafka and Spark Streaming.

UNIT 3

MICROSERVICES

STUDY GOALS

On completion of this unit, you will have learned …

- what the monolith architecture of a software product is and what its drawbacks are.
- what the microservice architecture of a software product is and how it functions as a modern alternative to monolithic architectures.
- how to implement a microservice application in a software product.
- how to migrate from an existing monolithic architecture to a microservice architecture.

3. MICROSERVICES

Introduction

Microservice architecture is an approach in software design used to create a software system with, or split a software system into, a collection of small interactive services (microservice). Each service is

- loosely coupled with other services,
- deployable independently of other services,
- specific in its functionality,
- developed and maintained by a small development team, and
- highly scalable, maintainable, and testable.

Unlike this, in a monolithic architecture, the whole software is made of multiple entangled modules, all contained in a single repository or file (for example, a .JAR or .EXE file). In this section, we will discuss monolithic architecture and its drawbacks briefly, followed by an introduction to microservice architecture as an alternative. Finally, we will discuss how to implement a microservice application.

In this unit, we will take a fictional food delivery application called FoodYNow as our example. Customers of this application can order meals from a nearby restaurant, which will then be delivered by app's local e-bikers. Therefore, this application has three types of users: customers, restaurant owners, and e-bikers.

3.1 Introduction to Monolithic Architecture

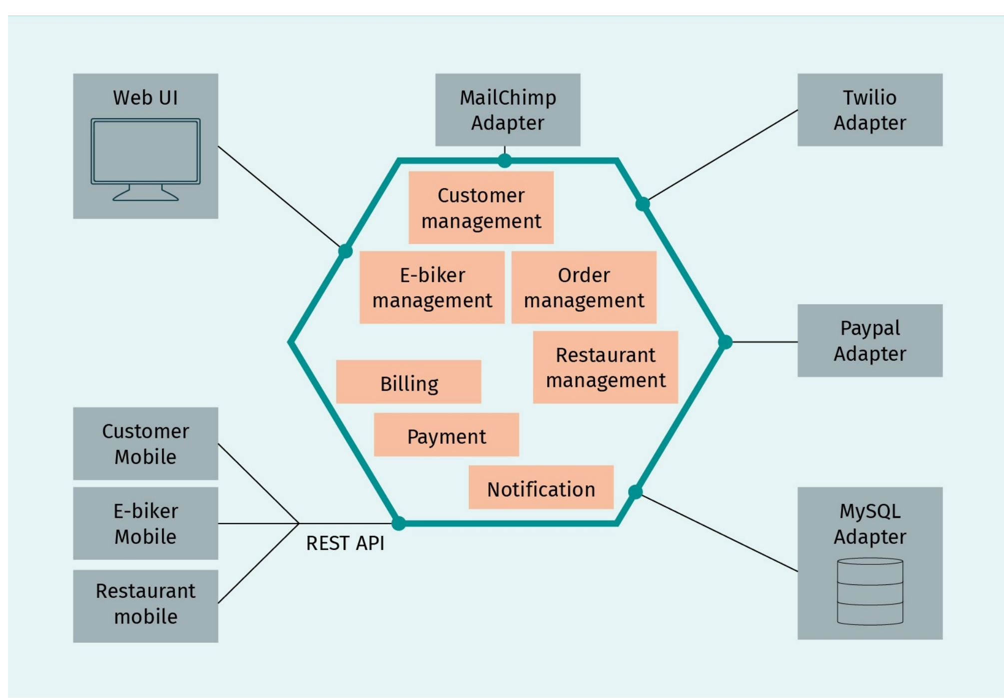
Before starting with the microservice architecture concept, it is advantageous to have a look at traditional monolithic architecture. After a brief introduction to this architecture, we will discuss the drawbacks it has, which eventually led to the introduction of a novel alternative (microservice architecture).

To explain this architecture, consider the FoodYNow application mentioned in the introduction. After deciding on the business concept of this application, the development team should start by designing, engineering, and coding the project. Following the traditional software design school, this application could consist of different modules all combined into a single program. These building block modules have different responsibilities, such as authorizing access to the user's account, managing HTTP requests, responding to the HTTP request via HTML or JSON/XML messages (presentation module), managing data of different entities (customers, restaurants, e-bikers, etc.), and delivering notifications. A simplified version of this proposed FoodYNow architecture can be seen in the figure below. At the heart of the FoodYNow application is the business logic, which consists of

modules responsible for different services, domain objects, and events. We have a module responsible for “Customer” management, a module responsible for “Order” management, a module for “Restaurant” management, and a module for “E-biker” management. There are also modules for “Billing,” “Payment,” and “Notification.”

These modules exchange data with the external world entities such as mobile clients and databases using the relevant connectors. For example, a MySQL adapter exchanges data with the SQL database, the REST APIs communicate with the “Customer,” “Restaurant,” and the “E-Biker” mobile devices, WEB-UI communicates with other web applications, the “PayPal” adapter makes communication with the PayPal payment platform possible, and the “MailChimp” and “Twilio” adapters are responsible for “Notification” management.

Figure 23: Monolithic Architecture of FoodYNow Application



Source: Alavirad, 2020.

In this architecture, all modules are packed in a single unit of a program that reminds us of a monolith: a single and very large organization that is very slow to change (Oxford Dictionary, n.d.). A monolith application is self-contained: all components (modules) required to run the application are integrated into the application code. After finishing the application coding, all developed modules are integrated (built) into a single file (such as a JAR, WAR, or EXE file). This single file is then deployed into a platform such as the Tomcat application server so that the application is made accessible to its users. Using such a singular, massive unit of code has some advantages, such as:

- Developing a monolithic application is reasonably straightforward, as normally there is a single language and a single platform to develop the software. At the beginning, it is also the most natural way to build an app, especially when with a small team. Therefore, such an application can be developed using a team of developers who are experts in that specific language and platform.
- As there is a single unit of the program, the deployment of monolith applications is also fairly simple.
- A monolithic application is self-contained, a characteristic that makes testing such an application less complicated, as there is normally no required external dependency to execute the test.
- The application being self-contained also makes management of the application easier.
- To some extent, a monolithic application is also scalable. In the scaling process, different copies of the application will be run on different nodes (if the application has been developed as a multithreaded application) and a load manager will manage the traffic among the nods to improve reliability and availability.

Let’s assume that your monolith FoodYNow application fascinates a lot of customers and the number of users grows exponentially after a year. This means that, sooner or later, you will need to improve your application by adding extra features and adapt its performance to match the growing load. After a year or two, your small and simple application will have turned into a large, complex, complicated application.

With its current monolithic design, your application has some drawbacks:

- Further development of a large and complex monolithic application is challenging, as it becomes difficult for any individual software architect and developer to understand the whole code (this applies especially to new members of the development team). As a result, the development process will slow down and the modular characteristic of the software will perish. Together, these obstacles will result in a decline in software quality over time.
- An increasing software size (with respect to the code) will overload the Interactive Development Environment (IDE), which will slow down the development process.
- The web container will overload as the size of the application increase, which will result in longer start-up times. Therefore, if the developers have to restart the application’s server several times per day, a majority of their time will be wasted waiting for the application to start up.
- Another drawback of a giant monolithic application is its continuous deployment. In this scenario, to update a single module of the application, developers have to redeploy the entire application. This process will interrupt the background jobs, for example, the **Quartz** jobs in Java applications. This could result in software failure. Therefore, developers might be very reluctant to implement frequent updates.
- A massive, monolithic application can only be scaled up one-dimensionally. To understand one-dimensional scaling, consider a monolithic application, where one of its modules implements CPU intensive image processing logic and another module is an in-memory database. As these two modules are integrated into a single application, it is not possible to scale the resources of each module separately (CPU for module A or

Quartz
A job scheduling library for Java applications, Quartz is used for organization-class applications to support process workflow and system management actions, as well as providing timely services.

memory for module B). The only scaling option here would be scaling all resources one-dimensionally (i.e., keeping memory and CPU together for all modules, irrespective of whether all modules need those extra resources).
- As all modules of a monolithic application are running within the same process, a fault in a single module, such as a memory leak, could result in the whole application experiencing a software failure.
- Let us assume that the monolithic software has been designed for a specific framework. It would be extremely resource and time-consuming to redesign it for a modern framework, as all of the software would have to be rewritten from scratch.

Therefore, monolithic application architecture is not a sustainable method for developing modern, dynamic applications which need to improve and change scale frequently throughout their life cycles. In the next section, we will introduce the microservice architecture pattern as an alternative approach to resolving the aforementioned problems.

3.2 Introduction to Microservices

As we have learned in the previous section, we encounter some difficulties and complexities when dealing with a massive size monolith application, such as scalability, reliability, and so on. To overcome these problems, you can develop the application as a set of loosely coupled services that communicate with each other through APIs—these services are known as microservices. Each microservice is an individual micro-application with a very specific functionality, such as customer management or order management. To communicate with other microservices in the application, and also with the application's external clients, each microservice provides an API.

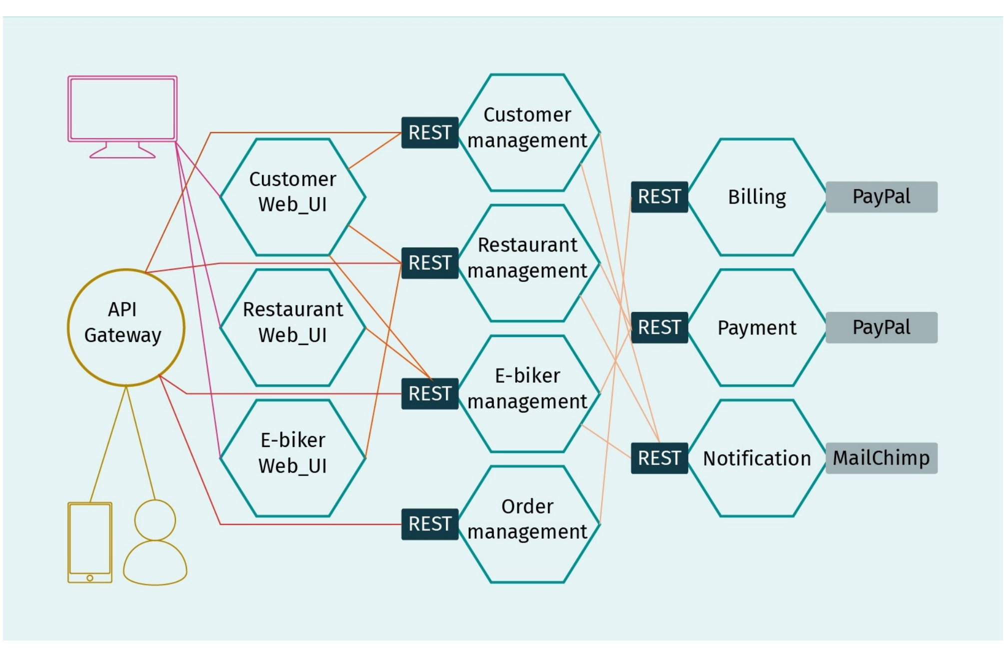
The microservice architecture approach is based on a combination of some proven concepts, such as (Amazon, 2020o)

- agile software development,
- service-oriented architecture,
- API-first design, and
- continuous integration/continuous delivery.

This architecture has been implemented by Amazon, Netflix, and eBay. In the figure below, the microservice architecture variant of the FoodYNow application has been depicted. With this novel design, instead of a big business logic which contains all functionality of the application, we have small microservices interconnected with APIs. For example, one microservice manages the e-bikers who are responsible for picking up the food from restaurants and delivering it to the customers. This service uses the notification microservice to inform an available e-biker about a pick-up from a nearby restaurant. Each backend microservice (such as "Customer management," "E-Biker management," and so on) is provided by an API to communicate to other microservices or Web UIs. There are also some adapters to connect to external services, like the "PayPal adapter" for the "Billing" and "Payment" microservices. The API gateway is responsible for communication between the external entities (e.g., customers', e-bikers', or restaurant owners' mobile clients) and the

platform, as the client applications do not have direct access to the backend microservices. This component is also responsible for load balancing, monitoring, and authorized access. We will discuss the API gateway in more detail later in this unit.

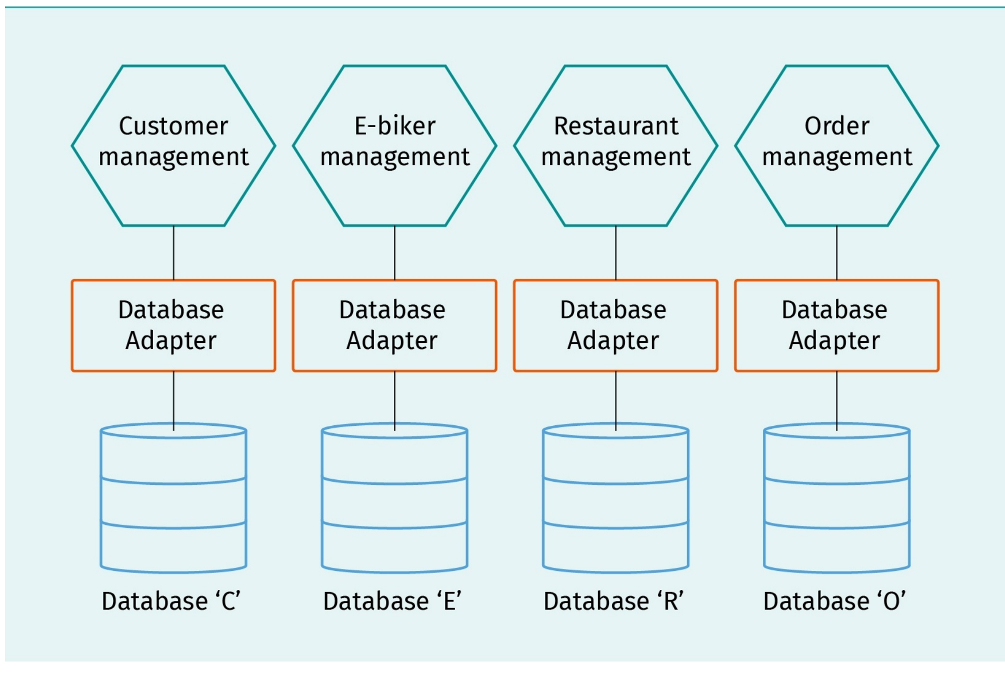
Figure 24: Microservice Version of the FoodYNow Application



Source: Alavirad, 2020.

To achieve a loose coupling between microservices, each is provided with a database and a database schema. In the figure below, the database of each microservice for the FoodY-Now application has been shown. Each microservice uses a suitable database and database schema that considers its functionality and requirements. We will discuss the data consistency approach between these databases in more detail later.

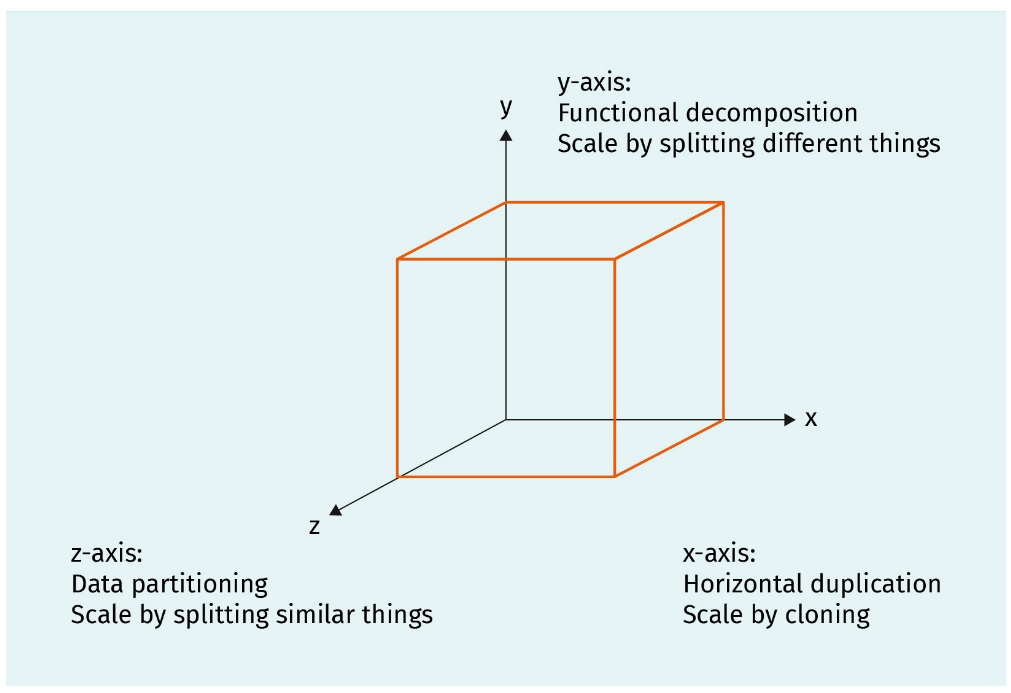
Figure 25: Microservices and Databases



Source: Alavirad, 2020.

If we consider the scale cube, introduced by Abbott and Fisher (2015), the microservice architecture is equivalent to the y-axis scalability, i.e., functional decomposition. In this cube, x-axis scaling corresponds to running multiple copies of the same application on multiple nodes when the loads are monitored and managed by a load balancer. The z-axis scaling is similar to the x-axis scaling; multiple copies of the same application run on multiple nodes, but each node is responsible for a specific portion of the data.

Figure 26: Scale Cube



Source: Alavirad, 2020, based on Abbott & Fisher, 2015.

Advantages of Microservices

Because every single microservice is much smaller in size than a massive monolithic application, it is easier to understand the microservices and to apply relevant modifications, to fix microservices bugs, and to improve them. In other words, such an application is more maintainable and extendable. Each microservice can be tested quickly, as there is a loose decoupling between services.

Unlike monolithic applications, each service can be deployed independently, which inspires developers to implement more changes. Developers also need not coordinate local change deployments with other microservices' teams. The software project can be broken down into small batches (each includes one or several services), where each development team is responsible for its batch. Therefore, the development process can be accelerated.

Also unlike in monolithic applications, the IDEs are not overloaded, so the development process will not be slowed down. When there are some bugs and faults with a microservice, it is easier to track and isolate those faults. It is also easier to migrate to new technologies, as it is possible to upgrade each microservice independently, unlike with the modular structure of monolith architecture, where all modules should be upgraded simultaneously.

Disadvantages of Microservices

Managing and testing the intercommunication between different services is not always straightforward. This intercommunication can be accomplished, for example, by messaging or **Remote Procedure Call (RPC)**. Designing functionality that spans multiple microservices is ambiguous. Distributed databases are another source of complexity in a microservice architecture. It is very common to have transactions that update multiple business entities. In the monolithic applications, this is quite clear as there are only a few (sometimes only one) databases and only these databases should be updated. In the case of microservice architecture, however, each service has its database and the transaction should follow the **CAP theorem**.

Remote Procedure Call
An RPC is a method of interprocess communication for distributed client-server based applications. In this approach, the invoked procedure is not required to exist in the same address space as the calling procedure.

CAP Theorem
The CAP theorem in distributed data stores states that such systems can only satisfy two of the the three following criteria: availability, consistency, and partition tolerance.

The testing procedure of microservice applications could be more difficult. For example, testing the REST APIs of a web application design based on a monolithic application with an automated test procedure framework like Spring Boot is fairly simple by starting up the application inside this framework. However, in the case of the microservice application, the associated microservice of the under-test API and all dependent microservices should be executed.

Complicated deployment is also another disadvantage of microservice applications. In the case of a monolithic application, multiple copies of the application run on several servers where each copy is identified by the host server locations. However, each microservice application consists of many microservices. For example, Netflix has over 500 microservices (SmartBear Software, 2015), and each of these services have several runtime instances. Add to this complexity a mechanism to discover the location of the required services by internal services or external applications. Therefore, a successful deployment should be as automated as possible, for example, CloudFoundry provides an automated platform for deploying microservices.

3.3 Implementing Microservices

Having discussed the basics of microservice architecture, we will now consider different aspects regarding implementing microservices. We will look at the following topics:

- client-microservice communication
- microservice intercommunication
- service discovery mechanisms in a microservice application
- distributed data management in a microservice architecture
- deployment strategies of microservices
- migration from a monolith application to a microservice application.

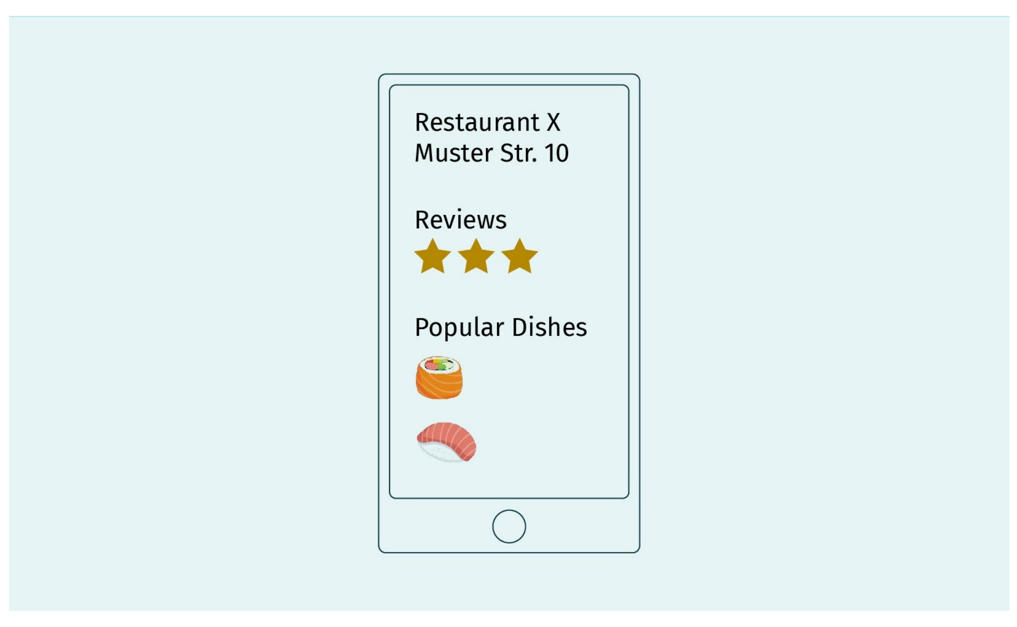
Client-Microservice Communication

In a monolith application, the external client-application communication is almost unambiguous: there is only one set of endpoints (application replications) that are assigned or loaded by a load-balancer component. However, in a microservice architecture, each

microservice provides a set of endpoints, as each microservice is a mini-application. To make this point clearer, consider the FoodYNow mobile application. On a specific restaurant page inside the customer mobile application (an external client), there is a collection of information such as

- general information about the restaurant (e.g., name, type, address, and opening hours)
- customer reviews of popular dishes
- current special offers

Figure 27: Mobile Client Application of FoodYNow



Source: Alavirad, 2020.

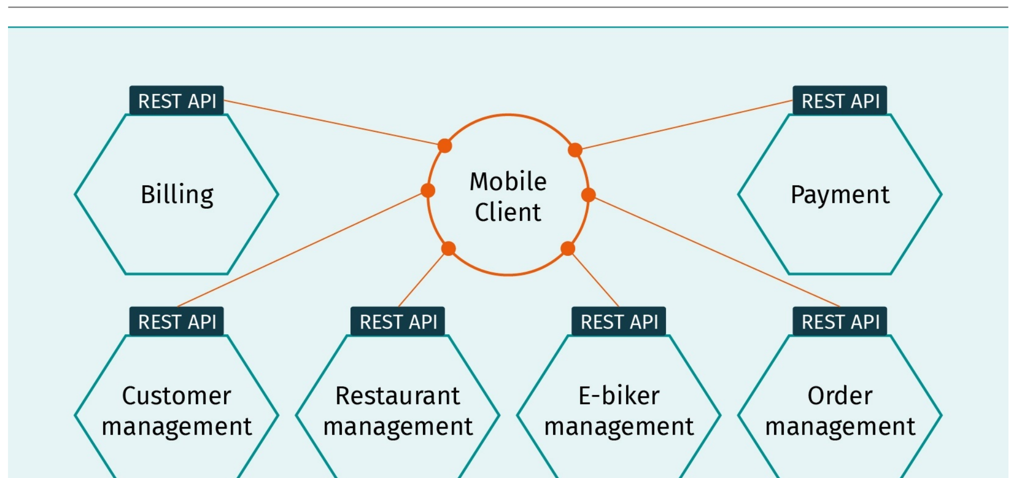
Assume this application has been designed in a monolithic architecture. To retrieve all this information about the restaurant from the database, the mobile client requests a single REST call to the application:

```
GET api.foodynow.de/restaurantsdetail/restaurantID
```

The request will be delivered by a load balancer to one of the several running copies of the FoodYNow application.

However, if we want to design this application based on the microservice architecture, we should have a microservice for almost every information unit displayed on the client mobile application and send a REST request to each of them. In this scenario, a load balancer makes direct communication between the mobile client and the application possible. Such an approach has some drawbacks, like microservices factoring in future development (merging or splitting microservices). It also increases the chance of a client-microservice protocol mismatch.

Figure 28: Mobile Client Application Direct Communication with a Collection of Microservices



Source: Alavirad, 2020.

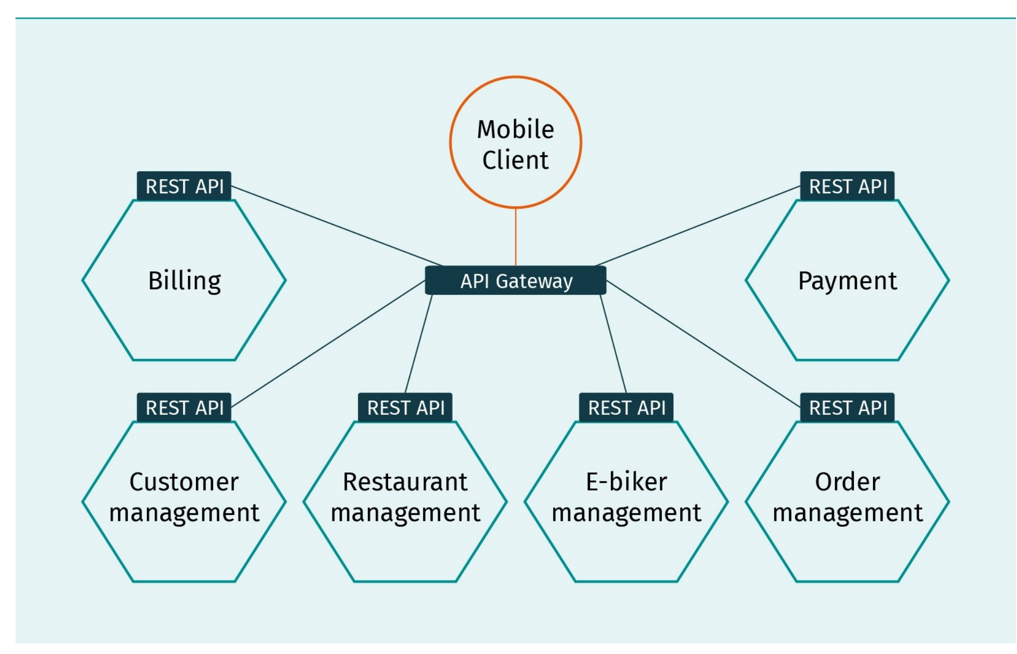
A more efficient approach is to instead design a server as a single entry point to the micro-service application, i.e., the API gateway. This component hides the complexity of the internal system of the application from the (mobile) client with tailored APIs for each client. The API gateway also plays the role of a protocol translator, request router, balancer, and combiner. Therefore, in this scenario, there is no direct external client-microservice communication: every single request from external client applications will be directed to the API gateway, which then redirects the requests to the relevant microservice or a group of microservices.

For example, if the FoodYNow application is designed based on the microservice architecture below, a single request by the client application, for example,

```
GET api.foodynow.de/restaurantsdetail/restaurantID
```

will be sent by the mobile client application to the API gateway. This request will be decomposed by the API gateway into several requests to the individual micro-services, for example "restaurant basic info" microservice, "customer reviews" microservice, and so on. The responses will then be combined in a single response and sent to the mobile client. In this case, the API gateway will also translate the incoming requests from web-friendly protocols of the mobile client, such as HTTP, into less web-friendly intercommunication protocols of the microservices.

Figure 29: API Gateway in Microservice Architecture



Source: Alavirad, 2020.

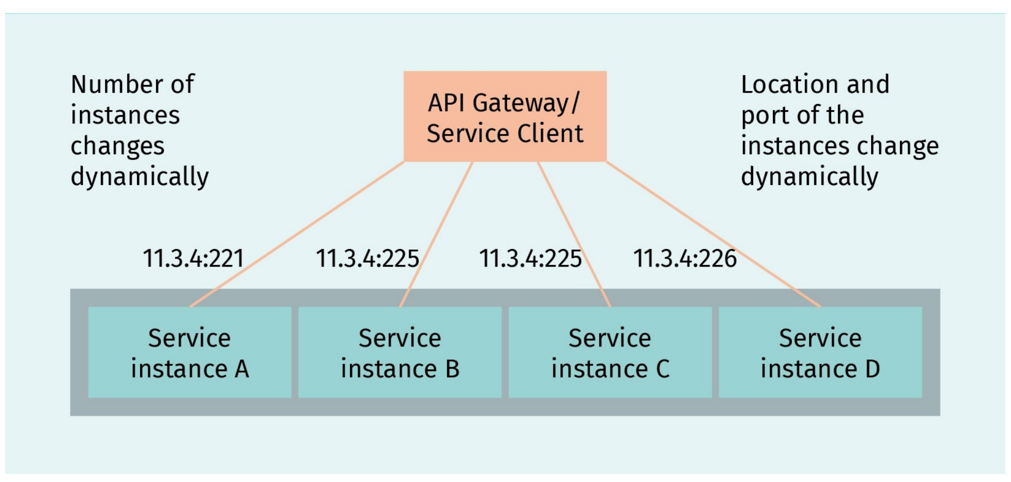
An example of an API gateway for microservices is Netflix Zuul (Netflix, 2020c). This application is the front door to all Netflix's server infrastructure and microservices (Netflix, 2016). Zuul dynamically routes the client requests from different devices such as mobile phones and tablets to the relevant Netflix microservices. It also provides Netflix's developers with services for testing and debugging new services, system health monitoring, and helps protect the infrastructure from cyberattacks. Zuul also balances the traffic among multiple Amazon Auto Scaling groups.

Zuul has other functionalities, such as load shedding: it drops requests that exceed the predefined load capacity. This component also tests a server by adding load gradually to it to gauge its performance (stress testing).

Service Discovery

Unlike with traditional applications, where the network location of services that are running on physical hardware is relatively static, with modern cloud-based applications the network location of applications is dynamic. In the case of microservice applications, the number of microservices also changes dynamically. For instance, the Amazon EC2 Autoscaling component changes the number of service instances depending on the load. Therefore, finding the relevant service is a challenge in implementing microservice applications.

Figure 30: Service Discovery Problem



Source: Alavirad, 2020.

There are two approaches to discovering service locations, including IP address and port (Richardson, 2020):

Client-side discovery: in this approach, the client (client application or API gateway) is responsible for discovering the location of services inside the system by requesting a **service registry**. The service client then uses a load balancer approach to deliver the request.

Service registry
A service registry is a database with the information about the available services, including their netwok locations and ports.

One benefit of this approach is fewer moving parts relative to other discovery approaches, as we will see in the rest of this section. However, there are also drawbacks with this approach:

- In this approach, the client has to be coupled tightly with the service registry.
- For each programming language used by the client, such Java or Scala, a client-service discovery logic must be developed.

Figure 31: Client-Side Discovery of Microservices

Source: Alavirad, 2020.

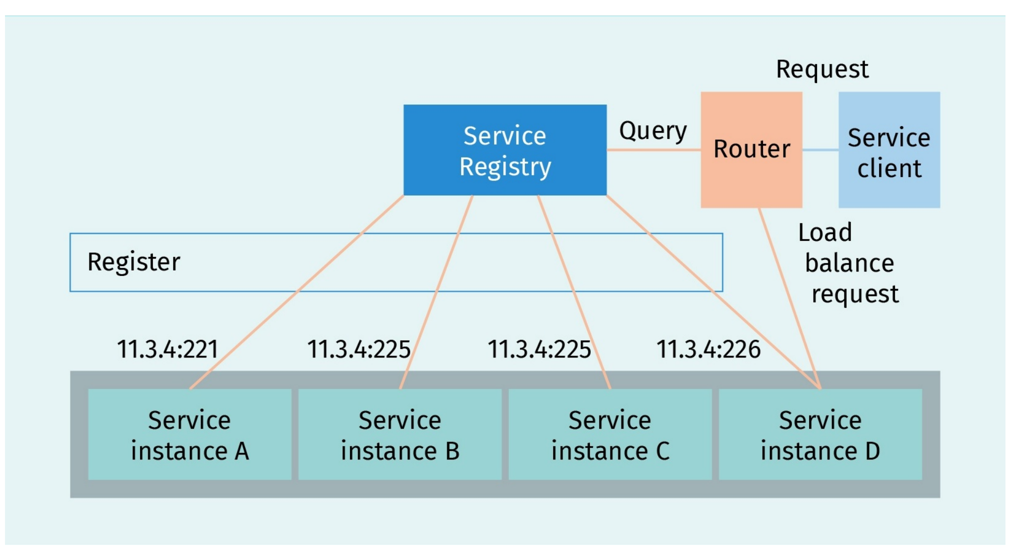
A client-side service discovery could be developed, for example, using Netflix Eureka (Netflix, 2020a) and Netflix Ribbon (Netflix, 2020b). Netflix Eureka is a REST-based service registry that provides an API for managing service-instance registration and a list of the available services. Ribbon is an HTTP client that routes HTTP request to the relevant services by querying to Eureka. Ribbon reads the status attribute of the available microservices from the Eureka server and assigns only the instances with an UP status for load balancing.

Figure 32: Designing a Client-Side Service Discovery Using Netflix Eureka and Netflix Ribbon

Source: Alavirad, 2020, based on Netflix, 2020a; 2020b.

In a server-side discovery approach, the client (e.g., API gateway) sends a request via a router to the service registry. After receiving the response from the service registry about the relevant service, the router sends the request via load balancer to that service.

Figure 33: Server-Side Discover Approach



Source: Alavirad, 2020.

In this mechanism, the client code is simpler compared to the client-side discovery, as the client is not engaged with the service discovery task. Some cloud platforms such as AWS also provide similar server-side discovery routers such as the Elastic Load Balancer (ELB). The drawback of this approach is that there are more moving parts (router), which all require development and configuration. The router must also support the required communication protocols (e.g., HTTP, or **Thrift**).

Thrift
A lightweight, language-independent software stack, Thrift is used for point-to-point remote procedure call (RPC) implementation.

An example of a server-side discovery router is the AWS Elastic Load Balancer (ELB). The client application makes an HTTP or TCP request to the ELB and it distributes the load among a set of registered Elastic Compute Clouds (EC2) instance. The service registry is integrated within ELB.

In both cases discussed above, a service registry is an important element of the service discovery mechanism. A service registry is a database of available services, their locations, and also instances of services. When a service starts up, it is registered on the service registry and when it shuts down, it is deregistered from the service registry. The registration could be accomplished in two different approaches:

- With the self-registration approach, the microservice registers itself by the service register.
- With the third-party registration approach, a third party component is responsible for registering and deregistering microservices.

Microservice Intercommunication

In this subsection, we will discuss the intercommunication mechanism between microservices in a microservice application.

In a microservice architecture, each service is a process that runs on a specific host (often in the cloud). Therefore, an inter-process communication (IPC) mechanism must be designed to manage the intercommunication of services. We will discuss four such IPC mechanisms.

Synchronous messaging

In this approach, the client uses request/response protocols such as REST and Apache Thrift to request a service (Raj et al., 2017). In many cases, the client blocks a thread until it receives a response from the server. This approach has benefits like the simplicity of request/response protocol implementations. However, it has several drawbacks. In this approach, we have a lower system availability, as the client and the server are blocked during the IPC messaging process. The service discovery for this approach is client-side discovery.

Asynchronous messaging

There are different types of asynchronous messaging mechanism:

1. Request/asynchronous response: In this mechanism, a client sends a request to a server and expects to receive the response: not promptly, but eventually, at some point in the future. Therefore, the client does not block the server, unlike the synchronous messaging mechanism.
2. Notifications. In this mechanism, a sender sends one message to one or several receivers and does not expect any response from receivers. The receiving channels for notifications could be SMS message, e-mail, REST callback, AMQP, MQTT, and so on (EdgeX Foundry, 2020).
3. Publish/subscribe. A publisher publishes a message to zero, one, or many subscribers.

The asynchronous IPC mechanism has several benefits, such as higher availability because the message broker buffers message until they are requested by the subscriber/receiver. The drawback of this mechanism is higher complexity compared to the remote process call approach. Apache Kafka is an example of an asynchronous messaging system.

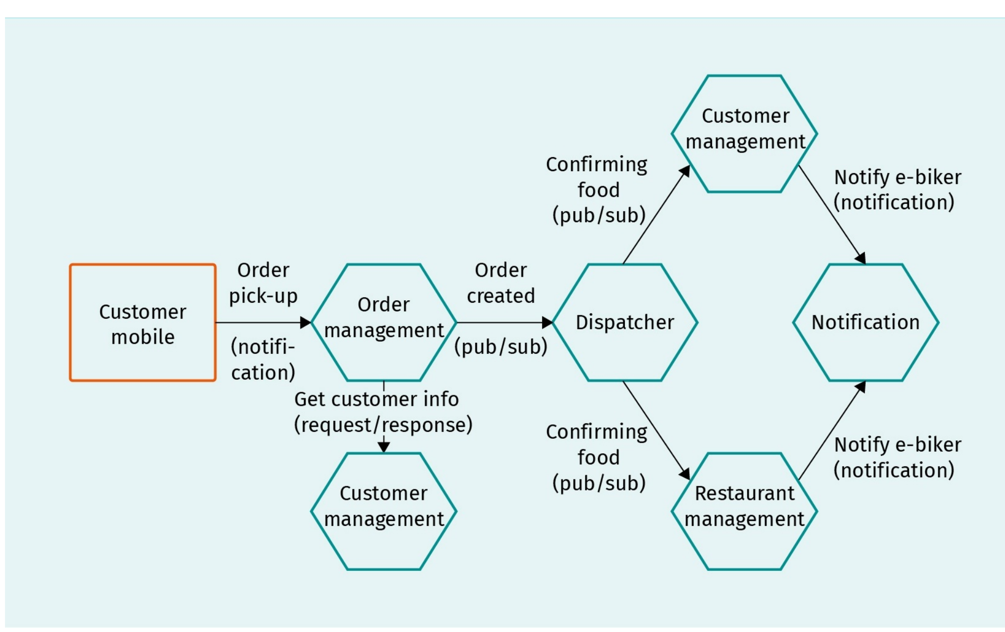
Domain-specific protocols

It is also possible to use some domain-specific protocols like e-mail protocols (e.g., IMAP) or media streaming protocols (HLS, HDS, etc.) to fulfill the communication between microservices.

In the figure below, the intercommunication between some of the microservices in the FoodYnow application have been shown. The customer mobile application sends a `notification` to submit an order which is collected (picked up) by the “Order manage-

ment" microservice (through the API gateway). The "Order management" microservice `requests` customer information from the "Customer management" microservice and after receiving the `response` it `publishes` a created order which is received by the "Dispatcher" microservice (Richardson & Smith, 2016).

Figure 34: Intercommunication between Services in FoodYNow Microservice Application



Source: Alavirad, 2020, based on Richardson & Smith, 2016.

The development of the API services depends on IPC mechanisms. For example, when we use RPC mechanism and using HTTP, the API should include URLs and when we use asynchronous messaging, the API should contain the message channels and the message types.

Distributed Data Management

To ensure the highest decoupling degree, it is necessary to encapsulate the data of each microservice privately and make the data accessible only through APIs. Besides, as each microservice inside the system has different functionality, it is very common that microservices have different types of databases and database schemas. For example, the "restaurant basic info" service may use a MySQL relational database, while the "trip" microservice uses a NoSQL database like Neo4J.

Therefore, it is very probable that a microservice application uses different types of databases. Such a data system architecture creates some challenges regarding data management.

For example, let us imagine that, on the FoodYNow application, a customer can use the bonus (credit) on his account to order food on the platform. The "Customer management" microservice contains information about the customer and the "Order management"

microservice contains information about the order. The “Order management” microservice should check if the cost of the order does not exceed the customer’s bonus. The problem here is that the data of each microservice is private and, for example, the “Order management” microservice has to request access to the “Customer management” microservice’s data through an API. It cannot access this information directly.

Another challenge is querying data from multiple microservices. In the previous example, we looked at retrieving data from “Customer” and “Order” microservices. If the “Customer” microservice uses a SQL database and the “Order” microservice uses a NoSQL database, the data retrieving process could be also challenging (because of the possible aggregation of different types of data).

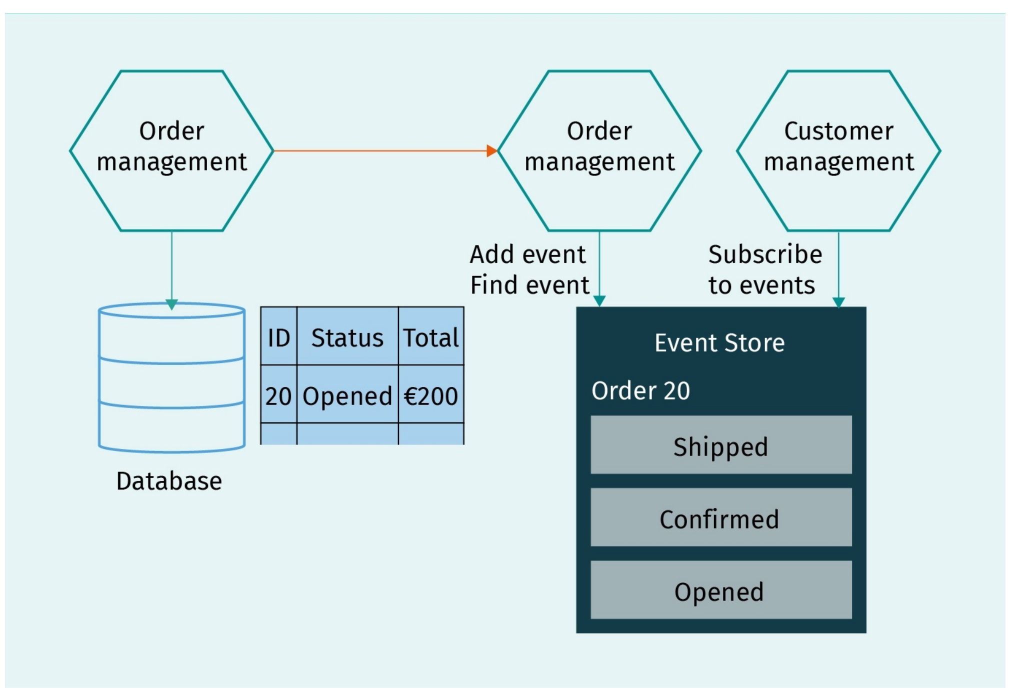
Message broker
A service that helps other services within a microservice architecture to perform intercommunication via messaging is called a message broker.

One approach for overcoming these data management problems, i.e., data consistency and querying data across multiple independent and different databases, is to use an event-driven architecture. In this approach, a microservice publishes a message to a **message broker**. When some pre-configured event happens, for example, a customer places an order (this intercommunication mechanism is different from what is presented in the previous figure), the “Order management” microservice publishes an event (`order_created`) to the message broker containing information about the order (`customer_ID`, `total_amount`, etc.). The “Customer management” microservice, which has been subscribed to the message broker, consumes this `order_created` event and checks if the `customer_bonus` does not exceed `total_amount`. If not, the “Customer” microservice publishes a `bonus_reserved` event to the message broker. The “Order” microservice uses this event to change the `order_status` inside its database from `new` to `open`. Such a transaction does not guarantee an ACID transaction, but does guarantee some weaker constraints like eventual consistency. In eventual consistency, if there are no new updates to the system, all accesses eventually will return the last updated value (Vogels, 2008).

Atomicity of transactions in an event-driven data management architecture is another fundamental challenge. Consider that the “Order management” microservice updates the `order_table` with a new row containing the information of a new order. However, the system crashes just before the “Order management” microservice could publish the `order_created` event into the message broker, which leads to the data inconsistency of the data system.

Event sourcing is an approach to achieve atomicity (Event Sourcing, 2015). The fundamental concept in event sourcing is as follows: instead of persisting the current status of a business object, a microservice persists the object’s state-changing events. The event-driven application will reconstruct the current status of the object (for example an order), by replaying the sequence of events. When an event happens, this event will be appended to the list of events. As inserting a single entity (new entity) is a single operation, this approach is atomic. For example, the “Order management” microservice updates the `Order_20` table inside the event store by inserting a new row with values like `order_created`, `order_confirmed`, `order_shipped`, etc. The event store is a database containing the events data and provides APIs for updating or retrieving the data. This component also plays the role of the message broker in this example.

Figure 35: Event-Sourcing Approach to the Atomicity of Event-Driven Applications



Source: Alavirad, 2020, based on Event Sourcing, 2015.

Microservice Deployment Strategy

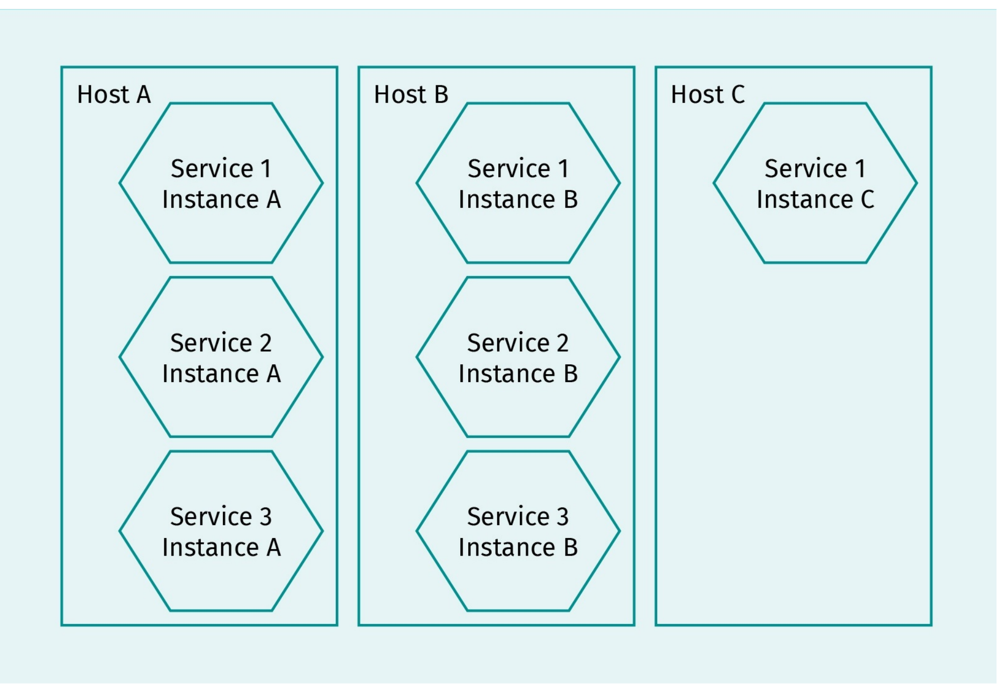
We will now discuss some microservice application deployment strategies. The deployment of monolith applications is simple: you deploy several monolithic application instances across a suitable number of servers.

In the case of microservices applications, each microservice is a micro-application that should be started with the required resources, such as CPU and memory. In addition, the number of running copies of a microservice is not constant, as it is a function of the demand and is managed by a load balancer. In the following, we will discuss some deployment strategies for microservice applications.

Multiple microservice instances per host

In this approach, we run multiple instances of different microservices on a physical or virtual host. In this strategy, the engineers can deploy each microservice as a single process, for instance a Java virtual machine (JVM) process. It is also possible to group multiple microservices in a single JVM process.

Figure 36: Multiple Microservice Instances per Host



Source: Alavirad, 2020.

There are two benefits to this approach:

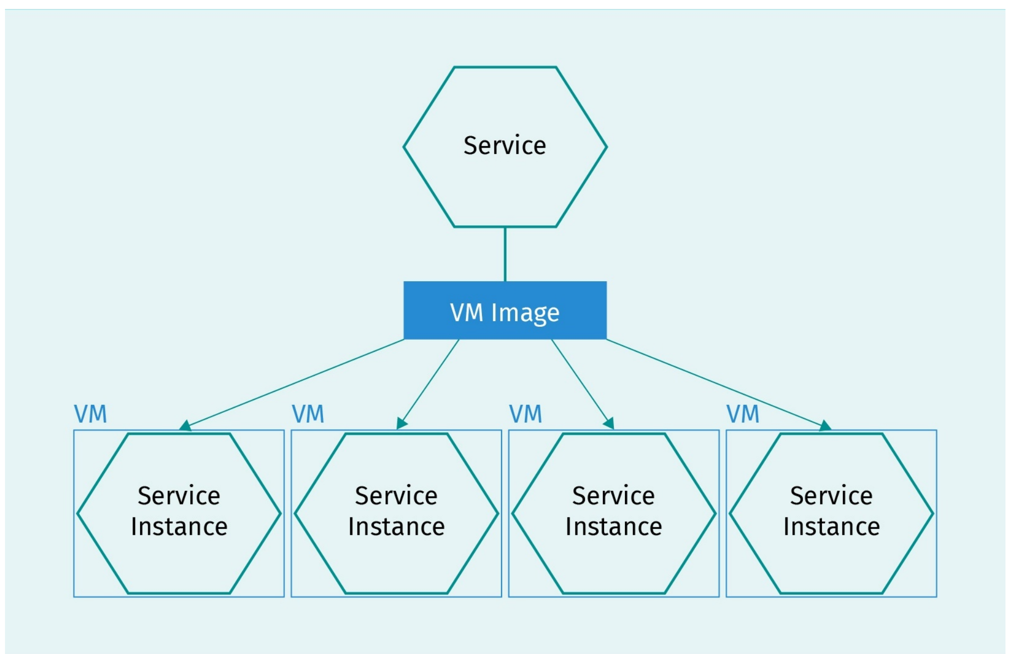
- As multiple service instances share the same server, the resource usage is economical.
- The deployment is fast, as it is only required to copy the microservice (e.g. .JAR or .WAR files in the case of Java applications) into the host and start it.

The main drawback of this approach is the lack of isolation of the microservice instances. Service isolation can only occur in the case of one process per microservice. Otherwise, there is little or zero isolation when multiple microservices are grouped in a single process. For example, when multiple Java microservices share the same JVM heap, a malfunctioning microservice could cause the failure of all other microservices running in this JVM. Besides, if multiple microservice instances are grouped in a single process, the resource management of each microservice instance is challenging.

Single service instance per virtual machine

In this approach, each microservice will be deployed into a single and isolated virtual machine such as an Amazon EC2 AMI. This approach has been used by Netflix to deploy its video streaming microservices. Multiple platforms provide tools for building virtual machines like Packer (2020) and Boxfuse (2020).

Figure 37: Single Service Instance per Virtual Machine



Source: Alavirad, 2020.

This deployment approach has several benefits such as:

- complete isolation of the microservices
- benefits from the virtual machines cloud platform infrastructure such as load balancing and auto scaling of AWS
- easier deployment because, after packing the microservice as a virtual machine, the VM's management API becomes the deployment tool and the service itself becomes a black box

The drawback of this approach is the wasting of resources. If we overestimate the required resources by a microservice, the extra resources will be wasted, as there are no other processes to use these extra resources. In addition, most VM cloud providers charge you for each VM, irrespective of its status (active or idle).

It is also possible to use a container to deploy microservices. A container virtualizes the operating system (virtualization at the operating system level) in such a way that multiple workloads can run on a single operating system (unlike the virtual machines that virtualize the hardware to run multiple instances of the operating system) (Chamberlain, 2018). In this model, we pack the microservice in a container image consisting of the application and the libraries required to run the application.

Serverless deployment

The final strategy we will discuss is serverless deployment. In this approach, we use the deployment infrastructure of a service provider like Amazon Lambda (Amazon, 2020n). In this scenario, we copy the ZIP file of the microservice together with the required metadata and the infrastructure of the service provider runs the microservice.

The infrastructure uses virtual machines or containers to isolate microservices. There are different examples of serverless deployment infrastructures like Amazon AWS Lambda, Google Cloud Functions (Google, 2020i), and Azure Functions (Microsoft, 2020f).

An AWS Lambda function is a stateless component that is used to handle events. To create such a function, the engineer packs the application written in Java, Python, or NodeJS in a ZIP file and uploads it on the AWS Lambda. Then when an event is published, the AWS Lambda runs an idle instance of your Lambda function to handle the event. AWS Lambda also runs enough instances of the Lambda function to handle the load reliably.

There are four methods to call a Lambda function:

1. Configure the Lambda function to be called when an event is created by an AWS service like S3, Kinesis, and DynamoDB
2. Configure the AWS Lambda Gateway to direct the HTTP request to a specific Lambda function
3. Call the Lambda function explicitly using the AWS Lambda Web Service API
4. Configure the Lambda function to run periodically

Some of the benefits of a serverless deployment are:

- There is no infrastructure to manage or maintain.
- It is easy to deploy.
- You can pay per request.
- It is automatically scalable.

The drawbacks of a serverless deployment are:

- Constraints and limits are imposed by the infrastructure provider.
- Often, only certain programming languages are supported.
- AWS Lambda is restricted to limited input sources.
- Unless lightweight microservices are used, the service's response time won't be acceptable.

Migration from Monolithic to Microservice

The last part of this section will discuss different strategies to turn your existing monolithic application into a modern microservices application. Of course, we will not discuss rewriting a monolith application here, as this approach is not a migration, but a new development and design procedure.

Strategy A

The most straightforward strategy is to develop new features of an existing monolith application in the form of microservices. These new microservices coexist with the main monolith application. This strategy stops the monolith application from growing. A gateway component is required to route the requests to the newly developed microservices or the old monolith application. To make the direct communication between the monolith part and new microservices, another component known as glue code is required. This component could be a part of the monolith application or the microservice and its main role is data integration as the monolith component needs the data from microservices and vice versa. Although this strategy helps to prevent the monolith application from being unmanageable, it cannot resolve the existing problems of the monolith application.

Strategy B

We can generally split any existing monolith application into three components:

1. The presentation layer, which is responsible for communication to the external world by handling HTTP requests using a REST API or an HTML-based web UI
2. The business logic layer, which contains the main code and functionalities of the application
3. The data-access layer, which is responsible for accessing the infrastructure components, e.g., databases

In this strategy, we separate the presentation layer from the business and data access layers and make their communication possible through a REST API. In this strategy we split the monolith application into two smaller components, where each of them can function like a microservice. After splitting the main monolith application into two smaller components, the developer team can develop, maintain, scale, and deploy each component easier. The REST API component developed for communication between components also could be used by newly developed microservices following strategy A. Again, this strategy is not the most ideal solution. We need another strategy to resolve the problems associated with our monolith application.

Strategy C

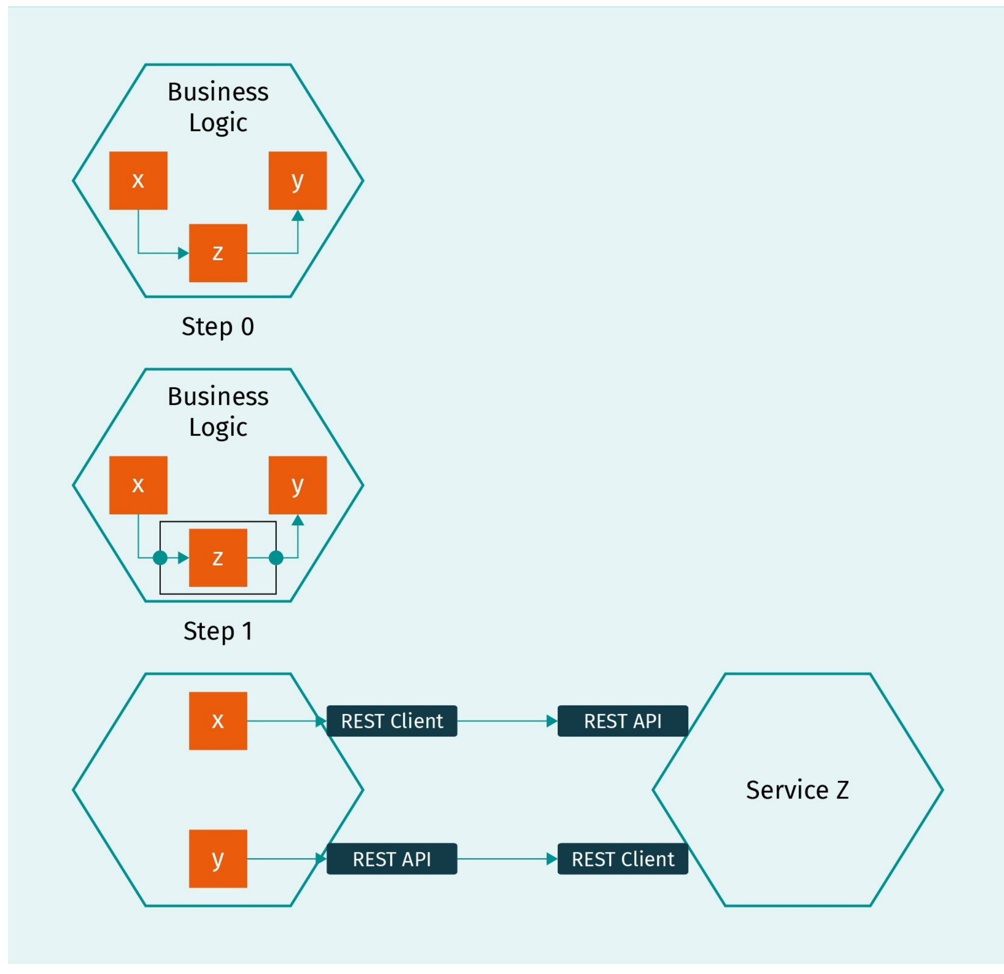
In this approach, we extract the modules of the existing monolith application and rewrite them into microservices. This strategy also shrinks the size of the monolith application and, in the end, we can transform our entire monolith application into a microservice application.

In this strategy, the main question is which module should we choose first to turn into a microservice. Here are some things to consider when choosing:

1. Choose a module that has a very different resource requirement pattern to the other modules, for example, a module that needs a cache or high computational resources. Isolating these modules is beneficial because, after isolating them, they can be scaled independently.
2. Choose a module that needs updating more frequently.
3. Choose a module that is loosely coupled with the rest of the modules in the monolith application.
4. Extracting a module that exchanges information in the form of asynchronous messaging with other components of the monolith application would be easier than extracting a module that communicates with synchronous messaging.

After choosing the right module to convert into a microservice, we should now define a coarse-grained interface between the module and the rest of the monolith application components. This process could be challenging, as there is a complex and tangled communication process between the module and the monolith application (as the example in the figure below shows). In this example, we have chosen to convert module Z into a microservice. Module Z calls module Y and is called by module X. In the first step, we define the coarse-grained interface between modules X, Y, and Z. We define an interface that is used by module X to call module Z and then we define another interface used by module Z to call module Y. Finally, we rewrite the module's code to be a standalone microservice. The communication will be realized by developing the relevant APIs, as shown below.

Figure 38: Choosing a Module from a Monolith Application to Convert into a Microservice



Source: Alavirad, 2020.

SUMMARY

In this unit, we learned about monolithic and microservice architectures. We started with a brief introduction to monolithic applications as a single body of code (such as a single .EXE file) which is made up of several modules, with each module having specific functionalities. We have shown that, as the size of the software grows, this architecture is not a sustainable design method for massive or complex software.

We then discussed microservice architecture as an alternative to monolithic architecture. A microservice application is a set of loosely coupled services that communicate with each other through APIs. We also discussed the advantages and disadvantages of this model, such as challenges with managing the distributed databases.

Finally, we discussed different topics regarding the implementation of microservices. Some aspects, such as client-microservice communication, microservices intercommunications, service discovery mechanisms in a microservice application, distributed data management in a microservice architecture, deployment strategies of microservices, and migration from a monolith application to a microservice application, were covered here.

UNIT 4

GOVERNANCE & SECURITY

STUDY GOALS

On completion of this unit, you will have learned …

- what data protection is.
- about the General Data Protection Regulation (GDPR).
- what the effect of the GDPR on data system design is.
- what system security is.
- some examples of security requirement engineering methods.
- ~~what data governance is.~~
- how data governance is used within organizations.

4. GOVERNANCE & SECURITY

Introduction

In this unit, we will introduce three measures regarding the data within an organization. To begin, we will discuss data protection, data regulation, and the rules regarding customer data collection and processing. In this section, we will focus primarily on the European General Data Protection Regulation (GDPR) as an example of a data protection act. Following this, we will discuss data security and how to secure collected data from unauthorized access and misuse. Finally, we will introduce the concept of data governance, a set of principles and practices used to support efficient performance in evaluating, generating, storing, processing, and deleting corporate data.

4.1 Data Protection

Data protection, which is also known as data privacy, is the process of protecting an individual's data from misuse by data collection and processing companies. This concept should not be confused with the protection of very sensitive and confidential data of an organization from unauthorized access, which is known as data security. Both data security and data protection are concerned with protecting data but focus on different types of threats. Data protection policies protect an individual's data against data collection companies, while data security measures protect companies' and organizations' data against unauthorized access and cyberattacks.

The main focus of data protection is giving an individual control over their private data (Kneuper, 2019). Data protection has stemmed primarily from the topic human rights, as is emphasized by Charter of Fundamental Rights of the European Union: "Everyone has the right to the protection of personal data concerning him or her" (European Parliament, 2012, Art. 8).

As cultural aspects also play a role in the creation of data protection measures and regulations, different data protection legislations can be found around the world. For example, there are the Singapore Personal Data Protection Act, the Indian Data Protection Law, the Asia-Pacific Economic Cooperation's APEC Privacy Framework (APEC, 2015), and, since May 2018 in the European Union, the General Data Protection Regulation (GDPR). In the rest of this section, we will focus on GDPR as an example of a data protection act.

General Data Protection Regulation (GDPR)

GDPR is the European Union regulation regarding the protection of an individual's (EU citizen's) data against data collectors within the EU, as well as the transfer of this collected data outside the EU's borders. The core concept of GDPR has been stated in article 5: "personal data shall be processed lawfully, fairly and in a transparent manner in relation to the data subject ('lawfulness, fairness and transparency')" (European Parliament, 2012).

In designing a data system that uses an EU citizen's data, engineers always should work closely with the legal department to design a product that is compliant with GDPR. Before starting to discuss how to develop a data protection-compliant software product, we should first define personal data as the main subject of data protection legislation. GDPR defines personal data as "any information relating to an identified or identifiable natural person" (European Parliament, 2012, Art. 4). We should therefore consider the email address of person X, the social security number of person Y, or the bank information of person Z as personal data which should be protected in data systems by applying GDPR measures.

Article 4 of GDPR distinguishes between three different entities that are involved in a data processing process:

1. The data subject is the identified or identifiable natural person who should be protected against data misuse. The data subject could be an individual website visitor, an employee of a company, or a customer of an online shop.
2. The data controller is the body that determines the purposes and means of personal data processing. The controller could be the customer of a software company and is the collector of data.
3. The data processor is the body that processes personal data on behalf of the controller following the rules and policies determined by the controller. There are many cases where the controller and processor are the same entity. An example of a data processor is a cloud service provider.
4. It should be emphasized that software companies, as far as not collecting and processing individual data is concerned, are not subject to the GDPR, as they do not collect or process the data of individuals. However, their product should still be compatible with GDPR to be applicable in collecting and processing the individuals' data in the EU.

GDPR and Software Engineering

Implementing data protection legislation in data systems is one of a system engineer's tasks that could be categorized under **requirement engineering**. Legal regulations are the primary motivation for implementing data protection requirements into the software design life cycle (Hjerppe et al., 2019). The GDPR's requirements for data protection impose restrictions on data systems that process individual data in the European Union.

Requirement engineering (RE)
The term requirement engineering comprises the identification, analysis, specification, and validation of all the characteristics and basic requirements of a software system that are required or relevant during its life cycle.

From the perspective of software development, there are a number of principles (Px) and rights (Rx) that should be considered during the design and development phases (Kneuper, 2019):

Principles of GDPR-compliant design

P1, Purpose limitation: The collected data could only be processed for the purposes which are originally collected for. For example, the data collected for newsletter subscription, could not be sold to a third party advertisement company (like Facebook) to push advertisements to the newsletter subscriber. If there is a new purpose of data processing, the

software development team must inform the customer of the software development process (the controller or the processor) about this new usage and they should confirm legitimacy (by informing the data subject).

P2, Data minimization: Collecting data "just in case" is strictly forbidden. The software engineers should ensure that for the defined purpose, the minimum possible data should be collected. For example, an application that manages email newsletter subscriptions, must not collect mobile phone numbers, and physical addresses of subscribers.

P3, Storage limitation: Not only the collected data should be minimized, but they should not be stored longer than the required or agreed period. Unlike P1 and P2, this task requires more engagement from the design and development team as they should implement the relevant functionalities in the software to identify the data which are not required anymore or existing longer than the agreed storage limit. For example, a platform that collects the applicants' information to share with the employers, cannot store the applicant data after the period which is mentioned in the terms and conditions of the platform.

P4, Confidentiality: The data collector and data processors should prevent any unauthorized access to the data subject. For example, the credit card information of the users of a payment platform, should not be accessible to all employees of the payment platform. This principle requires the engagement of the data system engineers to implement the relevant functionalities.

Rights of data subjects

In addition to the above principles, GDPR also defines some "rights" which should be also considered during the software design and development process (Regulation 2016/679):

R1, Right to transparent information (Article 12): The controller or the processor should provide transparent and easy-to-understand information about the purpose of data usage to the data subject. This information should be provided in written form. An example of the implementation of this is when a website notifies visitors about cookies.

R2, Right of access by the data subject (Article 15): The data subject should have the right to request their stored data and perform processes on this data. This right requires the presence of the relevant functionalities in a data system in order to retrieve the required information for a specific data subject. This task could be complex, especially in the case of unstructured data such as the photos, comments, and "likes" of a user on a social media platform.

R3, Right to rectification (Article 16): The data subject has the right to modify and update their stored and processed data. The software development team should therefore ensure the relevant features for editing individual data are present. In this case, a modification of unstructured data is challenging, especially when data redundancy must be considered.

R4, Right to be forgotten (Article 17): The data subject has the right to request the removal of their individual data. The most famous case of this right was the European court decision about removing the data subject information from Google search results under certain conditions if a data subject requests it (Mantelero, 2013). The implementation of such a deletion functionality could be challenging for the software development team.

R5, Right to data portability (Article 20): The data subject has the right to migrate their individual data to another processor. That is, a data subject should be able to move their data from Platform A to Platform B.

R6, Right of Data Traceability (Article 30): User requests and data must be traceable to enable the discovery of any potential data exposure to third parties and any potential transfers of personal data outside of the EU. To implement traceability in a data system, accurate logging should be used (Hjerppe et al., 2019).

R7, Right of database physical location: The data subject has the right to know the physical location where their data is stored. To properly account for this right, integration of cloud architecture in the software product may be necessary.

A GDPR-Compliant Software

Considering just the principles and rights above, implementing a GDPR "request interface" and "request service," as shown below, could make a software product GDPR-compliant (Hjerppe et al., 2019). It should be noted that the GDPR has 99 articles and a GDPR-compliant software should be compatible with all articles.

The GDPR "request interface" in the figure below is the component through which users can exercise their rights. At the top of this architecture, we have a public-facing system such as a web server. Through this system, an authenticated user can make a GDPR request. This request will be directed to the "GDPR request interface." This component is connected to a "GDPR request service" which is a part of the private system. The system administrator uses this request service to approve or decline a GDPR request received from the GDPR request interface. The communication between the "administrator user interface" and the "GDPR request interface" is realized by a "request API." The "request-specific database" stores some personal data required for handling requests. Sensitive personal data are stored on the "core personal data module." For example, the "request specific database" stores information about the usernames and email addresses and the "core personal data module" stores the real name and credit card information of users.

Figure 39: GDPR Request Interface

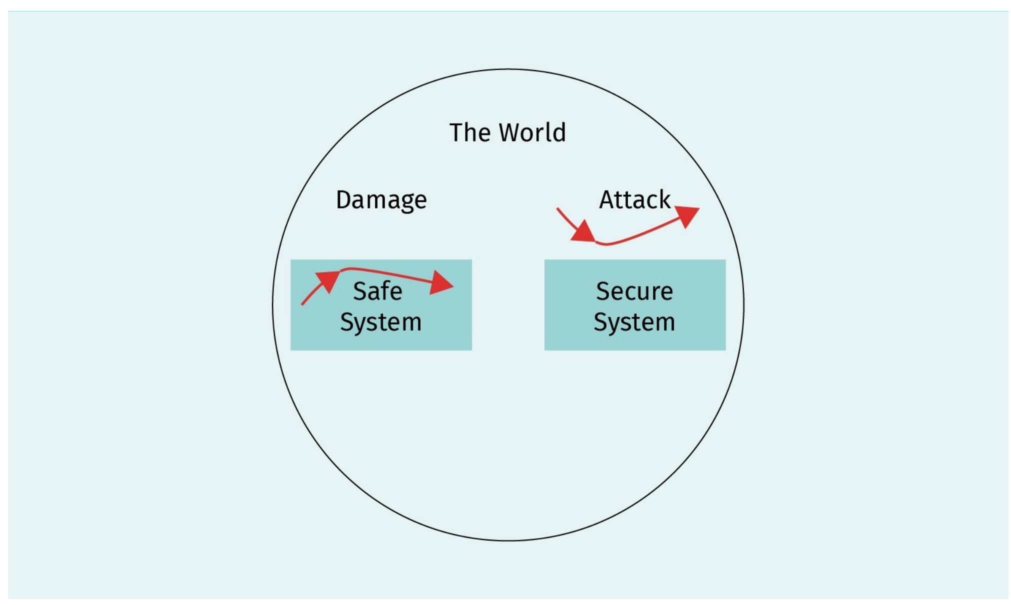
Source: Alavirad, 2020, based on Hjerppe et al., 2019.

4.2 System Security

What is System Security?

We start this section by introducing system security and differentiate it from system safety. These two concepts are sometimes used interchangeably, but there is a notable difference between them. We can distinguish between these two concepts with the following statement (Axelrod, 2013): a safe system should not harm the world and a secure system should not be harmed by the world. Therefore, the goal of a safety-critical system is preventing contamination and the goal of a security-critical system is protection.

Figure 40: Safe vs. Secure System



Source: Alavirad, 2020.

In more technical terms, the goal of security measures for a data system is to permit all intended use of the system and prevent any unintended or unauthorized use of the system, its data, and its information.

Security Requirement Engineering

Requirement engineering is critical to developing any software project. Security requirement engineering is not an exception. An evaluation of a data system is possible if the security requirements are correctly defined. In this subsection, we will discuss two approaches to developing security requirements for the life cycle of a data system (Allen, 2008).

Misuse and abuse cases

When we design a software product, we normally consider the use cases, not the misuse cases. Misuse cases are use cases for attackers. By assessing the misuse cases and thinking from the perspective of a hacker, we can better secure our data system against cyberattacks.

Hackers generally try to attack well-known locations inside the software system, for example, by using the intersystem communication. When a software architect designs a system that always validates all requests from the web-server to the database, hackers use this design vulnerability to make an unintended request to the database server.

Because a system engineer knows their system better than the attackers, they can use this extra knowledge to increase system security and reliability. To achieve this goal, the system engineer should find the answers to questions such as:

- What implicit assumption has been implemented in the system?
- Under which conditions can these assumptions be falsified?
- What possible security attacks could stem from this falsified assumption?

Creating a misuse case table by answering the above questions is very helpful in security requirement engineering. For example, the assumption that the user “won’t” change the cached data because they do not understand it is an assumption that can result in some security risks, as it is not an obstacle for attackers.

In the figure below, you can see a well-known use case and misus case diagram introduced by Alexander (2003). In this diagram, the use cases are shown in orange and the misuse cases in grey. By looking into the system (a car in this example) from the viewpoint of a hacker to the system (car thief), we can develop the security requirements: lock the car, lock the transmission.

Figure 41: Misuse Case vs. Use Case

Source: Alavirad, 2020, based on Alexander, 2003.

Sindre and Opdahl (2001) developed an iterative method, based on common risk and threat analysis, for developing a security requirements diagram using misuse cases. It is as follows:

1. Find the critical assets in the data system.
2. Define the security goals for each data asset.
3. Identify the threats for each security goal. These threats are caused by stakeholders that attack the system or its environment. In this step, a sequence of actions that may result in intentional harm should be developed.
4. Analyze the risks of the threats.
5. Define security requirements for these threats.

The result of the iterative misuse case analysis is a use case diagram that contains use cases, security uses cases, and misuse cases.

In the figure below, we have depicted the misuse case and use cases for an online shop (Chun, 2005). In this UML diagram, the relation between "`use case`" and "`misuse case`" entities could be "`threaten`" or "`mitigate`": the use case can mitigate a misuse case and a misuse case can threaten a use case. For example, here, the use case is "screen input" to reduce the chance of an outside crook spreading malicious code.

Figure 42: Misuse Cases and Use Cases for an Online Store

Source: Alavirad, 2020, based on Chun, 2005.

Security quality requirement engineering (SQUARE)

Security Quality Requirement Engineering (SQUARE) is a process for developing a tool for eliciting, categorizing, and prioritizing the security requirement in an information system (Mead et al., 2005). The main focus of this model is to integrate the security concepts in the very early stages of the software life cycle.

This process can be summarized in the following nine steps (Mead, 2006):

1. Agree on definitions. During this step, different parties involved in the data system project will discuss their concerns regarding system security requirements. It is also possible to use the standards, like IEEE standards, for a clear and standardized definition of security measures and requirements.
2. Identify the security goals. In this stage, the security goals of different stakeholders inside the organization will be identified. For example, for the Human Resources department, the security goal is to keep employees' data safe; for the Finance department, the security of the organization's financial data is the main concern.
3. Develop artifacts. Developing documented normal uses, threat scenarios, and misuse cases will support all subsequent requirement engineering activities.
4. Perform risk assessment. Using a risk assessment method and utilizing the artifacts from step 3 as input, the engineers can identify high-priority security exposures.
5. Select the elicitation technique. This technique is important when there is a diverse range of stakeholders. When there are different stakeholders with different cultural backgrounds, using methods like structured interviews is beneficial to resolve communication issues. When there is only a single stakeholder, an interview with them would be sufficient.
6. Elicit security requirements. This step is the implementation of the selected elicitation technique in step 5.
7. Categorize requirements. In this step, the security requirement engineers will categorize the collected essential security requirement and goals.
8. Prioritize requirements. The categorized requirements in step 7 will be prioritized in this step to find which security requirements have a high pay-off relative to their cost. There are also other factors when prioritizing security requirements, like loss of reputation, loss of customers, and so on.
9. Requirement inspection. In the final step, the requirements engineer inspects the prioritized requirements which are still incomplete and should be revised. Ultimately, the project team will have a clear list of security requirements to implement.

Security Pattern

In the final part of this section, we will review some of the security patterns. A security pattern provides a solution to a security problem by controlling (stopping or mitigating) a set of specific threats. This solution can be explained by using UML classes, states, and activity diagrams.

Each pattern is composed of the following parts (Rosado et al., 2006):

- Intent. This is a description of what the pattern does.
- Context. This the context of the problem.
- Problem. This is a statement of the problem.
- Description. This is a scenario that illustrates a design problem.
- Solution. This illustrates the solution to the problem.
- Use case. This gives some examples of implementations.

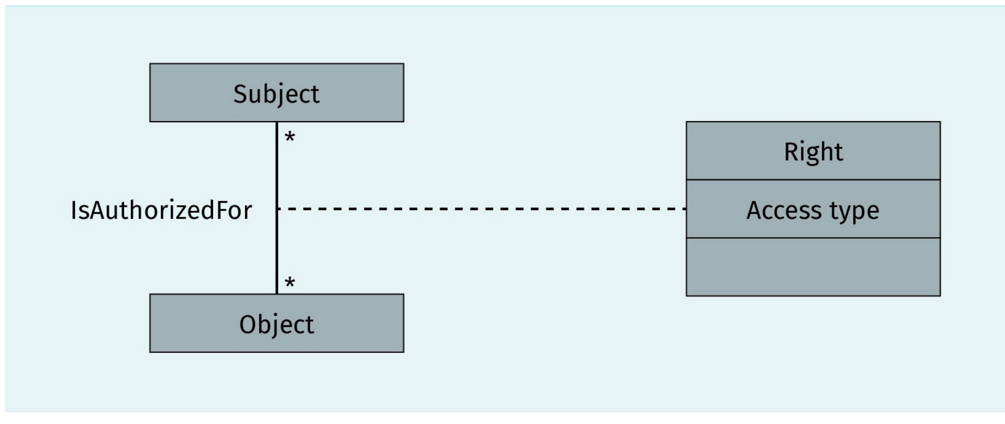
Table 1: Authorization pattern

Intent	Asks the question: who is authorized to access the data system?

Context	Any data system where access to the system should be limited and controlled.
Problem	The permission granted to a “security subject” so it has access to a “protected object” should be explicitly indicated.
Description	To structure the permissions, we distinguish between active entities (security subjects) and passive entities (protection objects).
Solution	The relationship between the “Subject” class and the “Object” class determines which “Subject” is authorized to access the “Object.”
Use case	This pattern is used in the access control systems of most commercial products such Unix, Windows, and Oracle.

Source: Alavirad, 2020.

Figure 43: Authorization Pattern



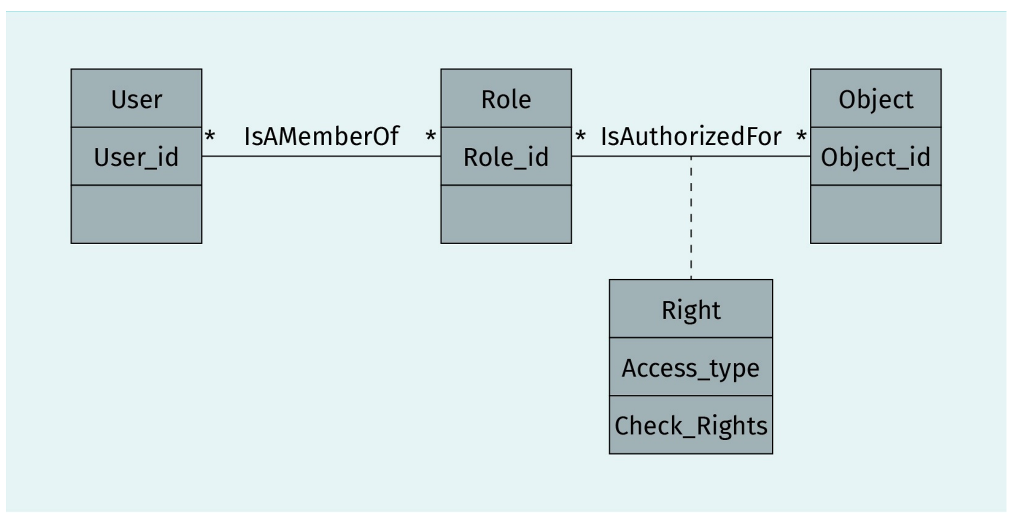
Source: Alavirad, 2020.

Table 2: Role Bases Access Control Pattern (RBAC)

Intent	Provides authorized access to the data system based on role
Context	Any data system where the access to the system should be limited and controlled according to the “Subject” role
Problem	The permission granted to a security subject in order to grant access to a protected object should be explicitly indicated according to its roles.
Description	This pattern improves the administration process by giving different access levels to different “subjects” or a group of “subjects.”
Solution	This is an extension of the authorization pattern by defining roles as subjects. “Users” are assigned to “Roles” and “Roles” are given “Rights” according to their functions (roles). The “Right” association class defines the type of access.
Use case	IBM’s WebSphere, Oracle, Microsoft Defender, and Azure

Source: Alavirad, 2020.

Figure 44: Role Bases Access Control Pattern (RBAC)



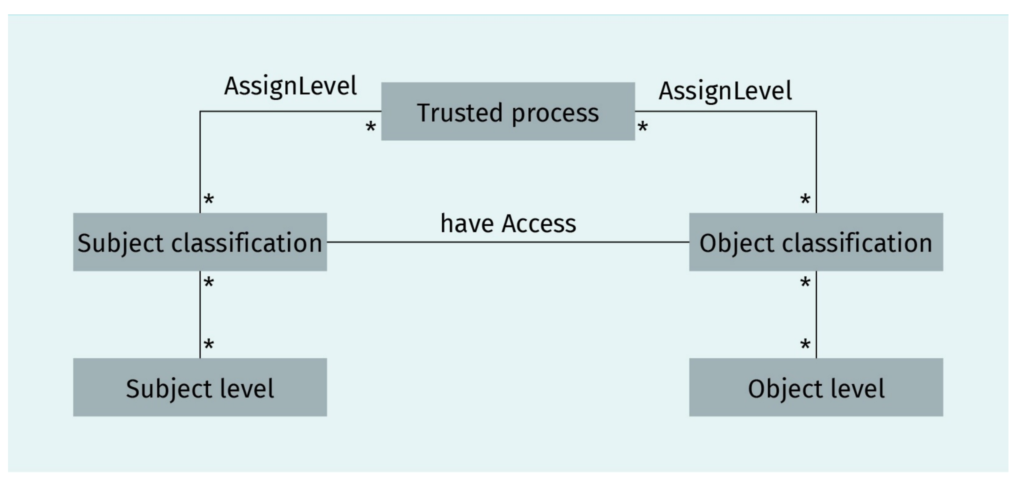
Source: Alavirad, 2020.

Table 3: Multilevel Security Pattern

Intent	Access management in a security system with several levels of security classification
Context	Appropriate for any system with several security levels
Problem	How can security in a system with several security classifications be managed?
Description	This provides different security levels for both “Subject” and “Object.”
Solution	To provide a multilevel security structure, we define the “Subject classification” and “Object classification.” The instances of these classes are used to define the level and object security categories.
Use case	IBM’s Informix

Source: Alavirad, 2020.

Figure 45: Multilevel Security Pattern



Source: Alavirad, 2020.

Table 4: Single Access Point Pattern (SAP)

Intent	This pattern provides an interface for communication from external entities to the system.
Context	Communication with external entities
Problem	When a system has multiple access points to the external entities, managing these access points could be a security challenge.
Descrip-tion	It prevents any direct access from external entities to the internal modules of the system. Each access request is directed through a supervised channel.
Solution	SAP is the single connection point to the external world and all incoming requests are direc-ted to the SAP instance. To apply certain policies, an instance of the "Check Point" class is applied.
Use case	Windows NT login applications

Source: Alavirad, 2020.

Figure 46: Single Access Point Pattern

Source: Alavirad, 2020.

4.3 Data Governance

We start this section with a story about a failed $125 million project at NASA (Douglas, 1999). In 1999, a failure in the transferring of information between the Mars Climate Orbiter spacecraft team in Colorado and the mission navigation team in California led to the loss of the Mars Polar Lander spacecraft in 1999. The reason? One team did the calculation using the **imperial system** instead of the metric system. The lack of a central policy on how to uses data within an organization (here, NASA) led to the project failing.

Imperial system
This is a system of measurement in use in the United Kingdom and other Commonwealth countries, consisting of units such as the inch, the mile, and the pound.

Data governance defines a set of principles and practices to support efficient performance in evaluating, generating, storing, processing, and deleting corporate data.

IBM (2020a) defines data governance as: “Data governance is the overall management of data availability, relevancy, usability, integrity and security in an enterprise” (para. 1).

The Data Governance Institute (2008) explains data governance as: “Data Governance is a system of decision rights and accountabilities for information-related processes, executed according to agreed-upon models which describe who can take what actions with what information, and when, under what circumstances, using what methods” (p. 3).

Data governance includes the methods, roles, measures, and metrics that guarantee the efficient and productive usage of data and information. It should not be confused with data management, change management, data cleansing, data warehousing, or database design. The goals of data governance could be listed as (Sweden, 2009):

- ensure transparency
- enable better decision making
- reduce operational friction
- meet stakeholder needs
- establish common approaches
- establish standard repeatable processes
- reduce costs
- increase effectiveness

The deliverables of a successfully executed data governance program are as follows:

- Data policies. A set of statements that controls data integration, storage, security, and usage. In simple terms, they tell us what to do and what not to do with our data.
- Data standards. They provide the methods to implement data policies concretely, like naming and modeling standards for data.
- Data management projects. Data governance regulates data management projects in the organization and increases the success rate of such projects.
- Data quality. The core product of data governance is higher quality data and information, i.e., uncomplicated access to data and manageable and auditable security measures of data.
- Data value. In an enterprise with well-implemented data governance, the value of data assets is more transparent and extractable.

Areas of Focus

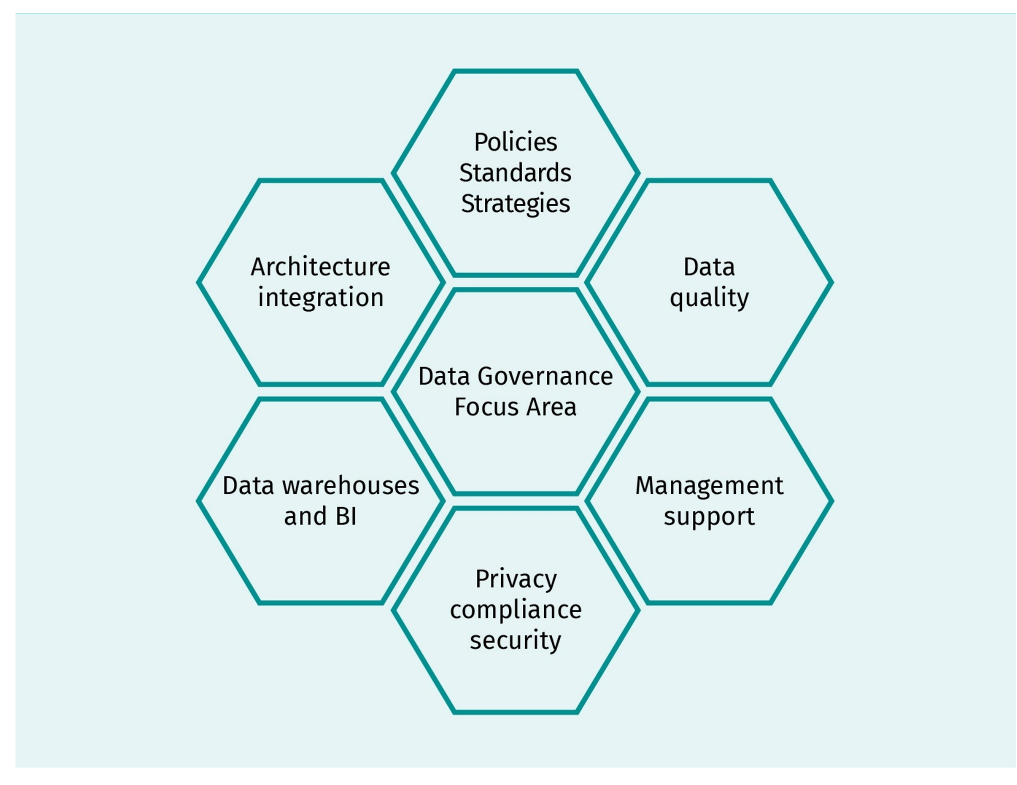
The objective of a data governance program falls within one of the following focus areas. In general, a data governance program is not limited to only one focus area and they cover normally most of the areas (Data Governance Institute, 2009).

Data governance programs with a focus on policies, standards, and strategy

These programs are required when, for example, an organization’s team needs help (orders) from cross-functional leadership. For instance, let's take a company that has decided to migrate from silo development to enterprise systems. The application development teams at this company may oppose the decision of data engineers. In this case, data governance policies, supported by cross-functional leadership, could resolve the situation regarding the architectural positions.

These types of data governance programs choose, review, approve, and monitor data policies and standards. They also contribute to business rules and data strategies.

Figure 47: DGI Definition of Data Governance Focus Areas



Source: Alavirad, 2020, based on Data Governance Institute (2009).

Data governance programs with a focus on data quality

These programs are important when there are issues with data quality, integrity, and usability. The quality efforts normally start with master data. These programs define directions for data quality and control data quality.

Data governance with a focus on privacy/compliance/security

These types of programs are typically important when there are issues around data privacy, access management, or compliance with regulatory, contractual, or internal requirements. These programs protect sensitive data, assess risk, and define controls to manage risks.

Data governance programs with a focus on data architecture

These types of programs are relevant when a major change, modification, or development in the enterprise data system architecture is needed and requires cross-functional decision-making. For example, migrating an enterprise data system from on the premises to a cloud architecture.

This program ensures data consistency in data models and definitions, provides architectural policies and standards, and supports metadata programs and enterprise data management.

Data governance programs with a focus on data warehouses and business intelligence (BI)

These types of programs concern a specific data warehouse or business intelligence tool. They define rules and standards before the new system becomes operational.

Data governance programs with a focus on management support

These types of programs are important when managers struggle with making routine data-related decisions. These programs help managers to make decisions with more confidence.

These programs measure the data value, monitor data, report data-related projects, and promote data-related messages.

Data Governance Framework

The data governance framework is the roadmap for implementing and maintaining data governance within an organization. It is a collection of strategies, legislations, policies, and relevant tools.

The Data Governance Institute (2008) describes the purpose of a (data governance) framework: “Frameworks help us organize how we think and communicate about the complicated or ambiguous concept” (p. 5). Among the advantages of a data governance framework, we can name:

- clarity
- ensuring value from the efforts
- creating a clear mission and vision
- maintaining scope and focus
- establishing accountabilities
- defining measurable successes

We will examine the data governance framework proposed by the Data Governance Institute (2009). Here, the main components of the framework and the sequence used to realize a data governance program are described. In this framework, we encounter the Why, What, How, When, and Who domains.

1. The “mission” explains why a data governance program is required in the organization. The mission should be in line with the enterprise objectives and is set by the management team of the enterprise.
2. The “focus area” states what the short-term and long-term goals of executing a data governance program are.

3. “Data rules and definitions” explain how enterprise rules could be transformed into data policies, data standards, and data definitions. To realize this task, the “data governance offices”, data stewards, and data owners should work closely together.
   a) The "decision rights" manage daily enterprise data assets. They define, for example, which topics data stewards have the right to make decisions about and which decisions should be made by the data governance committee.
   b) "Accountabilities" determine who is responsible for what within the organization.
   c) "Control mechanisms" monitor the realization of data rules and accomplishments of goals.
4. Who is engaged in a data governance program?
   a) Data stakeholders: they are directly responsible for the data within the organization.
   b) Data governance offices: these set policies and procedures for data governance.
   c) Data stewards: these enforce data governance policies within the organization.
5. The “data governance process” defines how the standardized and documented data governance program is deployed and when each milestone should be performed and repeated (the life cycle of the data governance program). It includes seven steps:
   a) Develop a value statement. Before starting the program and deciding on roles, the program’s value statement should be defined.
   b) Prepare a roadmap. The roadmap should be developed and shared with the stakeholders.
   c) Plan and fund
   d) Design the program
   e) Deploy the program
   f) Govern the data
   g) Monitor, measure, and report

SUMMARY

In this unit, we learned about data protection. We discussed a specific example of a data protection act: the General Data Protection Regulation (GDPR), which protects the data of the European Union citizens against the data collectors and data processors. We also discussed the effect of the GDPR implementation on a software product from the data system engineer’s prospect.

We then discussed software security and defined security requirement engineering (SRE). We briefly introduced two methods of SRE: misuse and abuse cases and security quality requirement engineering (SQUARE). We discussed some basic security plans, such as authorization pattern and single access point pattern.

Finally, we introduced data governance and its importance for organizations. We have learned that, to have a successful data governance program, a data governance framework is required.

UNIT 5

COMMON CLOUD PLATFORMS & SERVICES

STUDY GOALS

On completion of this unit, you will have learned …

- what cloud computing is.
- what Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) are.
- about the Amazon Web Services (AWS) cloud computing platform and the services it provides.
- about the Google Cloud Platform (GCP) cloud computing platform and the services it provides.
- about the Microsoft Azure cloud computing platform and the services it provides.

5. COMMON CLOUD PLATFORMS & SERVICES

Introduction

In this unit, we will briefly discuss cloud computing principles and introduce some of the well-established cloud computing platforms. We will first discuss cloud computing and three delivery models to provide cloud computing services: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). We will also discuss cloud service characteristics like agility, elasticity, and billing and metering.

We will then introduce the Amazon Web Services (AWS) platform and some of its services, like the Amazon Elastic Compute Cloud (EC2) service, Amazon Aurora, Amazon DynamoDB, AWS Security Hub, and Amazon CloudWatch.

Google Cloud Platform will be discussed along with some of its services, like Cloud Functions, Google Kubernetes Engine, Cloud Storage, Virtual Private Cloud, BigQuery, and Dataflow.

Finally, we will discuss Microsoft's cloud computing platform: Microsoft Azure. Through its many services, we will discuss Azure Kubernetes Service (AKS), Azure Functions, Azure SQL Database, Azure Storage, Azure Databricks, and many more.

5.1 Cloud Computing

The National Institute of Standards and Technology (NIST) defines cloud computing as :

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" (Mell & Grance, 2011, p. 2).

Microsoft defines cloud computing as "the delivery of computing services—including servers, storage, databases, networking, software, analytics, and intelligence—over the Internet ("the cloud") to offer faster innovation, flexible resources, and economies of scale. You typically pay only for cloud services you use, helping you lower your operating costs, run your infrastructure more efficiently, and scale as your business needs change" (2020q, para. 1).

Amazon also defines cloud computing as "the on-demand delivery of compute power, database storage, applications, and other IT resources through a cloud services platform via the Internet with pay-as-you-go pricing" (2020p, para. 1).

The companies that provide cloud services are called cloud providers. Examples of these are Amazon, Google, Microsoft, IBM, and SAP. Cloud providers offer a wide range of products and services which can be summarized in the following categories:

- Computer services, which are used as infrastructure to run applications in the cloud. Virtual machines (VMs), functions, container instances, and virtual services are among these.
- Database services, including fully-managed databases to be used with cloud-based applications. Relational, NoSQL, in-memory, and time-series databases are among these types of services.
- Storage services, which provide storage services for storing data such as objects and files. Among products of this category are file storage, archive storage, disk storage, and cache services.
- Networking services, which facilitate a secure connection between the cloud servers and the enterprise data warehouses. API gateways, load balancer, and VPN gateways services belong to this category of services.
- Security services, which are crucial to any cloud computing platform. These services protect data, applications, and infrastructure against threats by identifying them using advanced methods like machine learning. Application gateways, certificate managers, firewall managers, and threat detectors belong to this category of services.

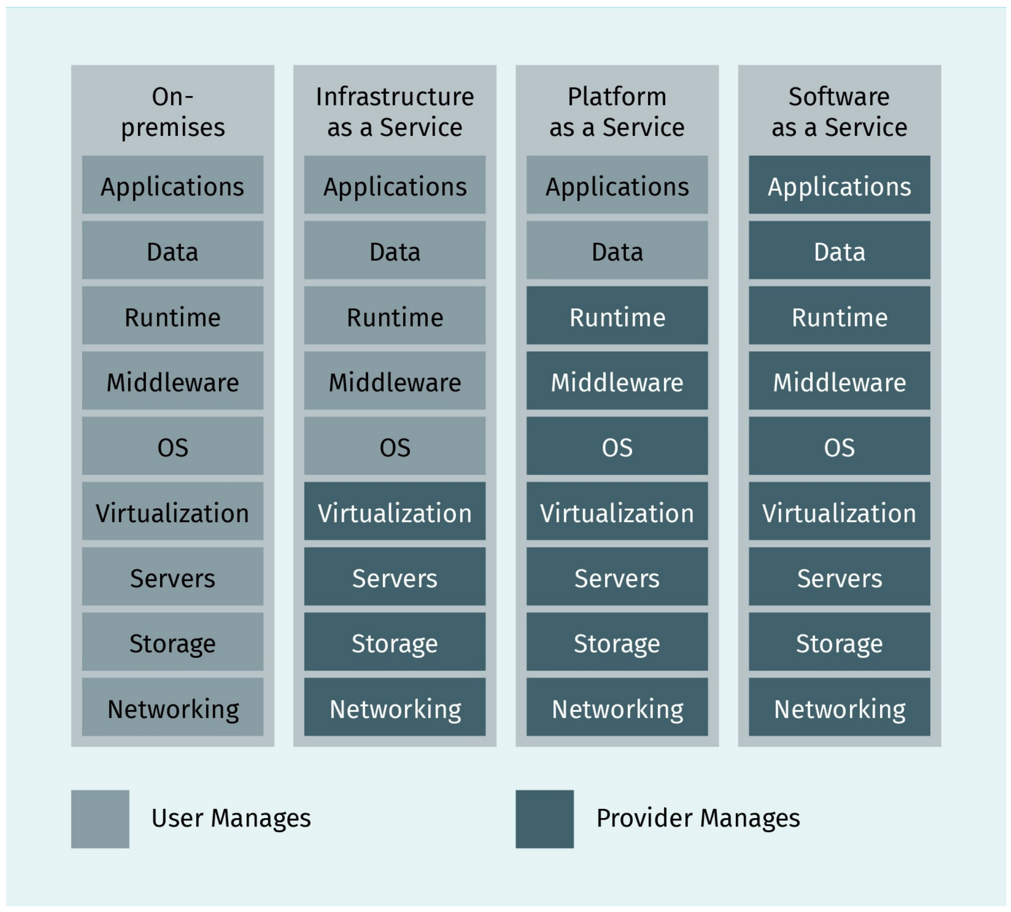
Cloud Delivery Models

There are three main cloud delivery models:

1. Infrastructure as a Service (IaaS). In this model, the cloud provider hosts all required infrastructure for the clients such as hardware, software, storage, networking, and so on. In this delivery model, the client has control over all the provided infrastructure. For example, the user can create virtual machines on an IaaS platform, install operating systems on those virtual machines, deploy middleware such as databases, develop and deploy an application, and run that application. Although the user has more control over the system with this model, they also have more responsibility concering the system maintenance. IaaS is normally used for migrating workloads, testing and development, and storage, recovery, and backup scenarios.
2. Platform as a Service (PaaS). In this model, the cloud providers offer a set of development and middleware services to the users to test and deploy their application on the cloud. In PaaS, the users do not have as much control over the infrastructure as the IaaS. However, the user has fewer maintenance responsibilities. For example, the cannot create virtual machines and install operating systems on them. These services are normally used for development framework and analytics and business scenarios.
3. Software as a Service (SaaS). The third model we will examine is SaaS, where the user has very limited control over the infrastructure and the deployment platform. In this model, the cloud providers host the infrastructure as well as the required applications. The users cannot install anything on the SaaS platform; they simply log in and use the deployed software.

In the figure below, you can see the difference between these three cloud delivery models and the on-premises model (in-house hosting). As you can see in the on-premises model, the user has full control over the infrastructure, while on the SaaS, the user has zero control over the infrastructure and applications.

Figure 48: IaaS vs. PaaS vs. SaaS



Source: Williamson, 2020.

Cloud Service Deployment Models

Taking NIST's definition into account, we can consider the following cloud service deployment methods (Mell & Grance, 2011):

- Public cloud: in this model, all services are fully deployed in the cloud (i.e., infrastructure, including software and hardware). The cloud infrastructure is provided for open use by the general public and it may be owned, managed, and operated by a business, academic, or government organization (or some combination of these). The infrastructure is installed on the premises of the cloud provider.

- Community cloud. The cloud infrastructure is provided for a particular use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, or compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of these. The infrastructure is installed on or off premises.
- Private cloud. This model is very similar to legacy IT infrastructure. The cloud infrastructure is provided for exclusive use by a single organization, including various consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of these. The infrastructure is installed on or off premises. This model is used when, for example, due to low latency requirements, workloads need to remain on-premises, local data processing is required, or the organization has legacy architecture.
- Hybrid. In this deployment model, the existing on-premises IT infrastructure of an enterprise will be connected to the on-demand (cloud) infrastructure of a cloud service provider in order to scale the organization IT infrastructure. In this model, the cloud infrastructure is a combination of two or more separate cloud infrastructures (private, community, or public) that are unique entities, but are joined in such a way that enables data and application portability (e.g., cloud bursting for load balancing between clouds).

Cloud Service Characteristics

Elasticity

This is the ability to increase or decrease resources dynamically or automatically (Clinton, 2019). This characteristic is important because, in most cloud computing platforms, users are charged for resources (like computing power and storage) on a per-use basis. Resources are used when they are needed for computational purposes. When they are not required, they will be de-provisioned so the user does not pay for idle resources. Scalability is similar to elasticity but is not necessarily automatic (Clinton, 2019).

Workload management

As the cloud service is a distributed service around the globe, it is very important to orchestrate these distributed services to manage workloads.

Agility

This is the ability to quickly allocate and de-allocate resources.

Availability

The cloud service should provide the required hardware and software to keep the services up with very little down-time.

Disaster recovery

The cloud service should be able to recover a down service quickly and automatically.

Fault tolerance

The cloud service should be able to keep services up even when some components are not functional.

Billing and metering

The cloud service should provide tools to measure and predict the resources and the relevant costs.

Management services

A cloud service should provide tools to users to manage cloud computing. For example, security services and data governance are very critical for any cloud computing platform.

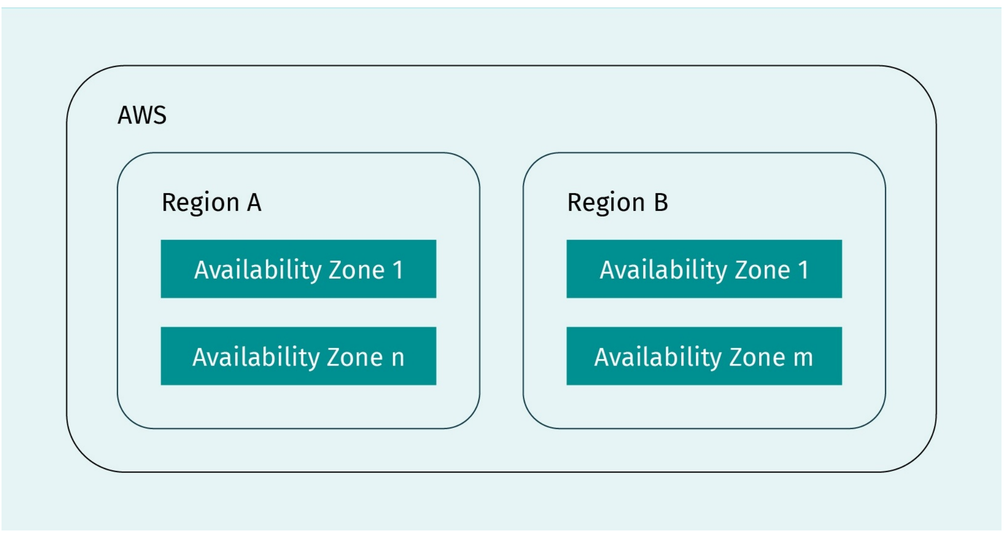
5.2 Amazon Web Services

In 2006, Amazon Web Services (AWS) launched a service offering IT infrastructure services to businesses. This service was called Web Services and still is the name of the Amazon's cloud computing platform.

The infrastructure of the AWS Cloud is distributed around the globe in different AWS Regions and Availability Zones. Each AWS Region, which is a physical location somewhere in the world, is composed of multiple AWS Availability Zones (shown below). Each Availability Zone is made up of one or more separate data centers, where each data center has redundant power, networking, and other resources, all housed in separate facilities. This redundancy within each Availability Zone provides higher availability, scalability, and fault tolerance. As of today, AWS has 24 Regions and 77 Availability Zones.

Each AWS Region is completely isolated from other AWS zones. However, Availability Zones inside an AWS Region are connected by low latency connections. Users can store their data within multiple AWS Regions and across several AWS Availability Zones within each AWS Region.

Figure 49: AWS Regions and Availability Zones



Source: Alavirad, 2020.

AWS Services

Amazon Web Services offers a wide range of services including compute, storage, databases, analytics, and networking services (Amazon, 2020o). As of today, AWS provides 140 services. In this section, we will discuss some of the main AWS services.

Compute services

AWS provides different compute services as computing power for running applications in the cloud. Amazon's Elastic Compute Cloud (EC2) service is the most well-known of these services and facilitates web-scale computing for developers by providing secure, scalable compute capacity in the cloud. Put simply, EC2 enables users to rent virtual machines from AWS to run their applications.

There are three types of EC2 instances (Amazon, 2020m):

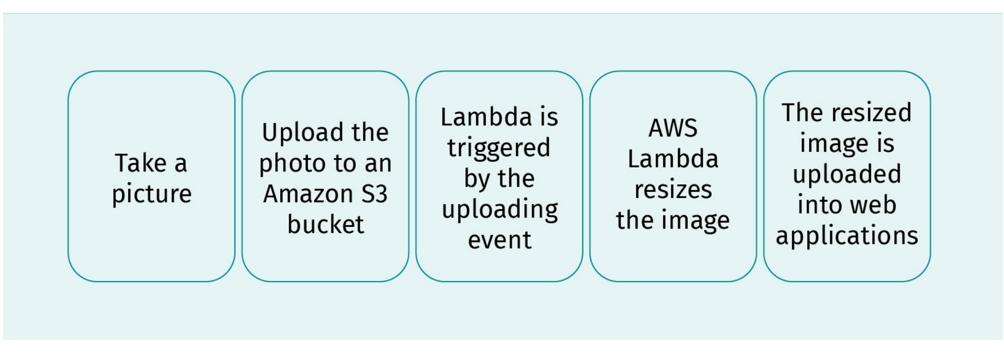
- On-demand. In this model, the user pays for the compute capacity on an hourly basis. The user chooses the required computing capacity for the application and pays only for the hours that the application has consumed the compute capacity.
- Reserved. In this payment model, the user reserves a specific amount of compute capacity and pays for it. This model can be up to 75 percent cheaper than the on-demand instance type, but the user must pay for idle resources. Therefore, the user should be sure about the amount of resources that will be used in advance.
- Spot. In this model, the user takes advantage of unused EC2 capacity in the cloud, reducing the cost by up to 92 percent compared to the on-demand method. Here, the user bids on EC2 computing capacity and the spot instances are available as long as the client is not outbid by a another bidder. These instances can therefore be shut down within seconds and they should not be used for critical applications.

To use Amazon EC2, an Amazon Machine Image (AMI) is initially created that includes an operating system, applications, and configuration settings. The AMI is then uploaded to the Amazon Simple Storage Service (Amazon S3) and registered to Amazon EC2. This generates an AMI ID. Once this is done, the user can start multiple virtual machines using predefined settings. The number of virtual machines can be increased or decreased in real time. To add or remove EC2 instances, AWS provides a scaling service called EC2 Auto Scaling.

Serverless computing
Services that provide and maintain the required hardware and software infrastructiure for running users' applications are called serverless computing services.

Another AWS compute service is Amazon Lambda, which is designed for the **serverless** running of applications (Amazon, 2020h). The user writes AWS Lambda functions, i.e., self-contained applications that are written in one of the programming languages and runtimes supported by AWS Lambda. The functions are then uploaded to AWS Lambda and executed by AWS Lambda efficiently and flexibly by managing the required resources. For example, a real-time file processing application code can be deployed to AWS Lambda. In this case, the user utilizes the AWS S3 to trigger AWS Lambda processing the data after it is uploaded in AWS S3. This service also can be used for real-time stream processing, such as IoT backends.

Figure 50: Using AWS Lambda for Real-Time Image Processing



Source: Alavirad, 2020.

Database services

Amazon Aurora is a SQL and PostgreSQL-compatible relational database engine managed entirely by Amazon Relational Databases Services (Amazon, 2020l; 2020i).

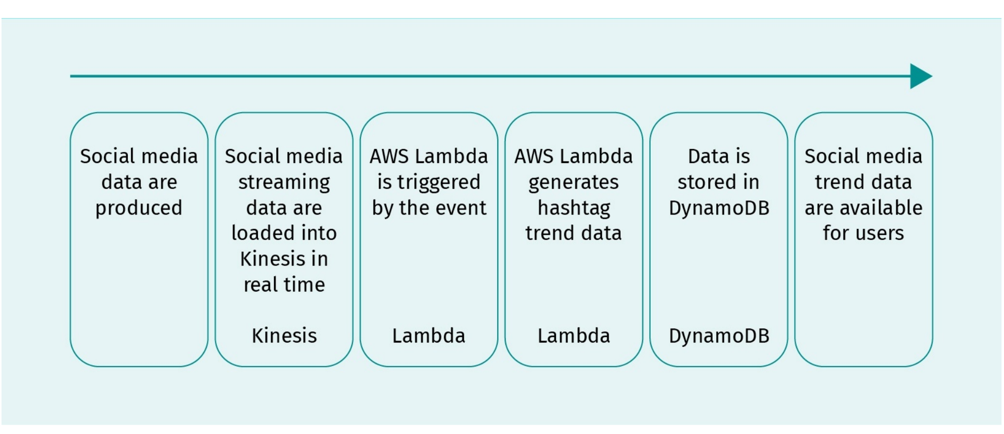
Amazon Relational Database Services (RDS) automates tedious management tasks such as hardware provisioning, database setup, patching, and backups (Amazon, 2020i).

Amazon DynamoDB is a key-value and document database that can be used for serverless web, mobile backend, and microservice applications (Amazon, 2020b). DynamoDB can handle more than ten billion requests per day and supports peaks of more than 20 million requests per second.

AWS Lambda, Amazon Kinesis (a real-time streaming data analysis service), and Amazon DynamoDB can cooperate to process streaming data in real-time in order to track application activity, process order transactions, perform clickstream analysis, cleanse data, gen-

erate metrics, filter logs, perform indexing, analyze social media activity, and perform telemetry and measurement of IoT device data (Amazon, 2020b; Amazon, 2020g). Such an architecture is shown below.

Figure 51: Social Media Streaming Data Analysis Using Amazon Kinesis, AWS Lambda, and DynamoDB



Source: Alavirad, 2020.

Storage services

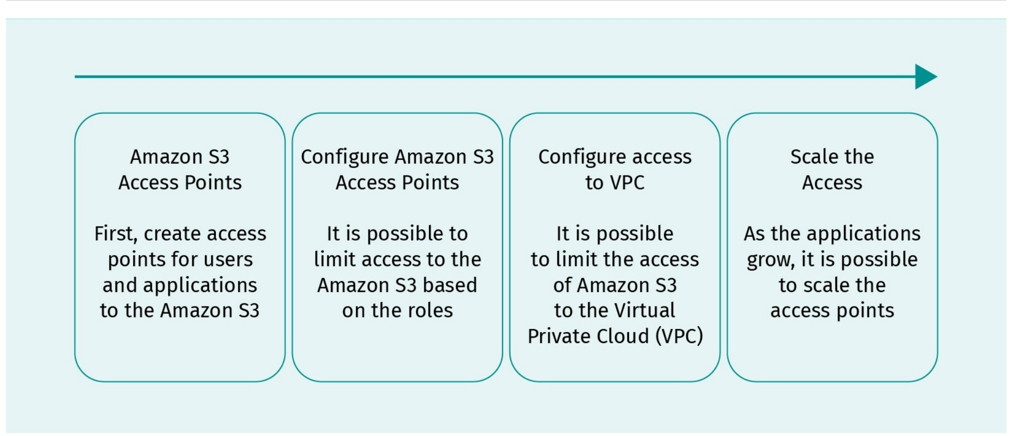
Amazon Simple Storage Service is an object storage service that supports the storage of many use cases from websites and mobile applications to IoT devices and big data analytics. This storage system is designed for 99.999999999 percent of data durability, as it automatically creates duplicates of data across multiple systems (Amazon, 2020j). The Amazon S3 Access Point service facilitates data access for applications using shared datasets on S3 (Amazon, 2020j).

Amazon S3 offers a variety of storage classes that can be used for different use cases:

- S3 Standard class for general storage of data with frequent access
- S3 Intelligent tiering class for data storage with unknown or changing access patterns
- S3 Standard-Infrequent Access class (S3 Standard IA) and S3 One Zone-Infrequent Access class (S3 One Zone-IA) for long-term data with less frequent access
- Amazon S3 Glacier class (S3 Glacier) and Amazon S3 Glacier Deep Archive class (S3 Glacier Deep Archive) for long-term archiving and preservation of digital data

Amazon S3 Standard, S3 Standard-IA, S3 One Zone—IA, and S3 Glacier are all designed for a 99.999999999 percent annual durability. This durability level is equivalent to an annual expected object loss of 0.000000001 percent. For example, if you store 10,000,000 objects with Amazon S3, you can expect, on average, to lose a single object every 10,000 years. In addition, Amazon S3 Standard, S3 Standard-IA, and S3 Glacier are designed to preserve data even if an entire S3 Availability Zone fails.

Figure 52: Amazon S3



Source: Alavirad, 2020.

Amazon Elastic File System (EFS) provides an elastic, scalable NFS file system for Linux-based workloads in AWS Cloud services and for on-premises infrastructures. This service can be scaled in size up to petabytes. Users can choose between two different storage classes: standard storage class and the Infrequent Access Storage class (EFS IA), which is appropriate for files that need not be frequently accessed. AWS can move the infrequent access files to the EFS IA to reduce the storage costs, as EFS IA is cheaper than the standard EFS. This functionality is activated by enabling EFS Lifecycle Management service on a file system (Amazon, 2020c).

The use cases of EFS range from home directories to big data analytics, content management, media and entertainment workflow, database backups, and container storage.

The workflow of storing data using Amazon Elastic File System is as follows:

1. Create a file system in the Amazon EFS console and choose the performance and throughput mode
2. Mount the EFS file system on an EC2 instance with NFSv4
3. Test and optimize the workloads' performance
4. Copy the data into EFS file system

Security services

AWS Security Hub is a single platform that provides a comprehensive overview of the security status and alerts from all AWS security services like Amazon GuardDuty, Amazon Inspector, and AWS's partner products (Amazon, 2020k). AWS Security Hub supports industry standards like Center for Internet Security (CIS), AWS Foundations Benchmark, and Payment Card Industry Data Security Standard (PCI DSS).

Amazon GuardDuty is a threat detection service based on machine learning, anomaly detection, and integrated threat intelligence. It continuously monitors for malicious activity and unauthorized behavior in order to protect the AWS accounts, workloads, and data stored in Amazon S3 (Amazon, 2020e).

Amazon Inspector is a service used to check the security compliance of applications deployed on AWS. This service helps the user to check for unintended network access of the EC2 instances (Amazon, 2020f). After performing an assessment, Amazon Inspector produces a detailed list of security findings, ranked by severity.

Management and governance services

Amazon CloudWatch is a monitoring service for DevOps engineers, developers, site reliability engineers (SREs), and IT managers. CloudWatch collects monitoring and operational data in the form of logs, metrics, and events, providing you with a unified view of AWS resources, applications, and services that run on AWS and on-premises servers (Amazon, 2020a).

CloudWatch captures monitoring and operational data in the form of logs, metrics, and events and visualizes them through automated dashboards. This gives a complete overview of AWS resources and applications, as well as services running in AWS and on servers. Users can correlate the metrics and logs to get a better overview of the health and performance of the resources. Users can also create alerts based on custom metric thresholds or monitor the resources for abnormal behavior using machine learning algorithms. To respond quickly and reduce the meantime to resolution, the user can configure automated actions to notify you when an alarm is triggered. It also provides a comprehensive insight by analyzing metrics, logs, and traces.

Analytics services

Amazon Elastic MapReduce (Amazon EMR) is a big data cloud platform that uses open source tools such as Apache Spark, Apache Hive, and Apache HBase, combined with several AWS products in order to perform tasks such as web indexing, data mining, log file analysis, machine learning, scientific simulation, and data warehousing (Amazon, 2020d). Below, you can see a simple architecture of an Amazon EMR job flow using Amazon EC2, Amazon Simple Storage System (S3), and Amazon CloudWatch.

Figure 53: Amazon Elastic MapReduce (Amazon EMR)

Source: Alavirad, 2020.

5.3 Google Cloud

Google Cloud Platform (GCP) began in 2008 when Google announced the release of AppEngine, a development and hosting platform for web applications in Google data centers. Since then, Google has been constantly adding more services to its cloud platform.

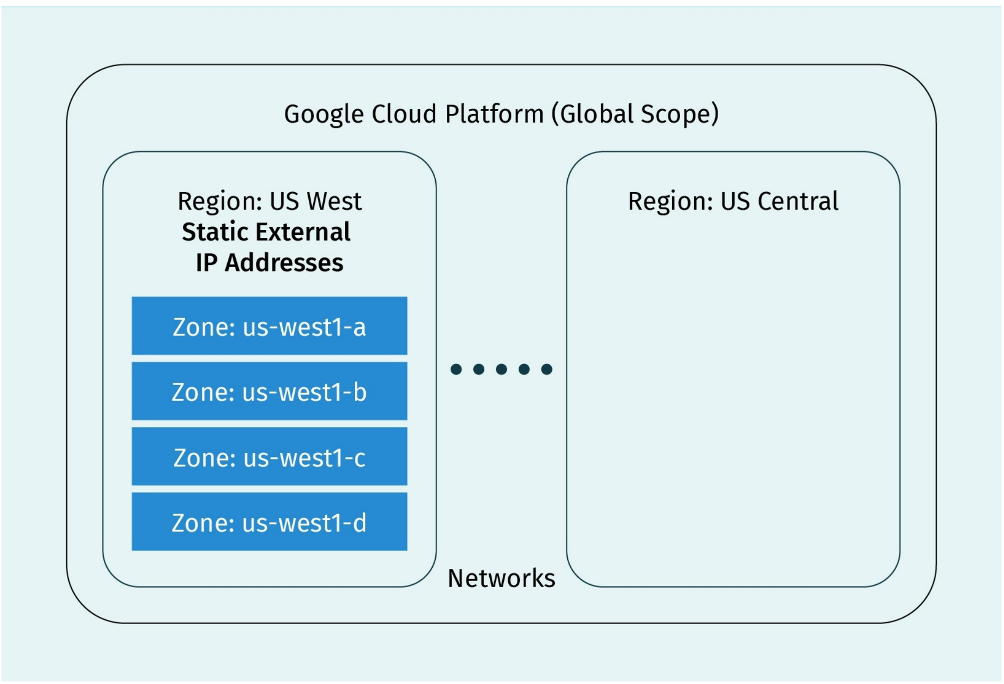
Google defines its cloud platform as a set of physical resources such as memory, computers, and virtual resources such as virtual machines, which are contained in Google data centers around the world (Google, 2020e).

Like AWS, Google also divides the geographical distribution of its data centers into regions and zones. Regions are specific geographical regions where users can host their resources, like East Asia, West Europe, and so on. As of today, Google Cloud has 24 regions. Each region has one or more zones and most regions have three or more zones (called a, b, c, d). Each zone is isolated from the other zones and are identified by a combination of the region and zone names. For example, the `us-west1` region is the west coast of the United States, and has three zones: `us-west1-a`, `us-west1-b`, and `us-west1-c`.

Regarding regions and zones, access to resources is divided into three categories:

- Zonal resources are the resources which are accessible by other resources that are hosted in the same zone, for example, virtual machine instances or zonal persistent disks are examples of zonal resources.
- Regional resources are the resources which are accessible by all resources that are hosted in the same region, regardless of zone. Static external IP addresses are an example of regional resources.
- Global resources are the resources which can be accessed by any resource regardless of region and zone. Images, snapshots, instance templates, and firewalls are among the regional resources.

Figure 54: Global, Regional, and Zonal Resources in Google Cloud Platforms



Source: Alavirad, 2020.

All allocated Google Cloud resources must belong to a specific project. A project forms the basis for creating, enabling, and using all Google Cloud services, such as managing APIs, managing permissions for resources, and billing. A project has the following parts:

- Project name. This is a readable name that can be edited and changed during the project life cycle, as they are not used by any Google API.
- Project ID. This is a unique and user-defined ID and is used by Google APIs. The project ID cannot be modified once it has been created or assigned, and it cannot be used again, even if the project is deleted.
- Project number. This is a unique, automatically generated identifier for the project.

Google Cloud Platform Services

In this section, we review some of the commonly used Google Cloud Platform services. Google Cloud has roughly 90 different services.

Computing and hosting service

Google Cloud provides several services for computing and hosting. In the following, we will discuss some of these services, ordered from least maintenance responsibility to highest maintenance responsibility:

- serverless computing
- managed application platform

- container technologies
- building the cloud-based infrastructure

Serverless computing in Google Cloud is possible by using the Cloud Functions, known as functions as a service (FaaS). Cloud Functions provide an environment for building and connecting cloud services in a serverless mode (Google, 2020i). In this approach, the user writes a simple and single-purpose function (e.g., in JavaScript, Python 3, Go, or Java) that is triggered by events from other cloud services. A cloud event could be a change in a database, a new file in a storage system, or a new virtual machine instance.

In the serverless model, Google Cloud takes responsibility for managing servers, configuring software, updating frameworks, and patching operating systems. Use cases of serverless computing are asynchronous workloads like lightweight ETL (extract, transfer, load), **WebHooks**, and IoT.

WebHook
A WebHook is an HTTP callback: an HTTP POST that happens when an event happens. It is a simple event notification via HTTP POST.

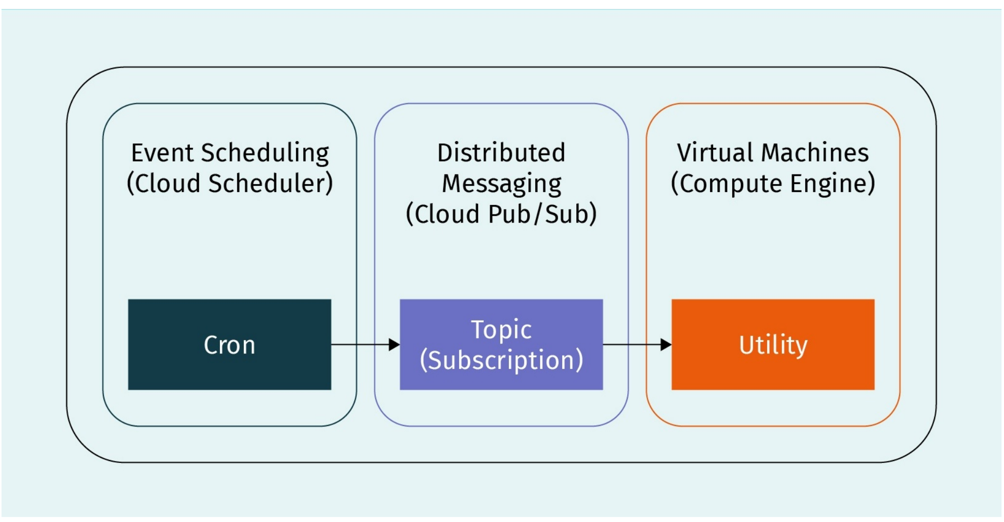
Another Google Cloud compute service is the managed application platform (Google, 2020a). App Engine is Google Cloud’s platform as a service (PaaS), which handles most applications' resource management. The user can write an application here in Go, Java, .NET, Node.js, PHP, Python, or Ruby. Applications on App Engine can be connected with Google databases like Cloud SQL, Firestore (in Datastore mode), and Cloud Storage. It is also possible to connect App Engine applications to third-party data storage systems like Redis, MongoDB, and Cassandra databases.

In container-based computing services, developers can focus on application development instead of deployment and integration into the hosting environment (Google, 2020k). Google Cloud provides a container-based computing service through Google Kubernetes Engine (GKE). Google Cloud's containers as a service (CaaS) is built on the open-source Kubernetes system.

Google Cloud also offers an unmanaged compute service: Compute Engine, which is their infrastructure as a service (IaaS). The Compute Engine user must configure, manage, and monitor the system. With this option, the user can use the virtual machines (instances) to execute applications (Google, 2020f). The user can also choose the region and zone for application deployment, as well as the operating system, development stacks, languages, and frameworks.

In most cases, it is beneficial to combine different computing service types. One example of hybrid computing service types is reliable task scheduling using Compute Engine and Cloud Scheduler. In this solution, we use Cloud Scheduler (Google, 2020m) for scheduling and Pub/Sub (Google, 2020l) for distributed messaging. As a result, the user can reliably schedule tasks across a fleet of Compute Engine instances. In this architecture, events are scheduled in Cloud Scheduler and are then transmitted to Compute Engine instances using Pub/Sub (Google, 2020q). A Cron job is a time-based job scheduler for Unix-like computer operating systems.

Figure 55: Reliable Task Scheduling on Compute Engine



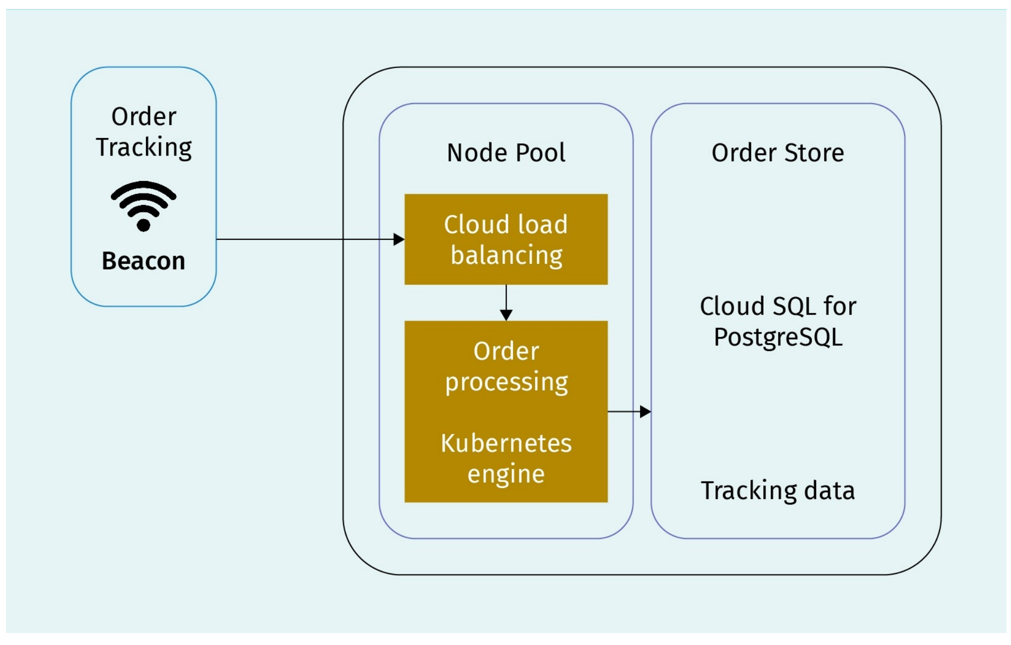
Source: Alavirad, 2020.

Database services

Google cloud provides SQL and NoSQL databases.

Cloud SQL is a fully-managed relational database for MySQL, PostgreSQL, or SQL. As a fully-managed relational database, Cloud SQL automates backups, replication, encryption patches, and capacity scaling (Google, 2020n). In the figure below, you can see an example of implementing Cloud SQL in retail. The data from beacons are sent to the OneMarket platform, which has a node pool consisting of cloud load balancing and a Kubernetes engine functioning as an “order processing” microservice. The results are then saved in the “order store,” which is a Cloud SQL database.

Figure 56: Cloud SQL in Retail



Source: Alavirad, 2020.

As a NoSQL database service, Google Cloud offers two products: Firestore, to store the document-like data (Google, 2020h), and Cloud Bigtable, for table-like data (Google, 2020c).

Firestore is a real-time, cloud-hosted NoSQL database. The database's client is notified in real time when the data inside Firestore change. The data are stored in documents (unit of storage) that contain key-value pairs and the documents themselves are organized into collections. Documents can have sub-collections and nested objects.

Each document is identified by its name. For example, a document that represents the user `AndMüller` looks like:

Code

```
AndMüller
(document)
first: “Andreas”
last: “Müller”
born: “1984”
```

Where `first: “Andreas”` is a key-value pair (field).

Documents themselves are nested inside collections. For example, we can have the `users` collection:

Code

```
users
(collection)
AndMüller
(document)
first: “Andreas”
last: “Müller”
born: “1984”
FraMüller
(document)
first: “Frank”
last: “Müller”
born: “1986”
```

Therefore, sub-collections live inside documents. A sub-collection is a collection associated with a user. For example, in the `users` collection, we can define a `messages` sub-collection for each document (user) which contains documents like `message_1`, `message_2`, and so on.

Code

```
users
(collection)
AndMüller
(document)
first: “Andreas”
last: “Müller”
born: “1984”
Messages
(subcollection)
Message1
(document)
from: ”Frank”
msg: “Hello”
```

As stated earlier, it is possible to use an arbitrary database on Compute Engine Service using persistent disks (e.g., setting up MongoDB).

Storage services

Google Cloud provides services to store media files, backups, and other file-like objects. Some of the storage services from Google Cloud are:

Cloud Storage which is a consistent and scalable large-capacity data storage available in several flavors (Google, 2020o):

- Standard storage is suitable for hot data (data that are frequently accessed) or the data which are stored for a short period. The typical monthly availability in a multi-region scenario is >99.99%.
- Nearline storage is suitable for infrequently accessed data and a storage duration of at least 30 days. The typical monthly availability in a multi-region scenario is 99.95%.
- Coldline storage is suitable for infrequently accessed data and storage duration of at least 90 days. The typical monthly availability in a multi-region scenario is 99.95%.
- Archive storage is suitable for data archiving, online backup, disaster recovery accessed data, and for a storage duration of at least 365 days. The typical monthly availability in a multi-region scenario is 99.95%.

Networking Services

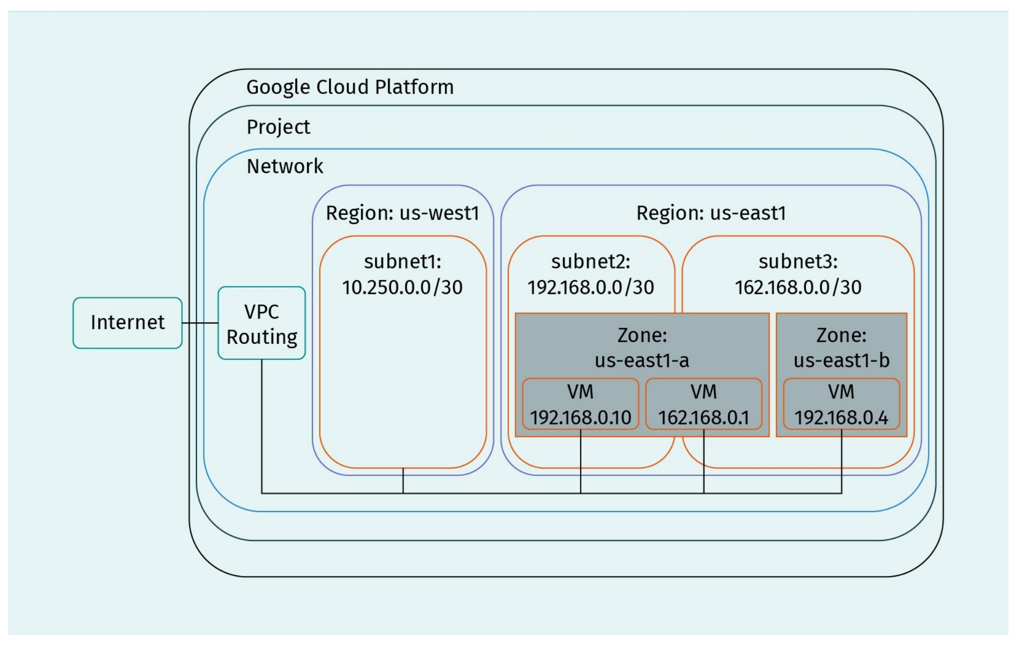
Google Cloud also provides networking services to assist users with load-balancing traffic of resources, creating DNS records, and connecting existing infrastructure to the Google Cloud.

Virtual Private Cloud (VPC) provides networking services for the virtual machines of Compute Engine, Google Kubernetes Engine (GKE) containers, and the App Engine environment (Google, 2020p). Any VPC network is a global resource. It consists of one or more regional virtual subnetworks (subnets) in data centers. All subnets are connected via a global wide-area network. All VPC networks are logically isolated. In the following figure, you can see an example of a VPC network. Inside the Cloud Platform, we have a project and inside this project, we have a VPC network spread over two regions: `us-west1` and `us-east1`, which consist of subnets and zones. As you can see, several zones of a region can share the same subnet (`subnet3` covers `us-east1-a` and `us-east1-b` zones). A zone can also be covered by several subnets (`us-east1-a` is covered by `subnet2` and `subnet3`).

In general, a VPC network provides the following functionalities (Google, 2020p):

- It connects the Compute Engine virtual machine (VM) instances, including Google Kubernetes Engine (GKE) clusters, App Engine environment instances, and other Google Cloud products built on Compute Engine VMs.
- It offers internal TCP/UDP load balancing and proxy systems for internal HTTP(S) load balancing.
- It facilitates a connection to on-premises networks by using a Cloud VPN.
- It distributes traffic from Google Cloud external load balancers to backend services.

Figure 57: VPC Network Example



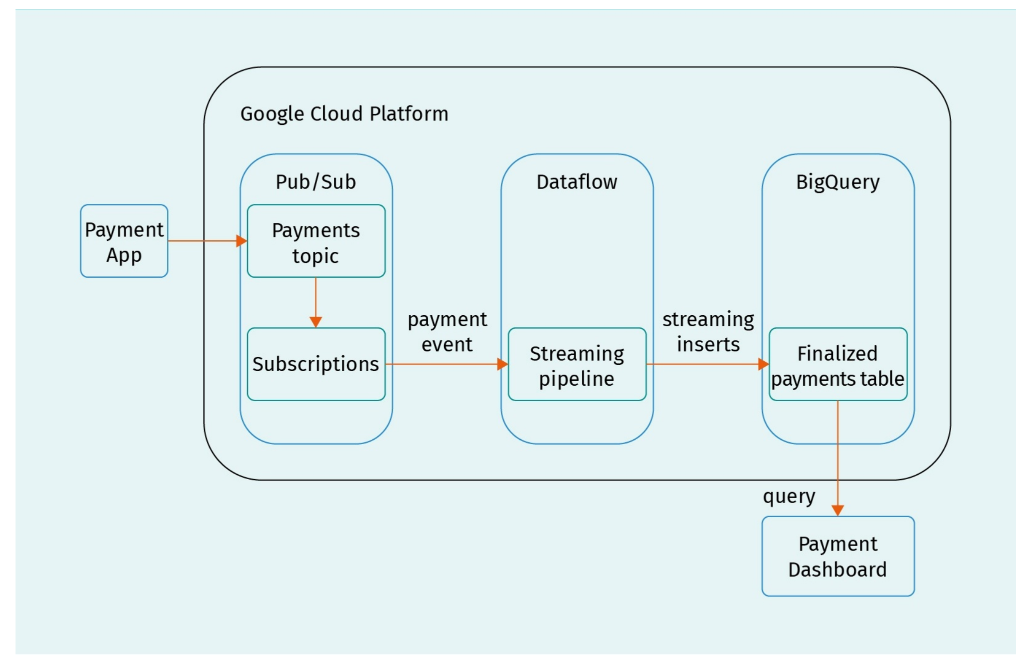
Source: Alavirad, 2020.

Big data services

Google Cloud also provides services to process and query big data in the cloud. Some of these big data services are BigQuery (Google, 2020b), Dataflow (Google, 2020g), and Pub/Sub (Google, 2020l). BigQuery is a petabyte-scale enterprise data warehouse for storing and querying massive datasets. Dataflow is a managed service and a set of SDKs for batch and stream data processing. Pub/Sub is a real-time asynchronous messaging service for sending and receiving messages between independent applications. This service decouples event producer services from event processor services.

Below, you can see a solution for processing high-priority and standard-priority payments (Google, 2020d). In this single pipeline, the payment notifications are sent by a streaming data source and are received by the Pub/Sub service. The payment information is then processed by Dataflow and is written to the BigQuery.

Figure 58: Big Data Processing Using Google Cloud Services



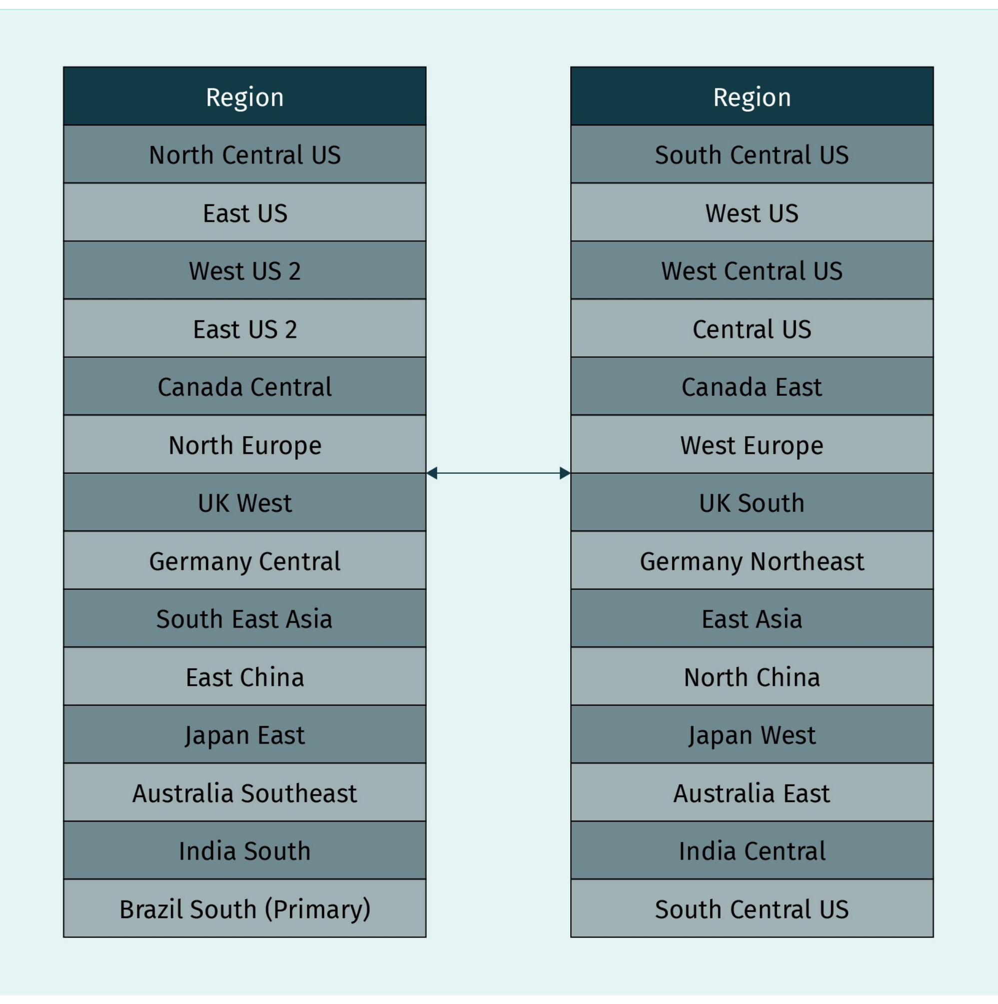
Source: Alavirad, 2020.

5.4 Microsoft Azure

Like Amazon AWS and Google Cloud, Microsoft Azure is a cloud computing platform which provides different services using infrastructure located around the globe.

In Azure infrastructure, a region is a geographical area which has one or more data centers. The datacenters of each region are connected. Azure has 60 regions. When the user selects a service, they should also select the region of this service. To decrease data redundancy, Azure implemented the region pairs concept: each Azure region is paired with another region in a similar geographical area (like the US and Europe). Azure prefers at least 300 miles of physical separation between two paired regions.

Figure 59: Region Pairs in Azure



Region	Region
North Central US	South Central US
East US	West US
West US 2	West Central US
East US 2	Central US
Canada Central	Canada East
North Europe	West Europe
UK West	UK South
Germany Central	Germany Northeast
South East Asia	East Asia
East China	North China
Japan East	Japan West
Australia Southeast	Australia East
India South	India Central
Brazil South (Primary)	South Central US

Source: Microsoft, 2020p.

Azure also divides the globe into geographies: geopolitical boundaries or country borders, which are also discrete markets. Each geography has two or more regions. The main advantages of geographies are respecting data residency, sovereignty, compliance, and resiliency requirements within geographical boundaries.

Azure geographies are divided into the following areas:

- America
- Europe
- Asia-Pacific
- The Middle East and Africa

Azure Services

Azure provides compute services to run applications.

Compute services

Azure Kubernetes Service (AKS) facilitates serverless deployment of Kubernetes applications (Microsoft, 2020h). Kubernetes is open-source software for deploying and managing microservices containers at scale. AKS also provides an integrated continuous integration and continuous delivery (CI/CD). Azure handles the health monitoring and maintenance tasks: Azure manages the Kubernetes masters and users manage the Kubernetes nodes.

The Azure Functions service provides servers for keeping users' applications running at scale (Microsoft, 2020f). The user writes a small piece of code (function) and Azure Functions runs the code. A function is triggered by an event: a response to data change and messages, scheduled triggers, or HTTP request triggers. Azure Functions supports C#, Java, JavaScript, Python, and PowerShell languages.

Database services

Azure SQL Database is a managed platform as a service (PaaS) for SQL database engines. This service handles most of the database management tasks, like upgrading, patching, backup, and monitoring (Microsoft, 2020n). This service runs on the SQL Server database engine. Using this service, users can develop a data-storage layer for applications on Azure.

There are two methods to deploy Azure SQL Databases:

- Single database, where there is a single and isolated database. This method is appropriate for microservice applications, where each microservice should have its own private and isolated database.
- Elastic databases pool, in which the several databases share resources like CPU and memory. The advantage of this model is that it allows allocating resources to a pool of databases. This allocation method results in saving in resources when the databases have unpredictable usage patterns.

Another product of Azure for databases is Azure Cosmos DB, which is a globally distributed multi-model database service. This service enables developers to scale throughput and storage across several Azure regions (Microsoft, 2020b). Cosmos DB supports different APIs like SQL, MongoDB, Cassandra, and Gremlin.

The main advantage of Azure Cosmos DB is its ability to replicate data close to the user. Therefore, application users can access the data close to their geographical location.

One use case of Cosmos DB is in the gaming industry. Mobile game applications need single-millisecond latencies for reads and writes in order to offer an engaging in-game experience. A gaming database should therefore be fast and able to handle a massive spike in requests during the first phase of application launch. Cosmos DB offers an elastic scaling up or scaling down performance to allow games to update the profiles and stats of up to millions of simultaneous users. This service also supports millisecond reads and writes to prevent latency during gameplay.

Azure Cosmos DB could also be used for event sourcing to empower event-sourcing architectures like online-shop applications. In this use case, Azure Functions microservices incrementally read inserts and updates (for example, order events) made to an Azure Cosmos DB by for another Azure Function (E-Commerce Checkout API).

Figure 60: Using Cosmos DB for an Online Shop Application

Source: Alavirad, 2020.

Storage services

Azure Storage Platform is Microsoft's modern storage solution for data storage scenarios. This platform is composed of the following services: a massively scalable object store for data objects (Azure Blob), a managed file share for cloud or on-premises deployments (Azure Files), a NoSQL store for schema-less storage of structured data (Azure Tables), block-level storage volumes for Azure virtual machines (Azure Disks), and a messaging store for reliable messaging (Azure Queues) (Microsoft, 2020o).

Azure Blob is Microsoft's service for the storage of object files. This storage service is optimized for large unstructured data. Access to Blob objects is possible via HTTP/HTTPS using Azure Storage's REST API, Azure PowerShell, Azure CLI, or an Azure Storage client library (Microsoft, 2020a).

1. The storage account is a unique namespace in Azure for user's data. Every object that is stored in Azure Blob has a unique address starting with the account name. For example, if the storage account is called `user1storageaccount`, the default endpoint of the Blob Storage would be:
   `http://user1storageaccount.blob.core.windows.net`
2. The container is similar to a file system and organizes the blobs. A storage account can have an unlimited number of containers.
3. Azure Blob provides blobs, of which there are three types:
   - Block blobs, which are used for storing binary and text data up to 4.75 TB.
   - Append blobs, which are suitable for append operations.
   - Page blobs, which are used for storing random access files up to 8 TB.

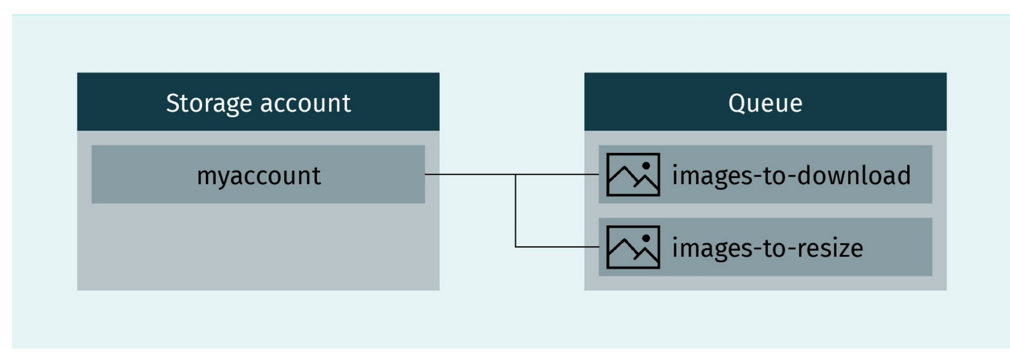
Figure 61: Storage Account, Containers, and Blobs in Storage Blog Service

Source: Alavirad, 2020, based on Microsoft, 2020a.

Azure Queues Storage is an Azure solution for storing large numbers of messages. The stored data on the Queue Storage can be accessed via HTTP/HTTPS using authenticated calls. The size of a queue message can be up to 64KB and a Queue Storage can contain millions of messages. This service is used for creating backlogs of work for asynchronous processing (Microsoft, 2020j). The building blocks of Queues Storage are:

1. The storage account is used to access the Azure Storage platform, including Azure Queues Services.
2. The URL format is used to address queues; such as
   https://<storage account>.queue.core.windows.net/<queue>
3. A Queue is the storage unit and contains messages.
4. Messages are stored in the Queue Storage

Figure 62: Storage Account, Queue, and Messages in the Azure Queue Storage Service



Source: Alavirad, 2020, based on Microsoft, 2020j

Security services

Azure provides several services for security, such as:

- Azure Sentinel, which is a security information event management (SIEM) and security orchestration automated response (SOAR) service (Microsoft, 2020m).
- Azure Information Protection, which enables enterprises to classify and protect their documents and emails by labeling them automatically or manually (Microsoft, 2020g).
- Azure Dedicated **HSM** (hardware security module), which is a cryptographic key storage service (Microsoft, 2020e).
- Azure Security Center for IoT, which is a cloud-based security service for end-to-end threat detection within hybrid cloud workloads and IoT solutions (Microsoft, 2020l).

HSM
This stands for hardware security module, and is a physical element that provides extra security. It does this by managing digital keys, encryption and decryption, and authentication for the use of applications (Rouse, 2015).

Here, we will focus on Azure Security Center for IoT. This service is composed of the following services:

- IoT hub integration
- device agents (optional)
- send security message SDK
- analytics pipeline

The workflow of Azure Security Center for IoT could be as follows (Microsoft, 2020k):

1. Turn on the security option in the IoT hub and install Azure Security Center for IoT device agents on IoT devices.
2. Azure Security Center for IoT device agents collects, aggregates, and analyzes raw security events from IoT devices. Raw security events could be IP connections, process creation, or user logins.
3. Device agents utilize the Azure send security message SDK to send security data into Azure's IoT hub.
4. The IoT hub receives the security information and sends it to the Azure Security Center for IoT service.
5. Azure Security Center for IoT's analytics pipeline receives and analyzes all received security information from different sources in Azure. The analyzed security information is sent to the Azure Security Center for IoT.
6. Azure Security Center for IoT generates recommendations and alerts based on the analyzed security information received from the pipelines.
7. The alerts and security recommendations are written to the user's Log Analytics workspace.

Analytics service

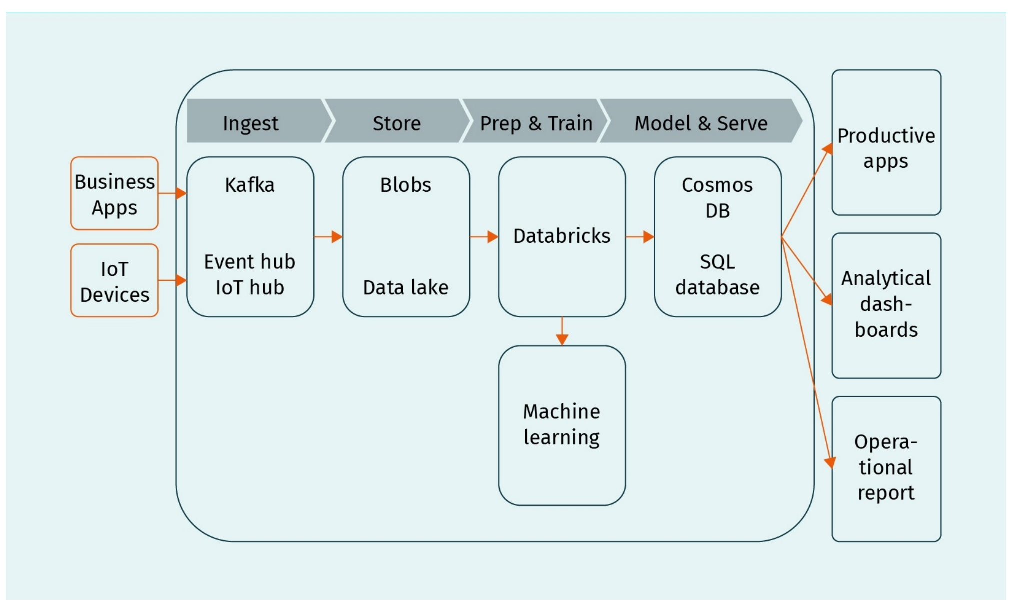
Azure also provides several analytics services, such as:

- Azure Data Explorer, which is a service used to analyze data streaming from different sources like applications, website, and IoT devices in real-time (Microsoft, 2020c).
- Azure Databricks, which is a platform for data analysis on Azure based on Apache Spark (Microsoft, 2020d).
- Azure Data Lake Analytics, which is an on-demand, distributed, cloud-based data processing architecture for big data analysis.

We will now discuss Azure Databricks in more detail. As stated above, this service integrates Apache Spark into the Azure Cloud Platform. The workflow of this big data analysis process is as follows:

1. The raw data are ingested into Azure. For this purpose, we can use Azure Data Factory (for batch ingestion) or Apache Kafka, Azure Event Hub, or Azure IoT Hub (for streamed, near-real-time ingestion).
2. The data are stored in Azure Blob Storage or Azure Data Lake Storage.
3. The data will be analyzed using Azure Databricks, empowered by Apache Spark.
4. The analyzed data will be written to databases like Azure Cosmos DB, Azure SQL Database for modeling, and storage.
5. The applications which are written in Azure can access the analyzed data to visualize and report the analysis results.

Figure 63: Big Data Analysis Workflow Using Azure Databricks

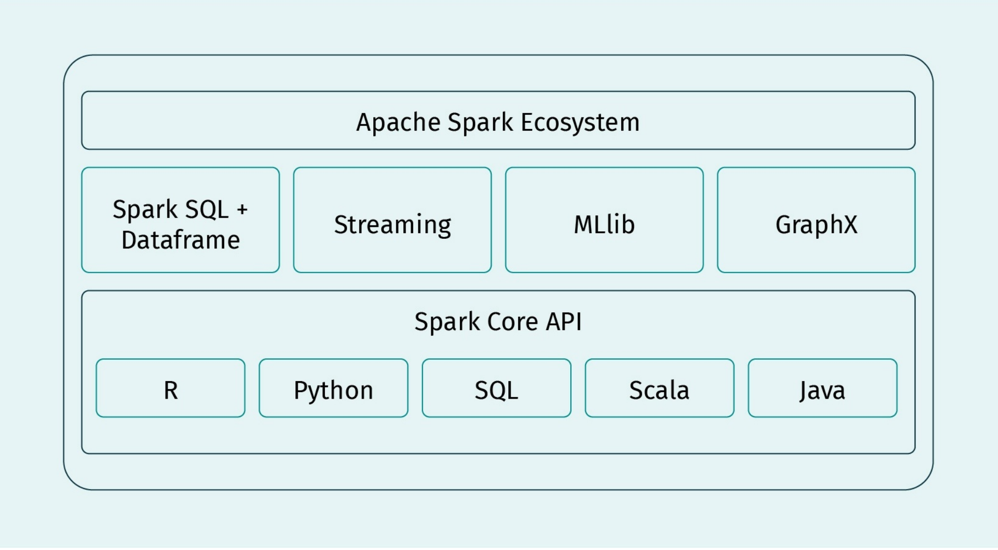


Source: Alavirad, 2020.

The Apache Spark implemented in Azure Databricks has the following components (Microsoft, 2020d):

- Spark SQL, which is a module for managing structured data
- the dataframe, which is a distributed collection of structured data organized on columns with unique names, equivalent to the Python dataframes
- Streaming, which is used for processing and analyzing of real-time data from different sources
- MLlib, which is a machine learning library for classification, regression, clustering, collaborative filtering, dimensionality reduction, and optimization primitives
- GraphX, which is Apache Spark's API for graphs and graph-parallel computation
- Spark Core API, which is used for supporting languages like Python, R, Scala, SQL, and Java

Figure 64: Azure Spark Ecosystems in Azure Databricks



Source: Alavirad, 2020, based on Microsoft, 2020d.

SUMMARY

In this unit, we have learned about three cloud computing platforms: Amazon AWS, Google Cloud Platform, and Microsoft Azure.

We started with a brief introduction into cloud computing and the different cloud delivery systems: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). With IaaS, the user has more control over the resources but also more responsibility for the maintenance, while with SaaS, the user has the least control over the resources but the cloud service provider takes control of all infrastructure and application maintenance.

Following this, we started with a brief introduction of each of these platforms and discussed some of the services each platform provides.

UNIT 6

DATA OPS

STUDY GOALS

On completion of this unit, you will have learned …

- about DevOps, DataOps, and the principles of DataOps.
- about MLOps and the various stages of an operationalizing data science project plan.
- what the application containerization deployment method is.
- what Docker and Kubernetes are.
- about the machine learning pipeline and its seven components.
- about the architecture of a typical machine learning pipeline.
- about Michelangelo ML workflow, an Uber product.

6. DATA OPS

Introduction

The main focus of the ~~final~~ unit of this book is not designing data systems, but operationalizing developed data analysis and machine learning solutions.

We start this unit with an introduction to DevOps, DataOps, and MLOps. In this section, we will discuss the eighteen principles (manifesto) of DataOps and the different phases of an operationalizing a data science project plan, i.e., building, managing, deploying, and monitoring.

We will then discuss the containerization approach to deploying and integrating data science pipelines into business applications and Docker, a tool used to containerize applications. We will introduce Kubernetes as a platform for managing the deployed containerized application.

Finally, we will introduce the machine learning pipeline and its seven steps. We will also briefly discuss Michelangelo, Uber's machine learning workflow.

6.1 Defining Principles

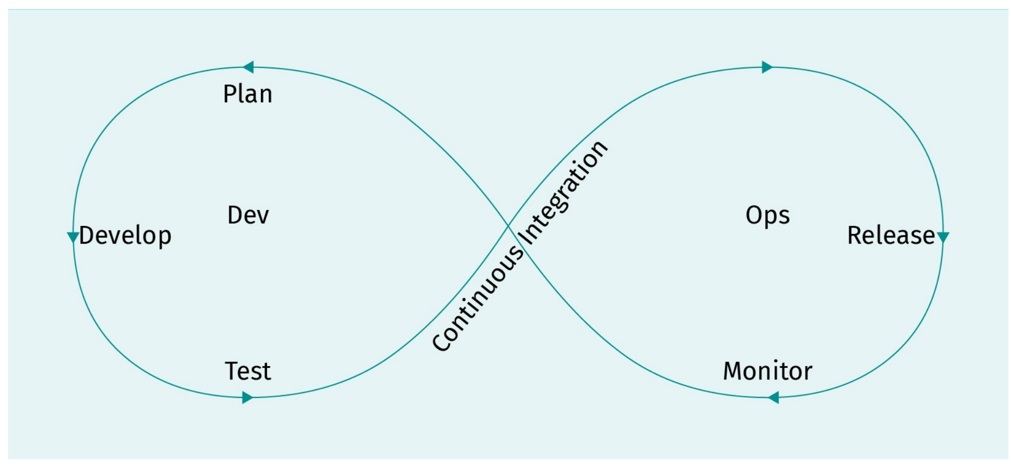
Reducing the time between the end of the development phase and the business realization phase is the aim of any successful software business. At the O'Reilly Velocity Conference in 2009, Flickr engineers John Allspaw and Paul Hammond held the presentation "10+ Deploys per Day: Dev and Ops Cooperation at Flickr." In their presentation, they demonstrated the interplay between development and operations teams during a typical software deployment process. They showed each team blamed the other team for any problems: "It's not my code, it's your machines!" They suggested that having seamless, transparent, and fully integrated application development and operations activities is the only way to avoid such struggles (Allspaw & Hammond, 2009). Nowadays, the idea they presented is known in the IT world as DevOps (Mezak, 2018). DevOps (a contraction of development and operations) is a set of practices used to reduce the time to market and achieve continuous integration and continuous delivery (CI/CD).

Continuous integration is the development method of continually integrating new code into the shared repository to avoid "integration hell," where many developers try to perform changes at the end of a sprint.

After changes (code) are implemented, continuous delivery strives to build, test, and release software with one-click deployments from development into operational environments. To accomplish continuous delivery, DevOps engineers must place every project

component, such as code, tests, and deployment instructions, into the source control (Altis.com, 2020). This method has been used by companies such as Facebook and Amazon to push releases as often as every minute.

Figure 65: The Lifecycle of DevOps



Source: Alavirad, 2020.

As the first aim of DevOps is to enhance the life cycle of a general software product by shortening the time to market after the development phase, DataOps was introduced to improve the quality of the data analytics initiatives. DataOps is a combination of tools and methodologies used to optimize the data analytics cycle time and quality (Bergh et al., 2019). DataOps is applied to the entire data analysis life cycle, from data generation and cleansing to data visualization and reporting.

Gartner (2020) defined DataOps as “a collaborative data management practice, really focused on improving communication, integration, and automation of data flow between managers and consumers of data within an organization” (para. 1)

In other words, DataOps reevaluates elements in an analytics project where (Bergh et al., 2019):

- individuals and interactions have more importance than processes and tools
- functioning analytics have more value than comprehensive documentation of the analytics solution itself
- the collaboration and engagement of the client has more value that the terms written in the contract
- experimentation, iteration, and feedback from different parties of an analytics project have more value than an enormous, upfront design
- the cross-functional responsibility of operations is more valuable than the siloed responsibilities

DataOps is based on eighteen principles (its manifesto), which are as follows (Bergh et al., 2019):

1. Continuous customer satisfaction: the highest priority of an analytics project is customer satisfaction by early and continuous delivery of valuable analytical insights over periods ranging from minutes to weeks.
2. Providing functioning analyses: the most important measure of data analytics performance is an insightful analysis, which is built on robust frameworks and systems.
3. Openness to change: the team should embrace customers' dynamic and evolving needs.
4. DevOps is a team sport: analysis teams always have a range of roles, skills, favorite tools, and titles. This variety and diversity of backgrounds and views enhances both innovation and creativity within the data science project.
5. Daily discussions: clients and DevOps teams must work together closely every day during the analytics project lifetime.
6. Self-organization: the best analytical insights, algorithms, architectures, requirements, and designs come from self-organized teams.
7. No heroism: as the speed and scope of analytical insight requirements increase continuously, the analytical teams should aim to minimize heroism and instead build sustainable and scalable teams and processes.
8. Reflection: analytical teams should optimize their performance by periodically reflecting feedback from customers and themselves as well as operational statistics.
9. Analysis is code: analytical teams use different tools and methods to integrate, model, visualize, and access data. Each of these tools and methods produces code and configuration data that describe the actions that should be taken in order to gain insight.
10. Orchestration: the consistent orchestration of data, tools, code, and teams is a vital element for a successful analytics project.
11. Reproducibility: since reproducible results are indispensable, the team should version everything, including data, hardware, and software configurations, as well as the code and configuration for each tool in the entire toolchain.
12. Disposable working environments: it is important to minimize the experimentation effort for team members by providing them easy-to-create, isolated, and secure development environments that closely mimic the production and operation environment.
13. Simplicity: a persistent emphasis on technical innovation and sound engineering increases agility, and simplicity (the art of minimizing unnecessary work) is essential.
14. Analytics is manufacturing: analytics pipelines are similar to **lean production** lines. We believe that a basic principle of DataOps is a focus on process thinking, which strives to continually enhance the delivery of analytical insight.
15. Quality above all: analytical pipelines should be constructed on solid bases that are capable of automatically detecting anomalies in code, configuration, and data. They should also deliver continuous feedback to the team for error prevention.
16. Quality and performance monitoring: it is essential to continuously measure and monitor performance and quality indicators to detect unexpected deviations and also to generate operational statistics.
17. Reusability: a basic principle of efficiency in gaining analytical insight is not to repeat previous works by individuals or teams.
18. Improving lead times: we should try to reduce the amount of time and work needed to transform a customer need into an analytical blueprint, implement it in the development phase, make it available as a reproducible product, and, finally, improve and reuse the end product.

Lean production
The economical and time-efficient use of operating resources, personnel, materials, and organization in company activities is called lean production.

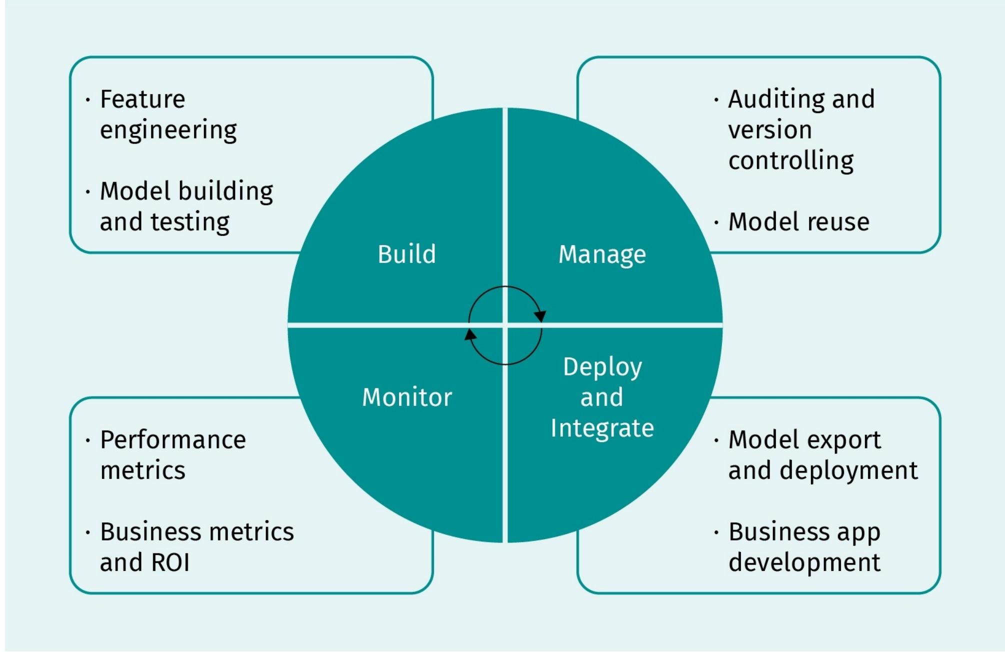
Implementing the DataOps and DevOps practices into machine learning (ML) and data science pipelines is called MLOps, which is defined as the process of operationalizing data science by getting ML models into production in order to monitor their performance and ensure they are fair and in compliance with applicable regulations (Sweenor et al., 2020). MLOps provides repeatable processes to deploy, monitor, and maintain the machine learning and data science pipelines in operational systems.

Assume that you have developed a machine learning application within your enterprise. To check if the application is MLOps compatible (having not only ML models, but also ML operations), you should answer questions like (Sweenor et al., 2020):

1. Are you able to manage the development of new models parallel to the existing production and operational models?
2. Can you explain why specific predictions are made by your model?
3. Have you versioned your code and implemented approval processes to support your ML operations?
4. Are application developers, DevOps engineers, IT managers, and line-of-business managers involved in the life cycle of the product?
5. Do you consider the whole process as a life cycle?

As stated in the Introduction, this unit is mostly about operationalizing data science. Below, you can see different phases of an operationalizing data science project plan. Such a plan is necessary to implement the data science project for analysis and predictions of real-world scenarios before they become outdated due to operational obstacles (Sweenor et al., 2020).

Figure 66: Steps to Operationalize Data Science



Source: Alavirad, 2020, based on Sweenor et al., 2020.

The data science model and all its transformations (e.g., the transformation of the raw data to be more appropriate as an input for the model) must be managed as an artifact throughout the pipeline and the life cycle of the data science project. To realize this, the team should consider the following steps (Sweenor et al., 2020):

Feature engineering
The process of using domain knowledge of the data to create features that make machine learning algorithms work is called feature engineering (Shekhar, 2018).

1. Build. The first step of operationalizing a data science project is to build analysis pipelines using programming languages like Python and R. To create a predictive data analysis pipelines, data scientists use machine learning standards. To improve the quality of predictions of the machine learning models, data scientists build transformations by performing **feature engineering**.
2. Manage. The data science models should be managed during their life cycles. Using a central repository makes the management process more efficient. On such a platform, it is possible to track the provenance, versioning, testing, deployment, accuracy metrics, links between models, datasets, etc., of the models.
3. Deploy and integrate. In this step, the data science pipeline is integrated into the business application environment. The deployed pipeline should also be consistent with the host runtime environment. There are different deployment methods. For example, modern MLOps architectures provide containerized deployment of the data model pipelines. There is a difference between deployment and integration (Sweenor et al., 2020). Deployment is the process of detaching the developed model from the development environment and execute it in a format to be appropriate for the host runtime environment. Integration is the process of embedding the deployed model into the runtime environment.
4. Monitor. After the data science model has been deployed and started its operational phase, it is necessary to monitor it constantly to evaluate the accuracy of its predictions and its business criteria.

6.2 Containerization

We have already introduced containers as a method of deploying and integrating data pipelines into business applications such as (Sweenor et al., 2020):

- an execution endpoint like a database, in order to analyze the data stored in the database
- a website, in order to enhance the user experience
- an IoT device, in order to monitor and predict the machine condiction

As stated before, deployment is followed by integration. Deployment methods depend on integration endpoints (targets or hosts) and vice versa. Among these deployment methods, we can mention (Sweenor et al., 2020):

- code generation, such as with C++, Python, Java, or C#
- serverless deployment, like Google Functions (function as a service)
- container deployment, like model inferencing within a Docker

For integration endpoints there are also many methods, like (Sweenor et al., 2020):

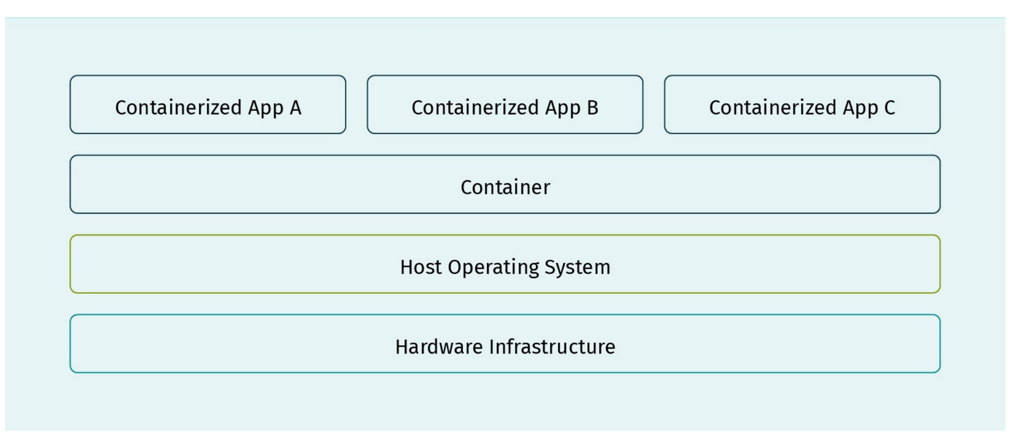
- Batch option. This is when, for example, one collection of data will be analyzed at a time.
- Interactive apps. This is where the model is run via an API.
- Real-time streaming. This is where the data will be analyzed as they are generated in real-time.
- Edge integration. This is where the model is executed on edge devices (like IoT devices).

In the rest of this section, we focus on the container method for the deployment of data science models.

Container

A container is a standard unit of software that wraps up the application code and all its dependencies (e.g., libraries) in a single package (container), so that the application can run and move between different environments (Docker, 2020b). In other words, containers abstract the applications from the runtime environments (hosts). They provide a deployment approach through which the development team can focus on their task while the IT-operational team manages the deployment of the application in different environments.

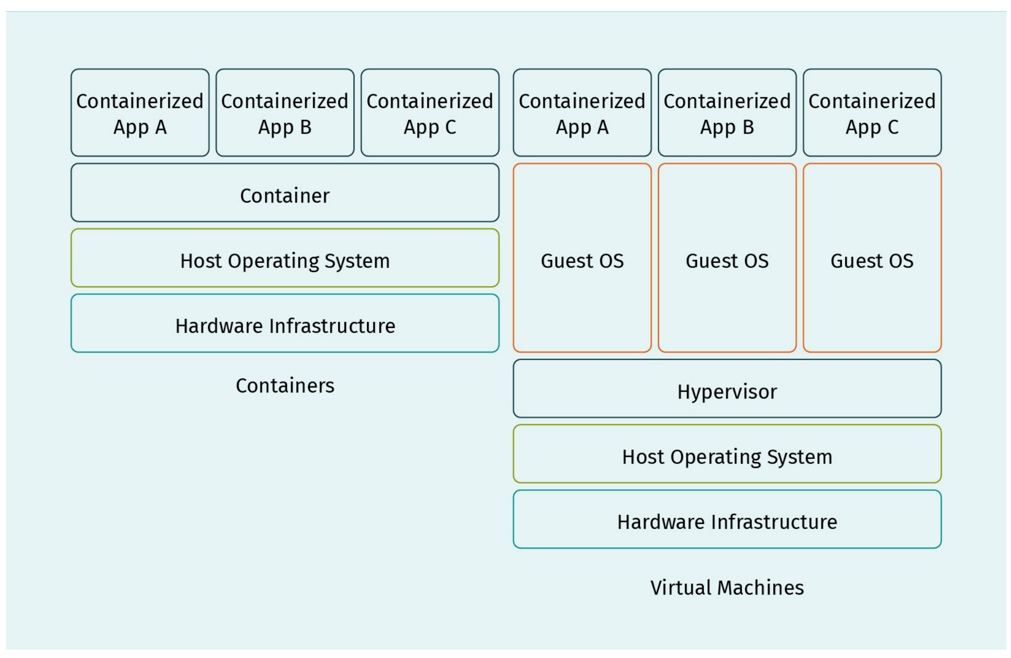
Figure 67: Containerization of Applications



Source: Alavirad, 2020.

Containers and virtual machines (VMs) have some similarities and differences. A virtual machine is a guest operating system (e.g., Linux) which is running on the top of a host operating system (e.g., Windows). VMs, like containers, isolate applications and their libraries from the host environment. However, containers provide a much lighter isolation mechanism with smaller disk occupation and lower overhead, as you can see in the figure below. Indeed, VMs virtualize at the hardware level, while containers perform the virtualization at the operating system level. Multiple containers run at the top of the kernel of the host operating system. The containers share the operating system kernel and they use less memory than the VMs to boot.

Figure 68: Containers vs. VMs



Source: Alavirad, 2020.

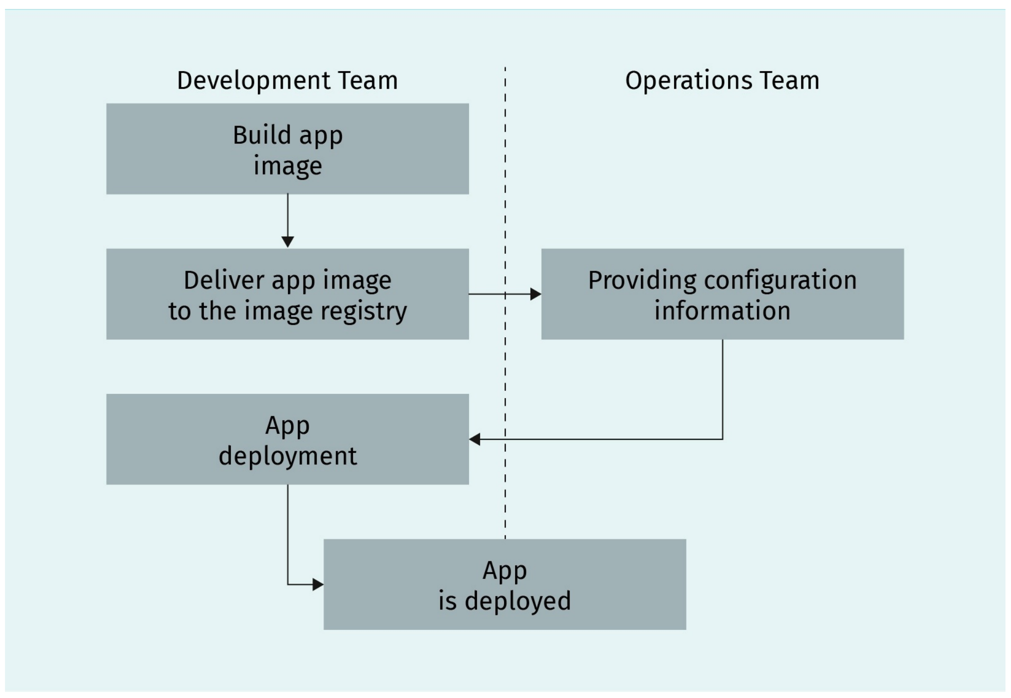
There are different forms of containers and, in the following, we will discuss two of them: Dockers and Kubernetes. The former is applied to containerize applications, and the latter manages the containerized applications in operational environments.

Docker

Docker is an open-source platform that provides the encapsulation of applications' code and dependencies in containers (IBM, 2020b). Docker containers can run on-premises or on on-demand hosts. It was originally developed for Linux operating systems, but now is also available for Windows and MacOS (using a virtual machine).

Docker containerization can facilitate both workflow and communication between the development and IT operation teams, as you can see in the figure below (Matthias & Kane, 2018). In this workflow, the development team builds the Docker image and delivers it to the registry. The IT operation team then provides the configuration information to the development team and they trigger the development. This simplification is possible because Docker permits the discovery of dependency issues during the development and test phases. Therefore, when the application is ready for deployment and integration, all possible issues have been resolved during the development and test phases.

Figure 69: Application Deployment Workflow Using Docker



Source: Alavirad, 2020, based on Matthias & Kane, 2020.

A simplified version of the Docker architecture could be assumed as a server/client architecture as shown below (Matthias & Kane, 2018). In this model, Docker is assumed as a server that performs building, running, and managing of containers and also a client which is used to manage and configure the server. There is also an optional document called the registry, which is used to store Docker images and the relevant metadata. It is possible to run the Docker daemon on any number of servers.

Figure 70: Client-Server Model of Docker Architecture

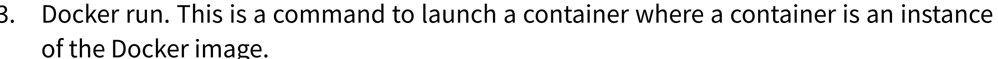
Docker client on local machine
Docker server on Linux host or VM
Docker registry on Linux host, VM, SaaS

Source: Alavirad, 2020, based on Matthias & Kane, 2020.

Let us introduce some of the main Docker components (Yegulab, 2019):

1. Dockerfile. This is a text configuration document that contains some information about how to build the Docker image such as host operating system, environmental variables, file locations, network information. Docker can build an image using the information provided in a Dockerfile.
2. Docker image. After providing the Dockerfile, the user can build a docker image using `docker build` instruction. A Docker image contains the components to run an application as a container (code, configuration files, libraries, environmental variables, etc.). The Docker image has multiple filesystem layers where the most top layer is a writeable/readable on the top of read-only layers. The filesystem layers are build by the command line (instructions) in the Dockerfile during the Docker image building process. The instructions `RUN`, `COPY`, and `ADD` create the layer of Docker images.

3. Docker run. This is a command to launch a container where a container is an instance of the Docker image.



4. Docker engine. This is the core of the Docker containerization platform to create and run container applications (Docker, 2020a). Docker engine acts as a client-server application. The server is a long-running daemon process (`docked`) and the client is a command-line interface (`docker`).

To better understand the concept of Docker, let us look at a typical Docker workflow (Matthias & Kane, 2018):

1. Revision Control. There are two forms of revision control provided by Docker. One is for tracking the filesystem layers (containing container images), the other one is for tagging the container images.

The filesystem layer structure facilitates the changes to the container application. If a change is necessary (and also applied), only the modified layers should be deployed, where each layer is identified with a hash. Docker uses as many as possible base layers and rebuilds only the layers affected by the code modifications.

Docker has an image tagging mechanism at the deployment time to provide application versioning. Using image tags we could find out for example what was the last version of the application that was deployed.

2. Building. The building of an image from the Dockerfile is performed by the `docker build` command.
3. Testing. Although Docker does not provide a specific testing mechanism, if developers do some unit or full-integration testing, Docker guarantees that the tested version will be delivered for deployment and integration. For example, if we have a successful unit test against a Docker image, we could be sure that there would not be any problem with the versioning of the underlying library of the Docker application during the deployment.
4. Packaging. Docker produces a single, multilayered Docker image.
5. Deploying. The deployment of the Docker applications would be handled by Docker standard client on a single host and there are tools for deploying the Docker applications on multiple nodes.

Kubernetes

We have now taken a look at how to use Docker to create containerized applications. The next step after deploying the application is to manage those applications in the production and operational environments. For example, when one container is down, another container should be started automatically. This container management task could be handled by Kubernetes. Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services that facilitates both declarative configuration and automation (Kubernetes.io, 2020). Google released the Kubernetes open-source project in 2014.

Kubernetes can be considered as a:

- container platform
- microservices platform
- portable cloud platform

Kubernetes provides a container-centric management environment. It coordinates the computing, networking, and storage infrastructure of user's workloads. This offers the simplicity of Platform as a Service (PaaS) together with the flexibility of Infrastructure as a Service (IaaS) and facilitates portability between infrastructure vendors.

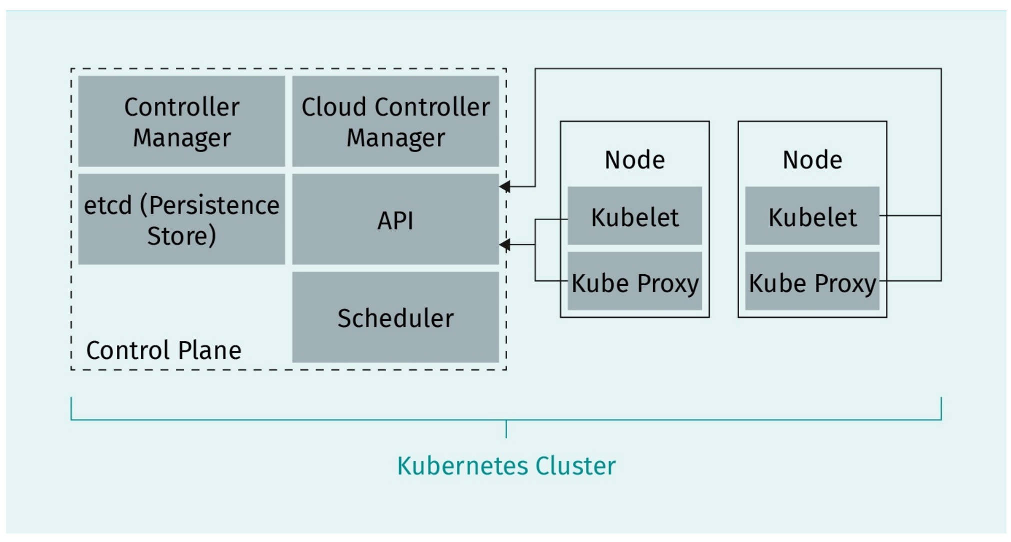
By deploying Kubernetes, we create a Kubernetes cluster which consists of nodes, each run containerized applications (Kubernetes.io, 2020). The application workloads (called **pods**) are hosted by nodes and both pods and nods are managed by the control plane in the cluster.

Pod
The smallest deployable units of computing that you can create and manage in Kubernetes are called pods.

To providing fault tolerance and also high availability in production environments, the control plane runs over multiple computers, where a cluster runs multiple nodes.

In this part, we explain the components needed to have a complete and working Kubernetes cluster. These components of the Kubernetes cluster are shown below.

Figure 71: Kubernetes Cluster Components



Source: Alavirad, 2020, based on Kubernetes.io, 2020.

Control plane components are responsible for making global decisions about the cluster (e.g., scheduling) and for recognizing and reacting to cluster events (starting a new pod if the replicas field of a replication controller is not satisfied). Control plane components can run on any computer in the cluster. For simplicity, setup scripts typically start all control plane components on the same machine where no user containers run on that machine. The components of the control plane are as follows (Kubernetes.io, 2020):

- kube-apiserver. This makes the Kubernetes API available and it is the front-end for the Kubernetes control plane. It is designed for horizontal scaling, i.e., it scales up by providing more instances.
- etcd. This is the key-value storage used as backup storage for all cluster data.
- kube-scheduler. This monitors new pods without nodes and selects the node on which the new pods should be executed.
- kube-controller-manager. This runs the controller processes. Each controller is a distinct process, but to make things easier, all controllers are grouped into a single binary file and executed in a single process.
- cloud-controller-manager. This allows users to link the cluster into the cloud provider's API.

On each node, two components run to maintain the running pods and providing the Kubernetes runtime environment.

- kubelet ensures that containers in a pod are running.

- kube-proxy enables Kubernetes service abstraction by maintaining network rules on the host and handling connection forwarding.

6.3 Building Data Pipelines

We start this section with a brief introduction to machine learning (ML) pipelines. We will then discuss the architecture of the ML pipelines in more detail.

Machine Learning Pipeline

An ML pipeline is an independently executable workflow of a complete machine learning task. Subtasks are encapsulated as a series of steps in the pipeline. The input of each step of an ML pipeline is the output of the preceding step (Microsoft, 2020i). Unlike general data science process models that typically emphasize the iterative and cyclical nature of designing a data science solution, a pipeline operationalizes and automates the necessary steps to to run a model in a production environment.

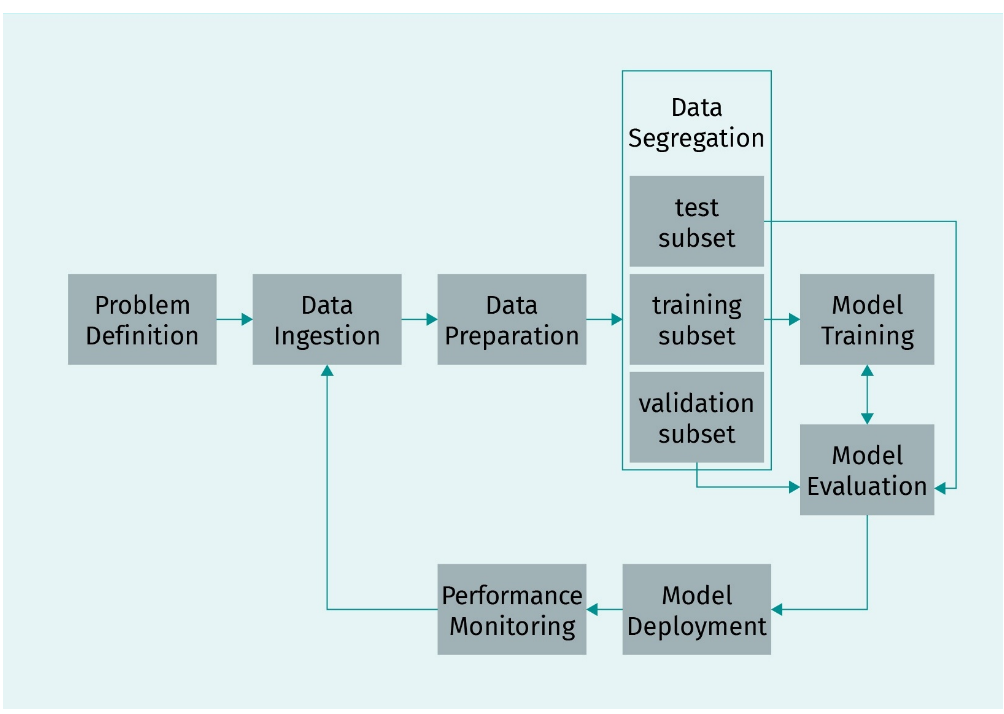
The key benefit of using an ML pipeline is automating the machine learning solution life cycle. When new data are available, they will be processed automatically, which reduces the cost of data science projects (Hapke, 2020).

We can list the main steps of a general machine learning pipeline (Koen, 2019b):

1. Problem definition. The first step is to define the problem which should be tackled by a machine learning solution/project.
2. Data ingestion. The next step is to recognize and collect the data to feed our machine learning application.
3. Data preparation. After ingesting raw data into the model, and before feeding them into the ML algorithm, they should be prepared. Invalid and duplicated data should be removed and some data need to be reformatted.
4. Data segregation. In this step, the data will be split into subsets to train, test, and validate the model against new data.
5. Model training. The training subset from the data segregation step will be used by the ML algorithm to recognize the pattern.
6. Candidate model evaluation. In this step, the test and validation subsets from the data segregation step will be used to evaluate the prediction of the ML algorithm. This step is iterative and will be executed until the most optimal model (defined by the model/problem criteria) is marked.
7. Model deployment. After finding the appropriate model, it should be deployed into an operational environment.
8. Performance monitoring. After deploying and integrating the ML model, it should be monitored continuously to evaluate its predictions in real-world scenarios.

These steps are shown below.

Figure 72: ML Pipeline



Source: Alavirad, 2020, based on Koen, 2020b.

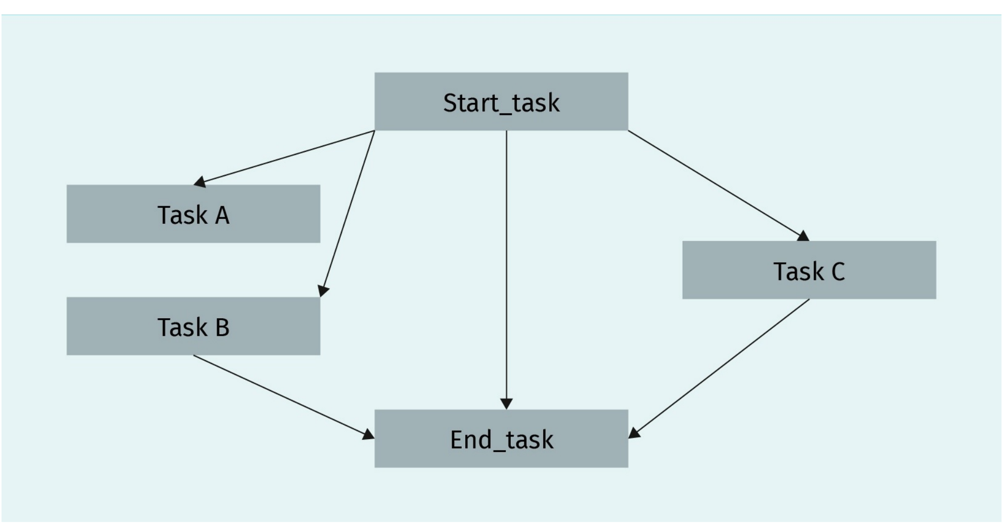
All the steps mentioned above should be orchestrated (e.g., the steps should be executed in the correct order) ~~in order~~ to have a reliable automated machine learning project. For example, a step cannot be executed before the input has been generated by the previous step (Hapke, 2020). Such orchestration can be done via many tools, such as Apache Beam, Apache Airflow, Kubeflow Pipelines for Kubernetes infrastructures, and MLflow.

Most of the ML pipeline orchestration tools (e.g., Apache Beam and Apache Airflow) and platforms (e.g., Kubeflow Pipelines) use a graph representation of task dependencies to manage the tasks in an ML pipeline. This graph representation is called a directed acyclic graph (DAG): a graph that has a finite number of vertices (a step in ML pipeline) and edges (dependencies between steps), where it is not possible to start at any vertex, follow a consistently-directed sequence of edges, and return to the same vertex. In a DAG, the edges flow in only one direction (they are directed). This starts with a start task and ends with an end task

DAG implementation guarantees that (Hapke, 2020):

- a task does not start without all dependencies are available (executed and calculated). For example, training of the model only happens after creating a training data subset.
- a graph is not linked to a previously completed graph. This results in the pipeline running until it reaches the end step.

Figure 73: A Directed Acyclic Graph



Source: Alavirad, 2020, based on Hapke, 2020.

In this section, we will briefly discuss Kubeflow Pipelines, which is a ML pipeline orchestration platform. The Kubeflow Pipelines platform consists of the following components (Kubeflow, 2020):

- User interface (UI). This component is used for monitoring and managing jobs.
- Engine. This is used for scheduling multi-step machine learning workflows.
- Software development kit (SDK). This is used for defining and configuring the pipelines and building Docker containers.
- Notebooks. This is used for interacting with systems using the SDK.

After installing Kubeflow Pipelines, some extra tools will be installed, like a workflow controller, an SQL database instance, and an ML metadataStore (which stores the metadata of a machine learning pipeline). When Kubeflow Pipelines is running, each component runs as its own Kubernetes pod.

Figure 74: Overview of the Kubeflow Pipelines

Source: Alavirad, 2020, based on Kubeflow, 2020.

Kubeflow Pipelines uses **Argo** to orchestrate dependencies of the individual components.

Argo
A collection of tools for managing workflows developed originally to manage DevOps tasks, Argo manages all tasks as containers within the Kubernetes environment (Argo, 2020).

ML Pipeline Architecture

As, nowadays, business operations are ~~very~~ time-sensitive, machine learning solutions should also be able to make predictions in real time. For example, when you purchase an item on Amazon, Amazon's machine learning platform should suggest new items based on your previous purchases and your search history.

We can break this real-time machine learning pipeline into two separate layers (Koen, 2019a):

- Online model analytics. In this layer, the ML model is applied to streaming data for real-time decision making.
- Offline data discovery. In this layer, the ML model will be trained by analyzing the historical data.

Considering this online-offline splitting approach, we will discuss the architecture and steps of a machine learning model for a real-time business application in more detail (Koen, 2019a):

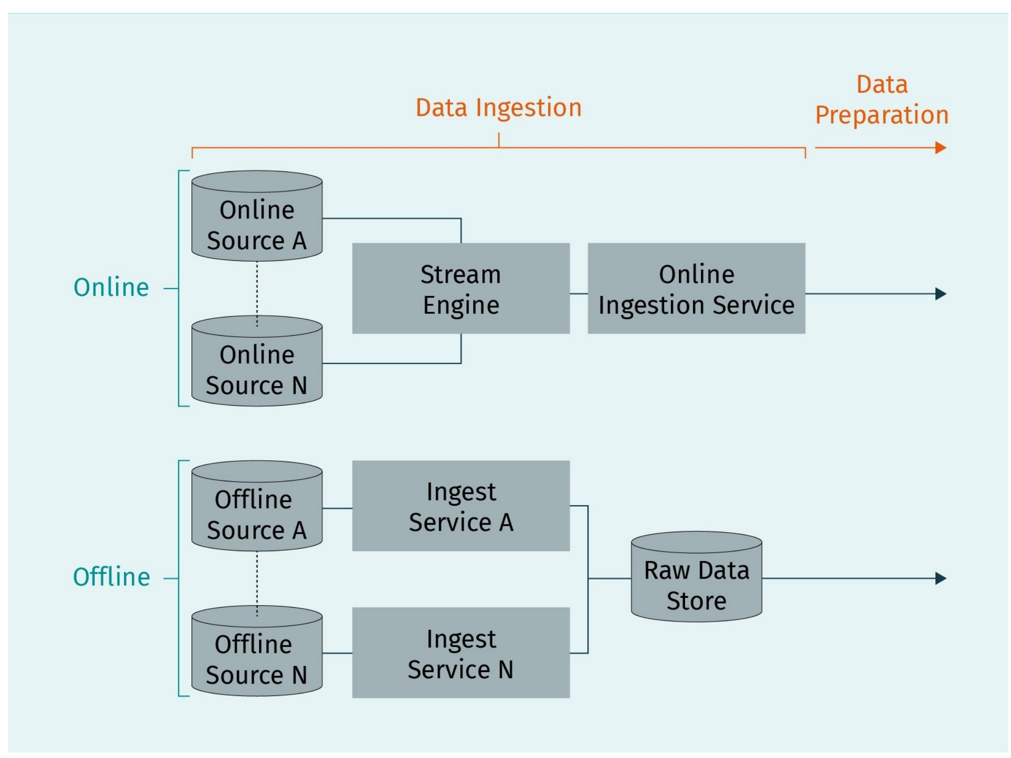
1. Data ingestion. The first step is to import the raw data into the application without any transformation. This is important because the original data should be recorded immutably. The data could be ingested from different data sources by request (pub-sub) or they can be streamed into the application. This step has both offline and online layers.
   a) Offline layer. In the offline layer, data is collected via an "ingestion service," which runs on a schedule or trigger. The data is stored in a database ("raw data store") and is labeled by a batch_id. To accelerate the data ingestion process, there is an "ingestion service" for each data source. Within each pipeline, the data are partitioned to decrease total run time using multiple server cores.
   b) Online layer. In this layer, data is ingested directly from data sources into the processing and storage components by the "online ingestion service." In this layer, data are stored on "raw data storage," as in the case of offline ingestion. At the same time, they are ingested into the processing components. An example of such an online ingestion service is Apache Kafka, which is a pub-sub messaging system, and Apache Flume, which is used for the long-term storage of data in databases.
2. Data preparation. In this step, ingested data are investigated for format differences, outliers, trends, missing values, and anomalies. The feature engineering process is also a part of this step, which includes extraction, transformation, and selection processes. This step has also online and offline layers.
   a) Offline layer. After the data are ingested into the raw data store, "data preparation service" is triggered to apply feature engineering on the raw data. The generated features are stored in the feature data store. In this step, the data are also partitioned to decrease the run time of the data cleansing and feature engineering processes.
   b) Online layer. The raw streaming data is ingested directly to the "online data preparation service" and the generated features are stored in an in-memory database ("online feature data store") for low latency access. The features are also stored in the "feature data store" for future training.
3. Data segregation. In this step, data is split into training, test, and validation subsets for training, testing, and validating the ML models. For this purpose, the existing labeled data are used. Therefore, this step has only an offline layer and the following subsets of data:
   a) Training subset. This is the actual subset that we use to train the model.
   b) Validation subset. This is a sample subset of the data that is used for an unbiased evaluation of the trained model.
   c) Test subset. This is a sample subset of the data that is used for an unbiased final evaluation of the trained model. The test subset could be also the same as the training subset, although this is not the best approach. This subset should be chosen very carefully. The test subset should be independent of the training subset, although both have the same probability distribution.

   There are different approaches to splitting the dataset into training, validation, and test subsets. For example, we could start with two subsets: training and test subsets. We then keep aside the test subset and split the training dataset into two subsets: the new training subset (e.g., 80%) and the validation subset (the remaining 20%).

4. Model training. In this step, the training data subset from the segregation step is used to train the ML model. This step also only contains an offline layer. The “model training service” receives the training parameters (e.g., model type, relevant features, etc.) from the “configuration service” and will be triggered by events or executed according to schedules. It also requests the training dataset from the “data segregation API.”
5. Candidate model evaluation. In this step, we evaluate the model performance using the validation subset. This step also only has an offline layer. The accuracy of this model is evaluated by comparing the model prediction on the evaluation data subset to true values using metrics parameters. The best model is selected to perform the prediction for the new datasets using the test dataset. The “model evaluation service” requests the evaluation dataset from the “data segregation API.” The models are requested from the “model candidate repository” and the evaluation results are saved back into the repository. The best model is labeled for the deployment.
6. Model deployment. In this step, the best model is deployed to the operational platform for offline (asynchronous) and online (synchronous) predictions. In this step, we can use the containerization approach to encapsulate the prediction model. The model can then be deployed using the continuous delivery implementation approach. In this method, all requirements are packed, evaluated, and deployed into a running container.
   a) Offline. In this layer, the model can be deployed into a container and executed as a microservice. The models can be run on multiple parallel pipelines.
   b) Online. In this layer, the model will be also deployed in a container and then deployed into a service cluster to improve scalability.
7. Model monitoring. In this step, we collect metadata during the model serving time, such as the number of times the model has been served, the predicted results versus the real results, and so on. We then use them for monitoring the model. The “performance monitoring service” is called when a new prediction is served. This service then evaluates the performance of the ML model and stores the results and pushes the relevant notifications.

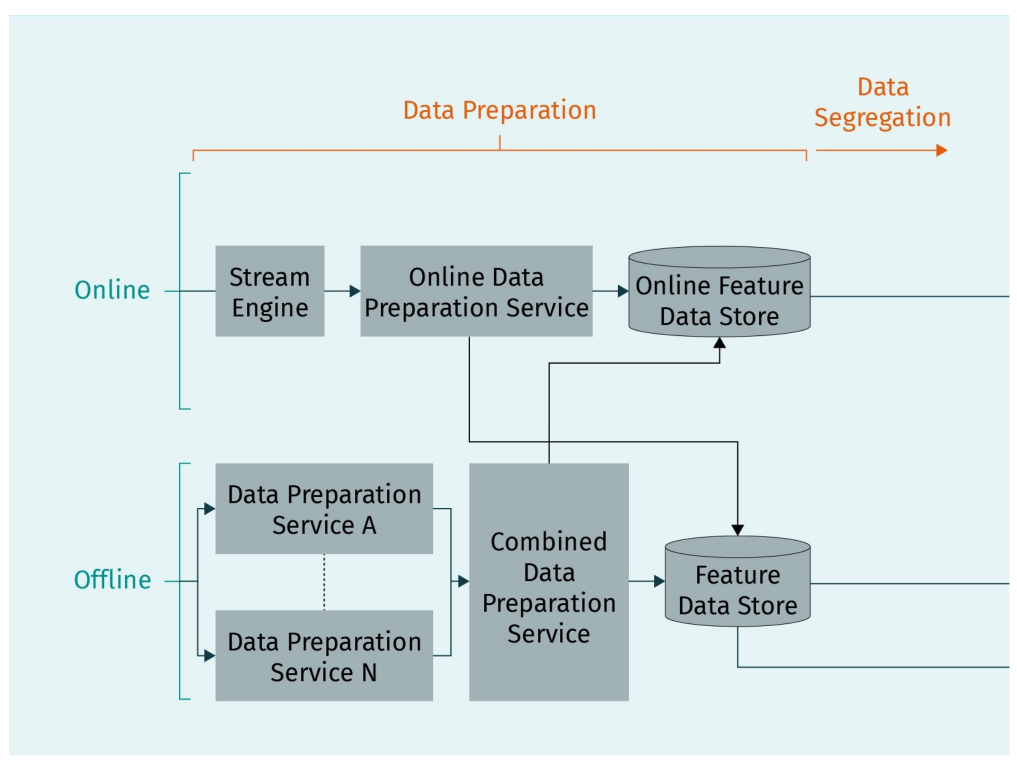
These steps are shown in the following three figures.

Figure 75: Data Ingestion



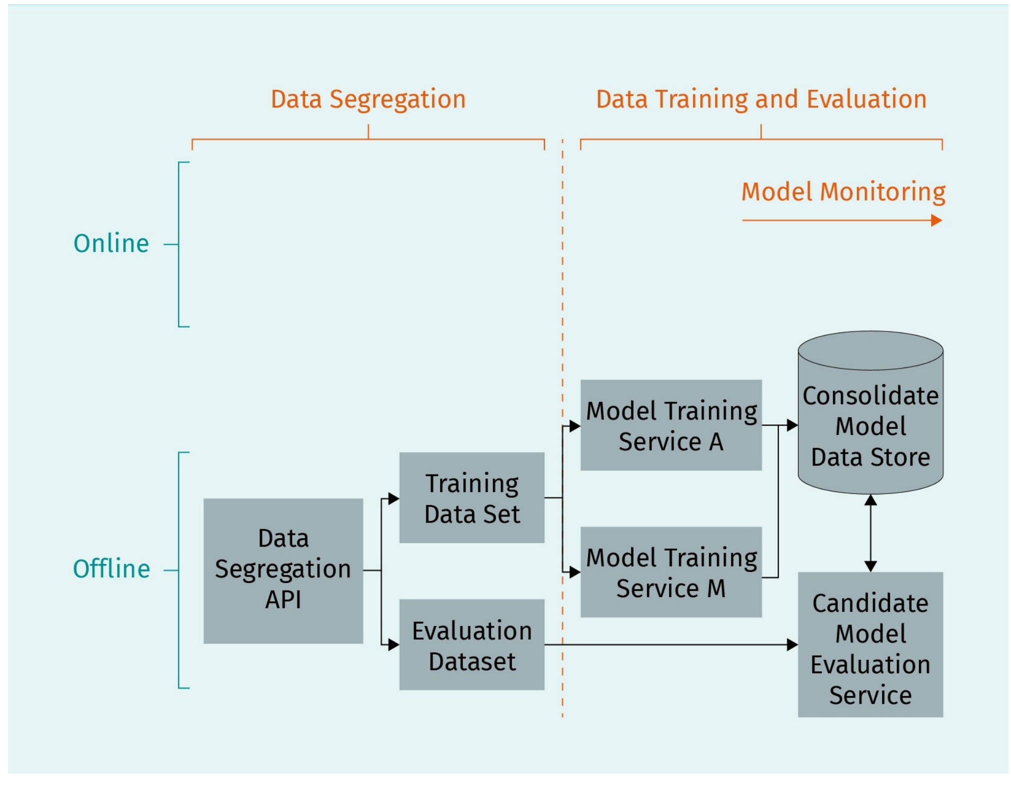
Source: Alavirad, 2020, based on Koen, 2020a.

Figure 76: Data Preparation



Source: Alavirad, 2020, based on Koen, 2020a.

Figure 77: Data Segregation and Data Training and Evaluation



Source: Alavirad, 2020, based on Koen, 2020a.

ML Pipelines at Uber

As a real-world implementation of an ML pipeline, we can refer to Uber's machine learning platform: Michelangelo, a platform for building, deploying, and operating machine learning solutions (Hermann & Del Balso, 2017). For example, the UberEATS application uses Michelangelo to predict a meal's estimated time of delivery (ETD). A typical ETD includes the following time slots:

- the customer’s order confirmation from the restaurant
- meal preparation by the restaurant
- dispatch of an Uber driver to pick up the food once it is ready
- finding a parking spot near the restaurant, walking to the restaurant, picking up the food, walking back to the car
- finding a parking spot near the customer, walking to the customer’s location, and delivering the food

At Uber, data scientists use information from the submitted order (e.g., type of order, time of the day, location, etc.), historical features (e.g., the average time for food preparation in the last seven days), and near-real-time calculated features (e.g., the average time for food preparation in the last hour) to feed the Michelangelo pipeline. Data prediction models are

then deployed to the Michelangelo model's serving containers and are invoked via network requests by UberEATS microservices (Hermann & Del Balso, 2020). The estimated time of delivery will be shown to the customer on their mobile application.

Uber has implemented the following six-step workflow for Michelangelo (Hermann & Del Balso, 2020):

1. Manage data
2. Train models
3. Evaluate models
4. Deploy models
5. Make predictions
6. Monitor predictions

SUMMARY

In this unit, we have learned about operationalizing developed data science models and solutions. We started by defining DevOps, DataOps, and MLOps. In this section, we also briefly reviewed the eighteen principles (manifesto) of DataOps and the main steps of operationalizing a data science plan: build, test, deploy, and monitor.

We then discussed the containerization approach to deploying and integrating data science pipelines into business applications. We first defined the container concept and then introduced Docker as a solution for containerizing applications and Kubernetes as a tool for managing deployed, containerized applications in an operational environment.

Finally, we introduced machine learning pipelines as the independently executable workflow of a complete machine learning task, where its key benefit is automating the machine learning solution life cycle. We also introduced the seven steps of a typical machine learning pipeline: data ingestion, data preparation, data segregation, model training and evaluation, model deployment, and performance monitoring. We then discussed the architecture of the ML pipeline in more detail and, finally, briefly looked at Michelangelo: Uber's machine learning workflow.