POLITECNICO DI TORINO

Computer Engineering

Master's degree thesis

Data Mesh: the newest paradigm shift for a distributed architecture in the data world and its application

Supervisor:
Silvia Anna Chiusano

Author:
Simona Genovese

Academic Year 2020/2021

Contents

List of Figures

Listings

1 Introduction

This chapter presents a first general introduction on the work done, a description of the host company that allowed the study of this analysis during the internship period and an explanation on the methodology adopted for this work.

1.1 About this work

The goal of this project is to expand on the notion of the Data Mesh and to propose a first real implementation of it. In particular, the case study focuses on the backend implementation of the various provisioning services of the data product and its resources, using the Scala programming language and the cloud environments Amazon Web Service (AWS) and Cloudera Data Platform (CDP). Agile Lab S.R.L. is the host company that has allowed the study of this topic and it is one of the very first in the world to approach Data Mesh in a concrete way.

1.2 About the company

Agile Lab S.R.L. is an IT consulting company born in 2014 and specialized in big data, artificial intelligence and machine learning which introduces itself as following:

> «Agile Lab is an Italian company that aims to realize software systems using the most advanced Big Data technologies for all the companies which want to generate value from their data even if that data are “big” or “small”. We operate in different business sectors but always with the same common approach: bringing innovation and driving forward our clients through technology, competence, positive and “agile” thinking. In the last 5 years, Agile Lab has grown up to be a 50+ people company located in five different offices.»

Agile Lab collaborates on projects in the banking, insurance, industrial, and utilities sectors, among others. Agile Lab provides its own *WASP* platform (Wide analytics streaming platform), an asset based on open source technology and usable in *as-a-service* mode, at the foundation and in support of the realization of projects that require the ability to manage large amounts of data to solve the typical complexities of mission critical use cases, ensuring greater accuracy in the service delivery phase and reduced resource consumption. WASP is a data streaming platform (also used by large companies) that focuses on real-time data processing and machine learning. The goal of Agile Lab's WASP project is to have data processed by decentralized and intelligent systems that can give contextual ideas without waiting, so that processes are not disrupted. The issue with many AI and machine learning platforms is the latency that occurs in decision-making processes (due to the requirement to get answers from systems). WASP's continuous intelligence, on the other hand, works in real time on the data stream, simplifying operations and shortening product time to market.

The *Data Mesh* is one of the new technological paradigms leveraged by Agile Lab in this context, and it represents one of the new technological paradigms, which simplifies the transition to a *data-driven* culture and which records rapid growth in 2020, allowing for faster work. The data mesh is a distributed and decentralized architecture that enables corporate agility and scalability, reduces time-to-market, lowers maintenance costs, and allows for a fair and transparent internal cost distribution. The data mesh goes beyond the traditional approach that sees a central data lake that acts as a collector of all information. In brief, the data is dispersed among a number of more agile structures, ensuring both the architecture's scalability and real-time reactions. Data Mesh usage also provides for lower maintenance expenses and more transparent cost management.

1.2.1 Agile Lab Handbook

The Agile Lab Handbook is a document that describes all useful information for the company and its employees. It was produced by the employees themselves and is updated as needed to reflect any improvements or changes. It is the central repository that allows to

indirectly communicate corporate behaviors, structures and operations, how to perform tasks based on the role and general information in an easy and efficient way, as:

- it is possible to consult it at any time by searching through the various sections and/or using a keyword;
- reading is an asynchronous operation, therefore when someone needs to obtain information they can look for it in the handbook without disturbing anyone;
- reading is faster listening.

Values

A peculiarity of the handbook is the underlining of corporate values (not taken for granted), which does not leave the individual to interpret what is right and wrong within a company context, but rather describes the behaviors to be adopted in order to work peacefully within the company and team. Some examples of the values described in the handbook are: try to help someone else when possible, even if they are not part of your team; openly express gratitude for assistance received; keep negative feedback in a one to one video call (or in the smallest possible setting) to avoid putting your colleagues in a bad light; no ego, try to comprehend various points of view rather than concentrating just on your own... The outcomes of these simple recommendations are many, including a shift in mentality and an overall organization optimization.

Organization

Agile Lab is organized into several "roles" and "workgroups", which are referred to as "circles". Each circle represents a collection of goals, domains and responsibilities. A circle can be subdivided into several sub-circles and roles. Each circle has a leader who is in charge of organizing the circle itself independently. Each individual can have several roles allocated to them, and each position can be filled by multiple persons. A role has clearly defined responsibilities and objectives. Roles will be established and deleted based on the company's current needs in order to achieve the greatest results, generating organizational flexibility. Here the list of Circles in a hierarchical view:

- General Company
  - HR
  - Finance
  - PeopleOps
  - Sales
  - Engine
    - Products
      - Aim2
    - SRE
    - Internal IT
    - Software Factory
    - Architecture
  - AgileSkill
  - Marketing

1.3 Agile methodology

Agile Lab, as the company name also shows, works following the *agile* methodology. In software engineering, the term refers to a collection of software development methodologies that first appeared in the early 2000s and are based on a set of common principles that divide a project into stages. Agile techniques are in opposition to the waterfall model (one release, at the end) and other conventional development models, offering a less organized approach with the goal of providing functioning and quality software to the client fast and frequently (early delivery/frequent delivery). A difference between the two approaches waterfall and agile is shown in Figure 1.1 and Figure 1.2.

Most agile approaches try to decrease the chance of failure by building software in short periods of time called *iterations*, which usually last a few weeks. Each iteration is its own little project, which requires constant collaboration with stakeholders and must

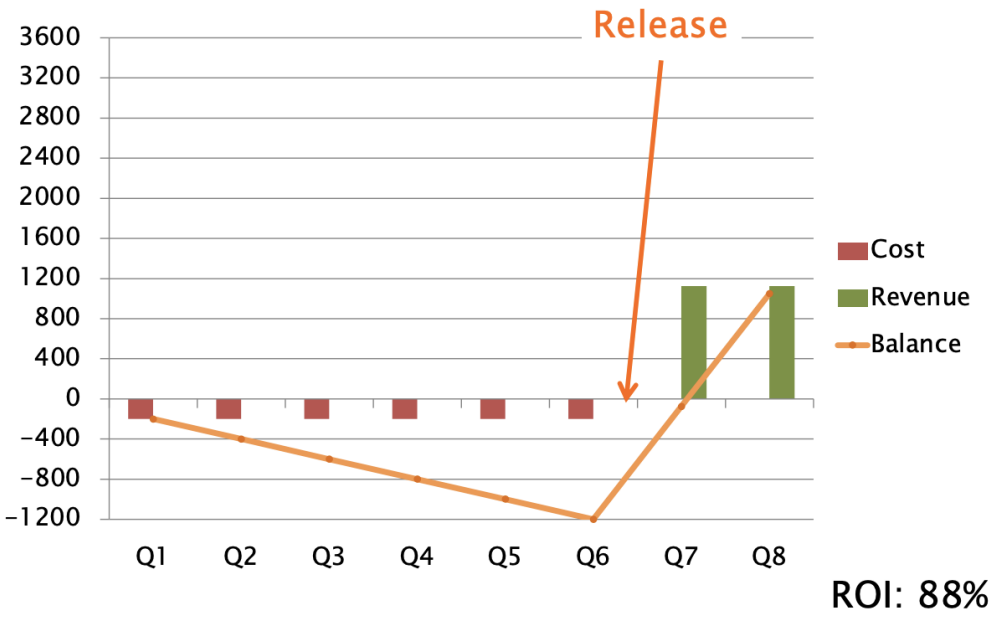

Figure 1.1: Waterfall model

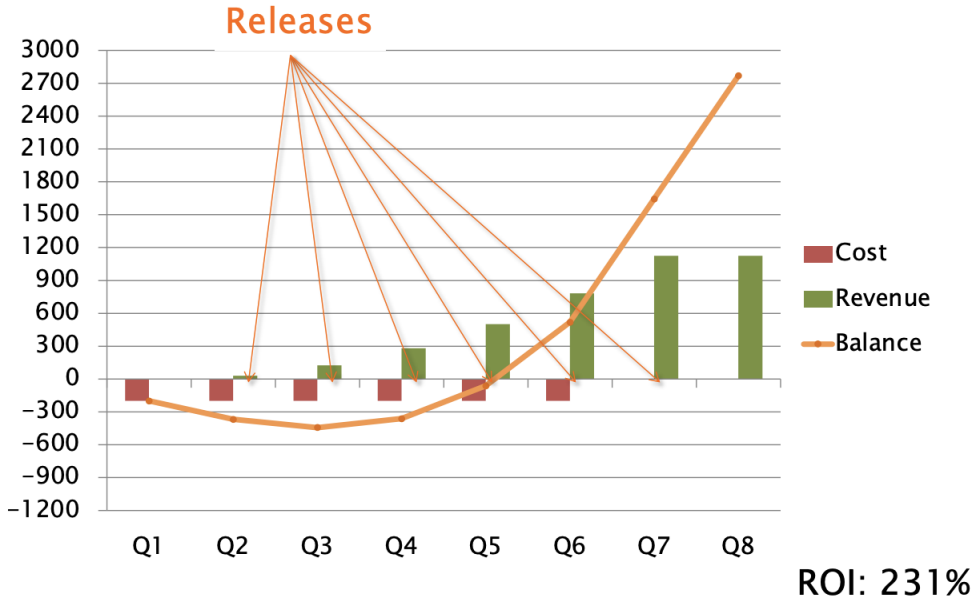

Figure 1.2: Agile model

contain everything that is needed to deliver a small improvement in software functionality: planning, requirements analysis, design, implementation, testing, and documentation. Even if the output of each iteration does not contain enough functionality to be considered complete, it must be released, and it must come closer and closer to the customer's demands with each iteration. The team must re-evaluate the project priorities at the conclusion of each iteration.

1.3.1 Scrum

Jeff Sutherland defines the agile *Scrum* methodology in *La Guida a Scrum* as follows:

> «Scrum is a process framework used since the early 1990s to manage the development of complex products. Scrum is not a process or a technique for building products but rather it is a framework within which various processes and techniques can be used. Scrum makes clear the relative effectiveness of your product management and development practices so you can improve them.»

Therefore, Scrum is a project management methodology that divides the project into short blocks of work (*Sprints*), each of which results in a software increment. It explains how to specify the specifics of the work to be done in the near future and sets up a series of meetings with specific features to allow for inspection and control of the work performed.

Scrum is founded on empiricism, which is an empirical control theory of instrumental and functional process analysis. Knowledge comes through experience, and decisions are made based on what is known, according to empiricism. Scrum optimizes predictability and risk control with an interactive technique and an incremental approach.

The Scrum framework consists of principles, events, artifacts and the Scrum Team and associated roles. The scrum process is shown in Figure 1.3.

Sprint

Sprint is the basic iteration in a Scrum approach. It is a process that lasts from 1 to 4 weeks (of fixed duration) in which a piece of working software is produced to be demonstrated

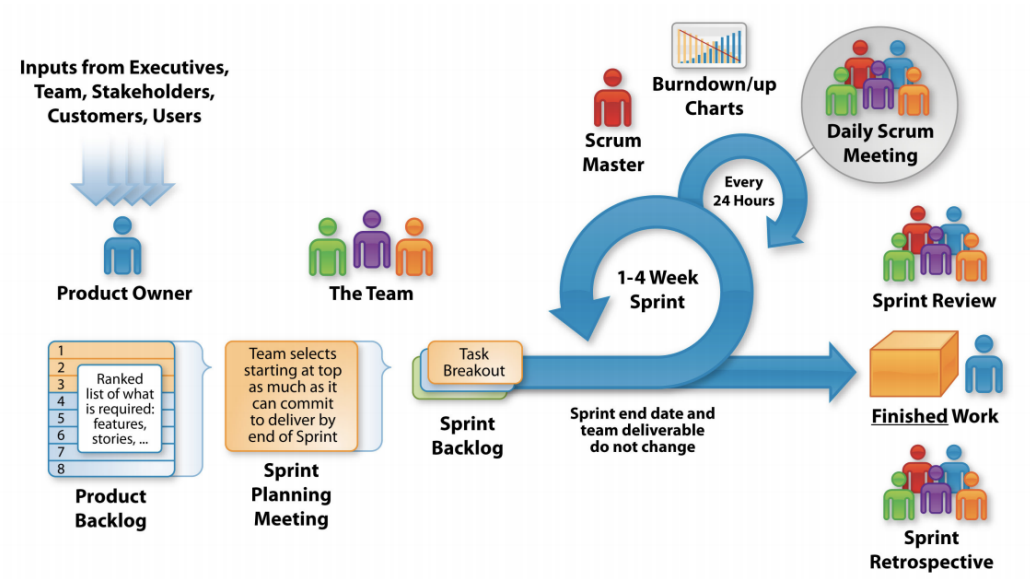

Figure 1.3: Scrum process in a nutshell

and reviewed at the end of the sprint.

The sprint begins with a *sprint planning*, in which goals are identified and stories are estimated. The objectives are selected from the *product backlog*, which represents the complete and ordered list of requirements (created and sorted by the Product Owner), and the items selected and included in the sprint (i.e. the items with the highest priority) form the *sprint backlog*. The team members determine, through estimates, how many items they will be able to complete during the sprint. A sprint is considered *immutable*, in the sense that once the objectives have been defined they cannot be changed until the next sprint planning.

During the sprint, the team meets daily for quick updates on the portion of the completed product (*daily stand-up*).

The sprint ends with a *sprint review*, comparable to a demo of the working piece of software to get feedback from stakeholders on the work done. At the end of the review, the members of the development team perform a *sprint retrospective*, trying to define what went wrong, what could be improved and what should remain unchanged for the

next sprint.

Principles

Any implementation of empirical process control is supported by three pillars:

- *Transparency*: process visible to whom is responsible. It requires that those characteristics be specified by a uniform standard so that spectators may comprehend what they are seeing. For example, all participants must agree on a single language of reference for the process, or everyone who does the work and must accept it must agree on a common definition of the word "*done*";
- *Inspection*: on artifacts and goal to detect variances. Scrum practitioners must review the artifacts created and the progress made toward the achievement of the established objectives, discovering any inconsistencies with respect to what is intended to be achieved. The frequency of inspections must not be such as to cause an interruption of the work in progress;
- *Adaptation*: to achieve the objectives. If the inspector determines that one or more parts of the manufacturing process are outside of permissible limits, and the final product cannot be approved, he must interfere either in the manufacturing process or in the material generated by the manufacturing process. To avoid additional divergence from the specified objectives, the intervention must be performed as fast as feasible.

Events

Scrum specifies four formal events:

- *Sprint Planning*: during this meeting, the work to be done in the Sprint is planned. This strategy is the result of the Scrum Team's joint efforts. In the Sprint planning, the work to be done is selected based on the estimates and collaborate with the entire team to create the Sprint Backlog, which outlines the time required to complete the task;

- *Daily Scrum* or *Daily Stand-up*: daily communication meeting of the project team. The meeting is held every day at the same time and lasts no more than 15 minutes. It is important to keep the other team members up to speed on the work that has been completed and/or issues that have been encountered without going too far (stand-up as you participate by remaining standing, so as not to give way for participants to get distracted and isolate themselves as happens in "traditional" meetings);

- *Sprint Review*: meeting needed to examine the increment and, if required, adjust the Product Backlog. The Development Team and stakeholders collaborate on what was accomplished throughout the Sprint during the Sprint Review meeting;

- *Sprint Retrospective*: it is an chance for the Scrum Team to inspect itself and create an improvement plan that will be implemented in the following Sprint. The Sprint Retrospective's goal is to look back on how the previous Sprint went in terms of people, relationships, processes, and tools, identify and classify the key components that went well as well as the areas where you can improve and create a strategy for improving the Scrum Team's productivity.

Artifacts

Scrum's artifacts are especially intended to increase the openness of critical information required for Scrum Teams to produce a "Done" Increment:

- *Product Backlog*: an ordered list of a product's "requirements." It comprises the Product Backlog Items (PBI) that the Product Owner prioritizes based on factors such as risk, business value, and deadlines. The "story" structure is often used for adding features to the backlog. The product backlog is a list of "what" needs to be done, structured in the order in which it has to be done.

- *Sprint Backlog*: the list of tasks that the development team must do during the next sprint, which is produced by picking a number of stories/features from the top of the product backlog.

Roles

The Product Owner, Development Team, and Scrum Master make up the Scrum Team.

The *Product Owner* represents the stakeholders and is the customer's voice. He is in charge of ensuring that the team adds value to the company. The Product Owner prioritizes and adds items (product requirements) focused on customer demands (usually *user stories*) to the product backlog.

The *Development Team* is in charge of providing a product that is possibly releasable at the conclusion of each Sprint, with feature increments.

The *Scrum Master* is in charge of removing roadblocks that prevent the team from meeting the Sprint objective and delivering the required deliverables.

2 | Big Data

An introduction to Big Data

This chapter introduces the concept of Big Data, what it is and why it is so important. Big data represents the core of this analysis, as the Data Mesh (presented in the chapter 3) is born thanks to the increase in data, more and more "big". This chapter also shows the logical components of a Big Data architecture and ends with the presentation of the technological choices for it.

2.1 What is Big Data

To explain what Big Data is in everyday life, consider social media interactions, website visits, or networked cellphones. All of this creates an enormous quantity of data compared to a few decades ago. Big Data is defined as large amounts of heterogeneous data from many sources and formats that can be examined in real time.

The term "big data" refers to computer data that is extremely vast, rapid, or complicated, making standard processing difficult or impossible. Large volumes of data may be accessed and stored for analysis, and this has been done for a long time. But it was not until the early 2000s that the notion of big data gained traction, when industry researcher Doug Laney articulated the current definition of big data as *the three Vs*:

- *Volume*: organizations collect data from multiple sources, including business transactions, intelligent devices (IoT), industrial equipment, video, social media, and more. In the past, storage costs would have been a problem but today it is much more accessible, thanks to platforms such as data lakes and Hadoop;
- *Velocity*: as the Internet of Things expands, data flows to companies must be managed in a fast and unprecedented manner. RFID tags, sensors, and smart meters have necessitated the management of these data rivers in near real time;

- *Variety*: data comes in a wide range of formats, including organized and numeric data in traditional databases, unstructured text documents, email, video, audio, stock data, and financial transactions.

A basic and succinct approach for defining new data, as a result of increased information sources and, more broadly, technological progress. The variables of Veracity and Variability have now been added to the Laney paradigm, resulting in the 5V of Big Data:

- *Veracity*: "Bad data is worse than no data," some industry insiders used to remark. The data must be accurate and dependable. This problem is exacerbated by Big Data: data management technology evolve, and the rate at which data is produced and sources expand. Quality and integrity of data, on the other hand, remain critical pillars for producing meaningful and trustworthy analyses; *Variability*: much more data, in different formats and from different contexts. When interpreting statistics, it's important to keep in mind that their significance might vary with time. Even more so if the user is a line of business employee rather than a data scientist.

2.2 Why Big Data is important

The value of big data is determined by its application rather than its quantity. For example, data from any source may be accessed, analyzed, and solutions found that help companies to:

- decrease costs,
- shorten deadlines,
- develop new goods and optimize offers,
- make better smart decisions.

With the combination of big data and analytics, in fact, business results can be obtained such as:

- in near real time, determine the reasons of failures, issues, and flaws;

- create coupons at the point of sale depending on the purchasing habits of consumers;
- recalculate the risks of your whole portfolio in minutes;
- detect fraudulent activity before it has a chance to affect the firm.

2.3 Big data architecture

The insertion, processing, and analysis of data that is too massive or complicated for standard database systems; a big data architecture is designed to handle these operations. Companies utilize several criteria to evaluate whether it is better to use Big Data solutions, based on the capabilities of the users and the tools available. The term "big data" is increasingly being used to refer to the value that can be extracted from datasets using advanced analytics, rather than the big volume of data, which in these cases they tend to be very large.

The logical and physical framework that governs how large amounts of data are ingested, processed, stored, managed, and accessible is referred to as big data architecture. The logical components of a Big Data architecture are shown in Figure 2.1.

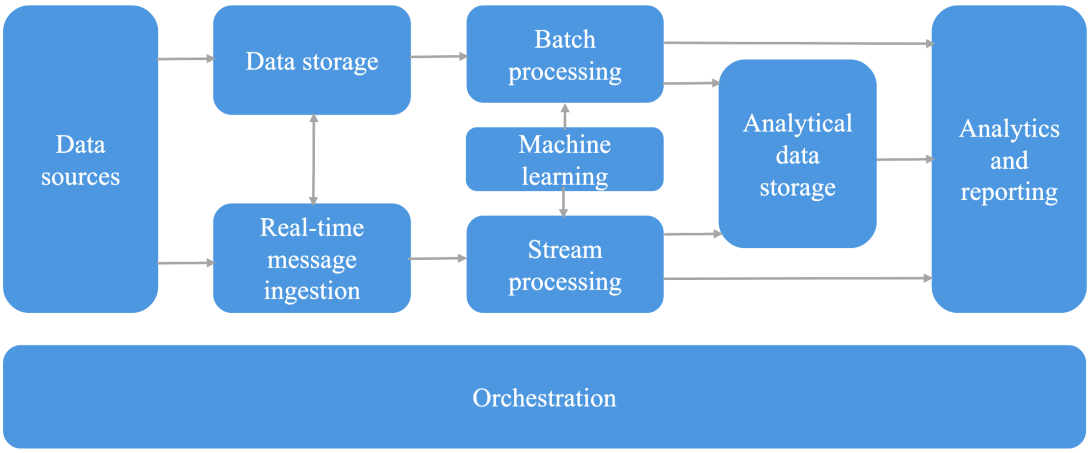

Figure 2.1: Big data architecture

The following components are found in most big data architectures:

- *Data sources*: one or more data sources are the starting point for all Big Data solutions (from data stores of applications such as, for example, relational databases, to static files generated by applications, i.e. log files, to real-time data sources, such as IoT devices);

- *Data storage*: batch processing data is generally stored in a distributed file archive, which can hold enormous quantities of big files in a variety of formats. A data lake is a term used to describe this sort of repository;

- *Batch processing*: because huge datasets need long-running batch processes to filter, combine, and generally prepare data for analysis, a big data solution must often process data files utilizing long-running batch processes;

- *Inserting messages in real time*: if real-time sources are used, the architecture must allow for the capture and storage of messages in real time for flow processing. This may be a simple data store where incoming messages are placed in a folder to be processed;

- *Stream processing*: following the real-time capture of messages, the solution must process the data by filtering, aggregating, and otherwise preparing the data for analysis;

- *Storage of analytical data*: many big data solutions prepare data for analysis before serving it in a structured manner that can be queried using analytical tools. A relational data warehouse may be utilized as an analytic data storage to answer these questions. The data can also be displayed using a low-latency NoSQL technology such as HBase or an interactive Hive database, which abstracts metadata about the data files in the distributed data storage;

- *Analysis and reporting*: the majority of big data solutions aim to deliver data insights via analytics and reporting tools. A data modeling layer, such as a multidimensional OLAP cube, might be included in the design to allow users to analyze the data;

- *Orchestration*: most big data solutions consist of repetitive data processing operations, encapsulated in workflows, that convert source data, transfer data between various sources and sinks, load processed data into an analytical data store, or put data straight into a report or dashboard.

2.4 The technological choices for Big Data

Aside from the abilities required for Data Science management, the *technological infrastructure* is another factor that allows Big Data to be improved. To mention an example of the current trends, the most innovative technologies allow the development of advanced analysis (supported by Machine Learning algorithms), real-time data ingestion and analysis, and the integration of more diverse forms of data.

The main technological options in the different *Data management* phases span from defining the Data Strategy to Data Management, without forgetting the Data Integration and Data Visualization phases:

- *Data Strategy*: is the "art" of enhancing data, or creating a long-term action plan aimed at achieving a specific goal, i.e. internal and/or external data valorisation. A strong Data Strategy that is aligned with corporate objectives and capable of managing Big Data is structured from Data Governance to Advanced Analytics, via direct data monetization and data management through technical and organizational rethinks.

- *Data Management*: it allows to manage increasingly complex data. It is estimated that Data Scientists spend 80% of their time preparing and cleansing data. According to the three Vs definition (Volume, Velocity, Variety), as already stated, the more complicated the data is, the more time must be spent on preliminary analysis tasks. Furthermore, the regulations demand that data processing be given more attention. As a result, data management, data governance, and data preparation tools are always evolving.

- *Data Integration*: from the Data Warehouse to the Data Lake. Nowadays, many

companies manage data with traditional solutions such as silos or data warehouse, but the more forward-thinking companies are embracing more modern data integration technologies. Data Lake is one of them. The Data Lake should be flanked by a Data Warehouse in an ideal scenario in order to build an Integrated Model that satisfies the many demands of data analysis: all of this means extracting the true value of Big Data.

- *Data Visualization*: it means talk about big data. The need to tell more or less complex phenomena through forms, graphics and infographics finds an answer in Data Visualization. More advanced visualization tools have become required in organizations as a result of the introduction of Big Data: instead of the traditional report, we are focusing on dashboard concepts to satisfy the demand of users to depict complex and dynamic phenomena in real time and with complete autonomy.

3 | Data Mesh

The paradigm shift towards the next generation data architecture

This chapter introduces the concept of Data Mesh, starting from the data models present on the market today, such as the Data Warehouse and the Data Lake. The problems of these two models are analyzed in order to introduce the Data Mesh, which presents a solution in the data world. Hence, this new model with its principles and logical architectures is discussed.

3.1 Introduction to Data Mesh

The concept of Data Mesh may be briefly explained as a decentralized architecture, in which each unit of the architecture is a domain-driven data ownership that is treated as a product. The complexity is shifted into infrastructure layer allowing a self-serve infrastructure.

3.2 How and why Data Mesh

The data mesh is largely a management model. Companies that have adopted data lake-based systems as an enabling technology over the years have never actually worked to create an organizational process suitable for creating value on the data, which has always remained the same in practice, namely the one used for data warehousing most recently twenty years ago.

The truth is that the context changes. Continued adoption of a centralized paradigm is a mechanism that will not work for organizationally complex organizations in the future, because 9 times out of 10, people who begin working on the data, generally a centralized

IT team of data engineers, have no prior knowledge of the data. This is the real problem, which in turn generates many others that have a significant impact on the business, but it is a normal process. The data mesh provides an organizational and, as a result, architectural solution to this challenge. In addition, the amount of money spent on data has increased significantly, but so has the data itself: in this data-driven culture, companies are treating data as a business resource, working hard for analyzing them. Recent decades' systems are starting to raise some red flags: something is not working, something is causing failure... In order to analyze what is going wrong, it is good practice to look back into the past: *"Everything starts with a story"*, says Joseph Campbell, and the tale behind the Data Mesh is about the models that are still in use today, such as the *Data Warehouse* and the *Data Lake*.

3.2.1 Latest generations

When it comes to data, the challenge starts with two isolated fields (Figure 3.1):

- *Operational data*: operations on databases for adding or modifying one data unit at a time, in order to reflect the last transaction's current value.
- *Analytical data*: read-only system that stores historical data in order to train machine learning models, improve the user experience and make business decisions. The architectures of the latest generations (data warehouse and data lake) are part of this category.

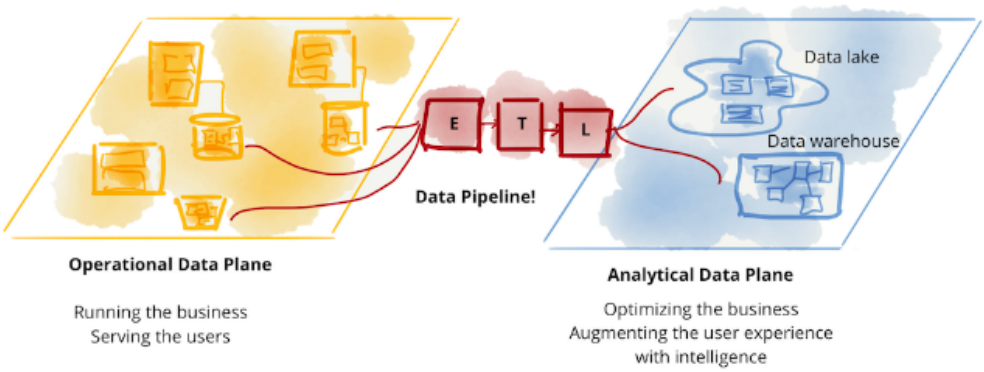

Figure 3.1: Latest generations architecture

These two worlds are divided from a huge amount of ETL (Extract, Transform, Load) pipelines that keep failing as the data grows, due to the fact that data travels from one end of the network to the other (Figure 3.1). Somehow, this process is *breaking down*.

Data warehouse

The concept of *data warehouse* (DWH) was born in the 1980s. A data warehouse is a storage system that contains structured and filtered data, which are extracted from different sources within the operational system and, through ETL transformations, are processed for analytical purposes in the field of *On-Line Analytical Processing* (OLAP). In other words, it acts as a central collecting point and provides analysts with a full picture of heterogeneous data (Figure 3.2). The data must be *structured* in order to be managed by the DWH, which means it must be represented by relationships that can be expressed using hard tables and diagrams.

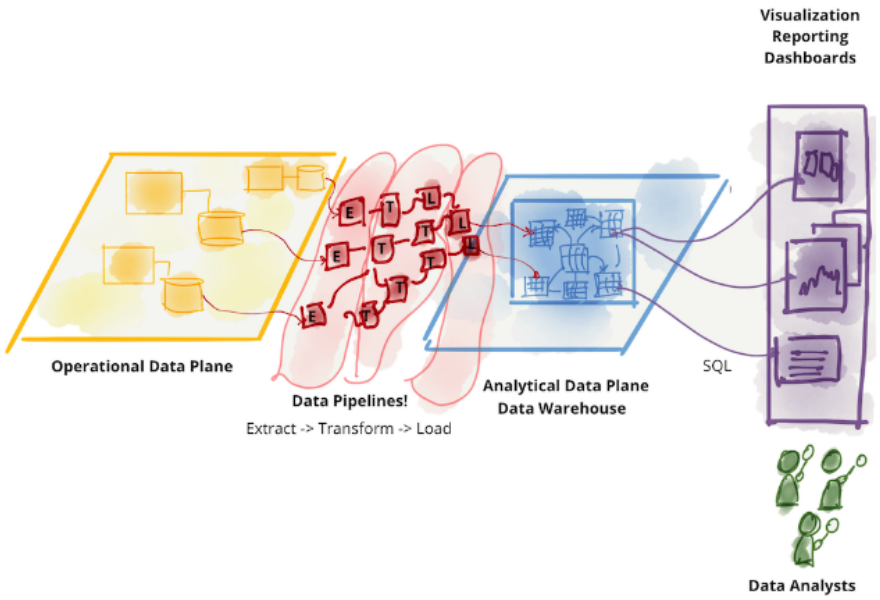

Figure 3.2: Data warehouse architecture

Data lake

The evolution of the data warehouse emerged in 2010s, called *data lake*. A data lake is a central storage repository (lake) that holds big data taken from many sources, stored in their a raw format and modeled downstream, rather than being converted a priori for a universally defined canonical model as was in data warehouse model. It is feasible to integrate vast volumes of data in any format and from any source using this management mode. Of course, the stack of technologies has expanded even further in order to support, for instance, the storage of huge semi-structured blobs, as well as the orchestration of pipeline management to carry out these tasks and modify the data (architecture shown in Figure 3.3). A *cloud-based data lake* provides services in a easier way, but the paradigm is essentially the same.

3.2.2 Assumptions to challenge

"We have been busy innovating and building technologies, so then why the failure modes we are seeing at scale?" Dehghani asked. In order to see what it can be changed, the assumptions to challenge are:

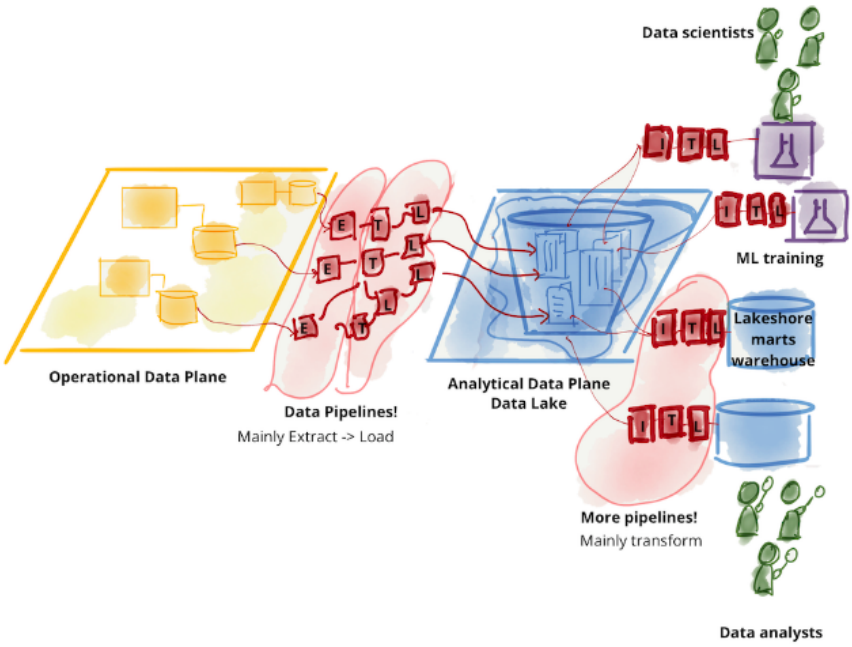

Figure 3.3: Data lake architecture

1. *Monolithic and centralization of data management system*: a centralized system is a bottleneck between data sources and data consumers, so there are points of international friction as you scale. Centralizing data also means losing the link between data and their source.

2. *Hyper-specialized teams*: understanding how data is related, where it is created, and how it is really used is the challenge and hyper-specialized data engineers are needed for this. To win the race, every company is scrambling to find the correct amount of data engineers to serve an ever-increasing number of users and data sources – this system is full of friction.

3. *Architecture decomposition orthogonal to change*: architects and data scientists are constantly splitting the task into pipelines for ingestion, transformation, and service in order to attain a particular degree of operational scale.

4. *Disconnected execution*: the way these platforms are built with this centralized silo is disconnected from those who use the data and from those who provide the data. The result is that all the people on the platform are unhappy.

3.3 Data Mesh objective, principles and logical architecture

Data mesh is the newest frontier that aims to destroy the data lake and data warehouse concepts. These assumptions have been accepted for years, and now the data mesh paradigm shift allows for the management of data at scale, exploring how architecture and ownership may be partitioned differently, as well as what the platform and domain's roles are. The principles of data mesh are:

1. *Domain-driven, distributed architecture and ownership*: how can a decentralized system be achieved? How to decompose data around domains?
2. *Data as product*: how to treat data as an asset?
3. *Self-serve infrastructure as a platform*: how to hide the complexity of the infrastructure that runs and operates on big data?
4. *Federated computational governance*: how to have a harmonious system and a well-structured ecosystem?

The high level design of the data mesh architecture is shown in Figure 3.4.

3.3.1 Domain-driven, distributed architecture and ownership

The first pillar of the data mesh architecture introduces the idea of breaking the monolithic system into pieces: instead of taking data and storing them into a centrally owned data lake or platform, data mesh decomposes the architecture through the concept of *domain*, where domains need to host and serve their domain datasets in an easily consumable way. The data domain is established at the source (which rarely changes). In this way, the source is natively oriented to the domain and the data remains connected to it, without losing the data ownership (as it happened in past generations). Furthermore, the domains are also close to the point of destination: the data is aggregated according to the business and,

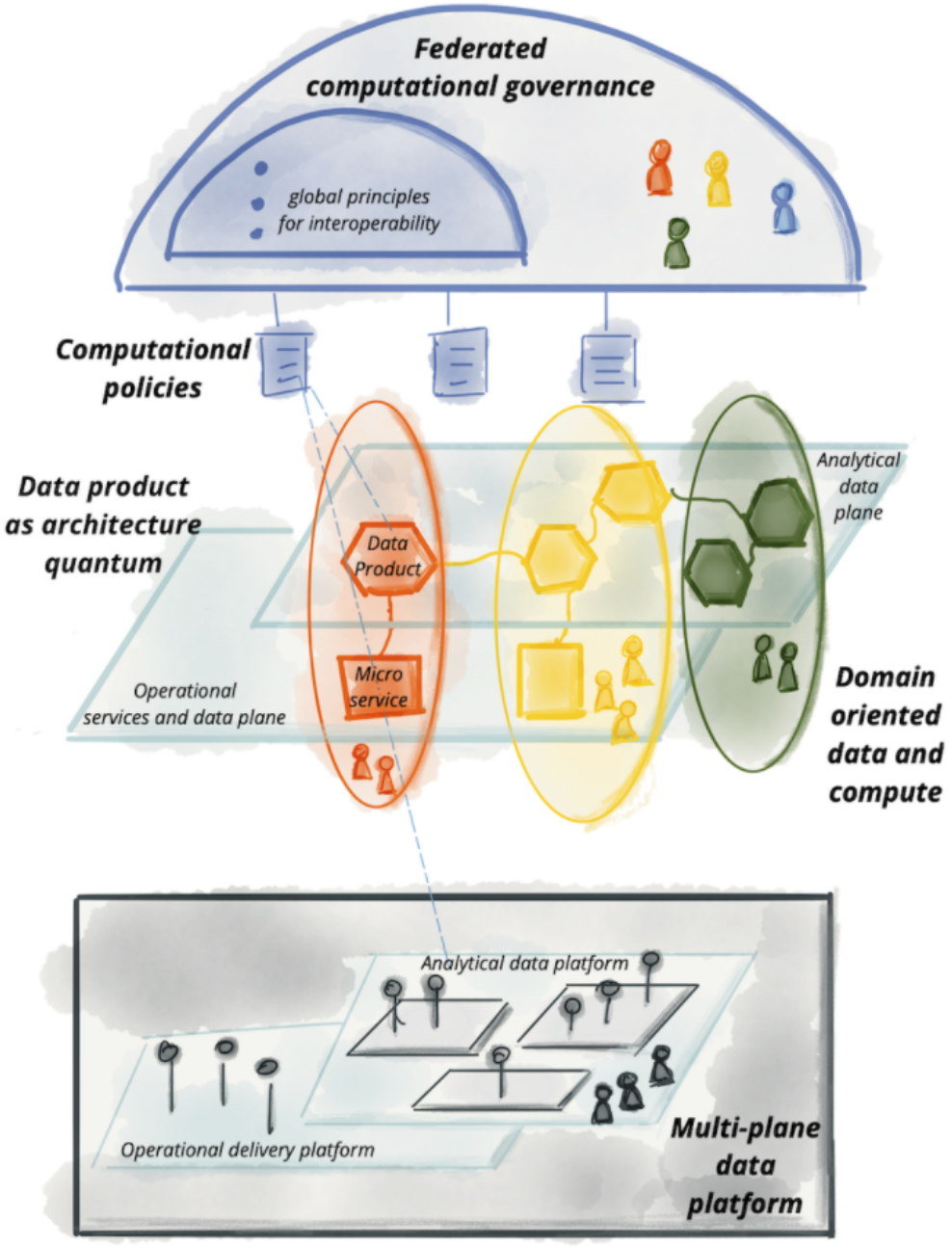

Figure 3.4: High level design of data mesh architecture

therefore, the data domains are also consumer oriented. This whole new system works due to the fact that the data is managed directly by their owners, by the people who know the data, by the same people who generate that data or who will use it; then it makes sense for these people close to the data to directly assign it to the correct domain. In essence, analytical data aligns with domains, domains align with the source and, sometime, domains are aligned with consumption.

Logical architecture: domain-oriented data and compute

In order to facilitate this breakdown, an architecture that organizes analytical data by domains must be modelled. Therefore, a domain, which is independent of other domains, provides operational APIs as well as an analytical data endpoint. To allow the domains to serve their own analytical data, the design must remove any friction or connection. Clearly, there may be dependencies across domains, such as when an output port from one domain simultaneously serves as an input port for another.

3.3.2 Data as product: data product thinking

Data mesh perceives data *as* a product rather than data *by* product, and each node in the mesh is referred to as a "data product." As a result, because data is a valuable asset, a successful product must please its consumers, data analysts and data scientists (Figure 3.5). To do so, data sets must be shareable/discoverable, easily consumable, understandable, self-descriptive with adequate documentation, addressable (once identified, you must know where to direct them), trustworthy to use, inter operable, secure, and so on.

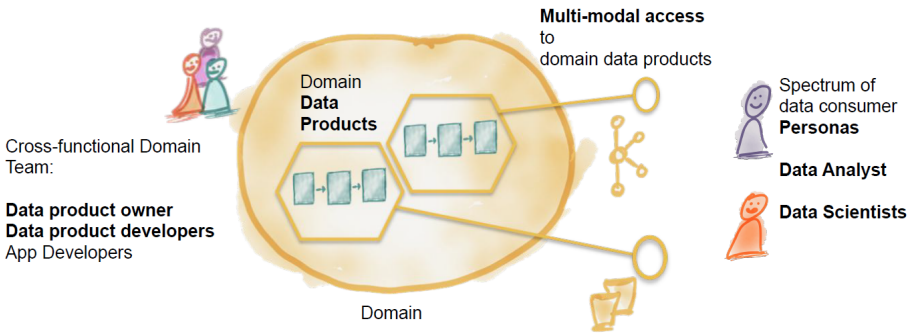

Figure 3.5: Characteristics of a domain data product

Logical architecture: data product as architecture of quantum

The data product is the key concept of the data mesh. Microservices are the architectural quantum of services, the smallest unit of service architecture that can change independently and deliver value independently, just as the data product is the smallest unit of a mesh. "Architectural quantum", as defined by Evolutionary Architecture, "is the smallest unit of architecture that can be independently deployed with high functional cohesion and includes all the structural elements required for its function". For the data product these elements, according to Dehaghani, are three:

- *Code*: the code accepts data from upstream via polyglot input data ports and provides code and interfaces for providing output data via polyglot output data ports. These input and output ports that define a data product's design can be distributed independently and have their own life cycle. The code defines:
  - how to consume, transform and serve upstream data via pipelines;
  - how to access to data via APIs, what is the semantic and syntax schema, what are the metrics for observing and analyzing data in order to have an idea about the data quality or the customer experience and other metadata;
  - the enforcement traits, such as the ACLs (Access Control Lists) policies, compliance, provenance, etc.;
- *Data and metadata*: data can be served as events, batch files, relational tables,

graphs and other formats depending on the nature of the domain data but maintaining the same semantic. Therefore, the analytical and historical data and metadata are in a polyglot form. As previously stated, the data must be usable, and in order to do so, there is a set of metadata (intrinsic to the data, such as its semantic definition) that includes data computational documentation, semantic and syntax declarations, quality metrics, and so on; also, metadata communicates the traits used by computational governance to implement the expected behavior, for instance, through the access control lists;

- *Infrastructure*: the infrastructure component enables the data product's code to be built, deployed, and executed, as well as storage and access to large data and metadata.

3.3.3 Self-serve infrastructure as a platform: enable autonomy

Enable autonomy using the platform thinking concept: in order to provision the infrastructure in each domain are requires specialized data engineers, but this would be challenging. Only by having access to a high-level abstraction of infrastructure that it could be possible eliminates the complexity and friction of provisioning and managing data products. In this way, teams can autonomously control their data products (Figure 3.6). This actively demonstrates that at the highest level of abstraction it is not convenient to have engineers who are actually just data engineers, nevertheless, any normal software engineer capable of creating these data products is sufficient. Furthermore, engineers should not be concerned with infrastructure configuration; instead, they should focus on determining which are the input ports, output ports, what type of access policy they need to design, and who may access which ports.

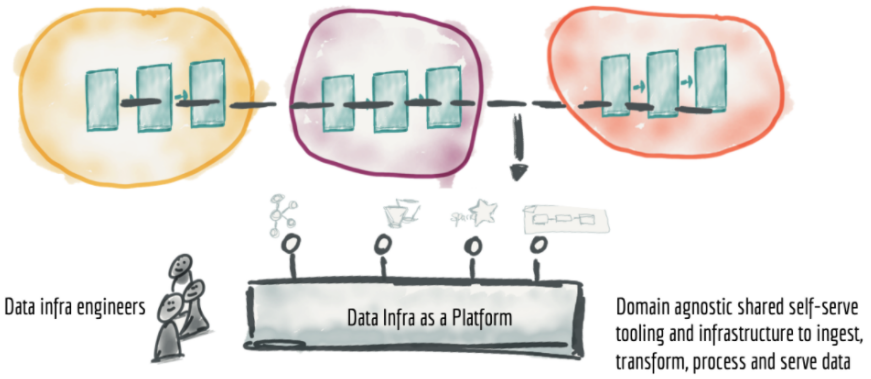

Figure 3.6: Extraction and collection of domain-independent data pipeline infrastructure and tools into a separate data infrastructure as a platform

Logical architecture: a multi-plane data platform

Through self-serve interfaces, a platform plane[1] (Figure 3.7) enables a variety of related features. Dehaghani suggests three different examples of planes:

- Data infrastructure plane: it helps with the provisioning of the infrastructure needed to execute the components of a data product and the mesh of products.
- Data product developer experience plane: a typical data product developer's main interface. It manages the lifecycle of a data product using simple declarative interfaces.
- Data mesh supervision plane: a set of global capabilities best offered at the mesh level - a graph of connected data products. For example, searching or exploring the mesh of data products is the best way to find data products for a certain use case.

[1] A plane is representative of a level of existence.

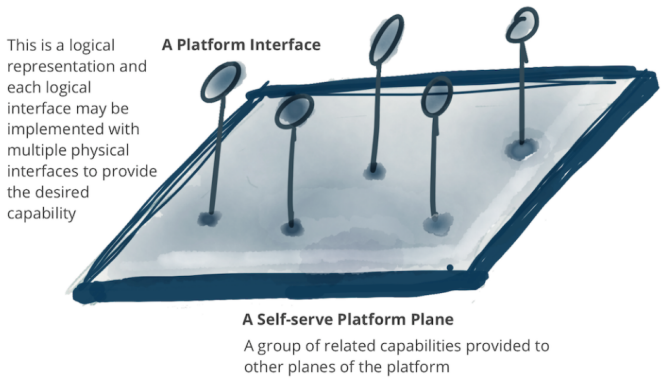

Figure 3.7: A platform plane that provides capabilities through self-serve infrastructure

3.3.4 Federated computational governance: build an ecosystem

The last (but not least) pillar that challenges the data mesh paradigm is the creation of a federated and global governance that allows for interoperability and ecosystem thinking. Companies are now battling to achieve a canonical universal model for data and, in a distributed and autonomous environment, a federated governance model that is based on standardization and interoperability needs to be defined. Owners of data products and owners of data platforms make local and independent decisions while adhering to global rules to ensure a healthy and interoperable ecosystem. In conclusion, the ecosystem governance system should be able to lead to standardization in terms of how data is defined, discovered, and interoperable.

Logical architecture: computational policies embedded in the mesh

A supporting organizational structure, incentive model, and architecture are required for a federated governance model to achieve the aim of taking global choices and standards for interoperability, with global policies, while preserving the autonomy of local domains (Figure 3.8).

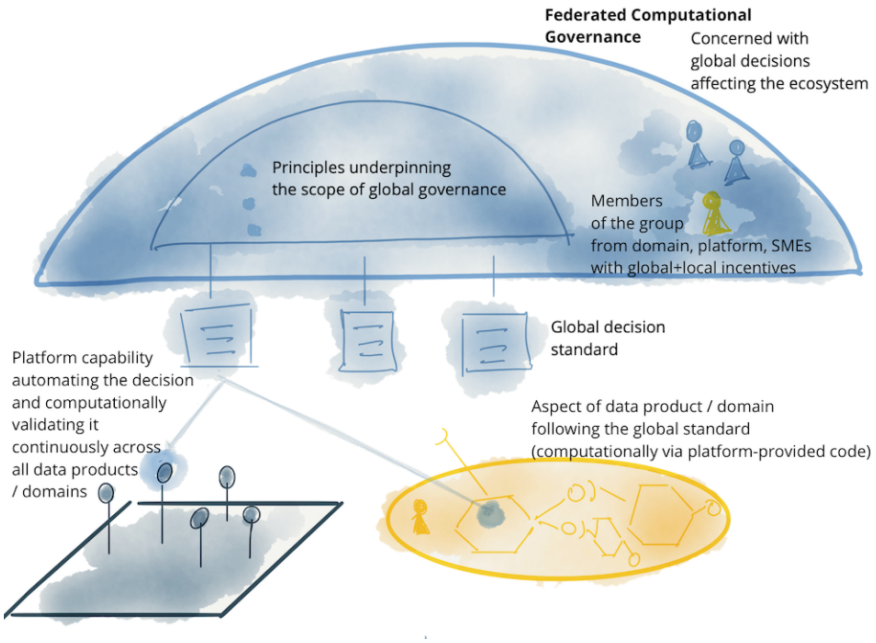

Figure 3.8: Federated computational governance model

3.4 Microservices

The distributed architecture is obtained thanks to *microservices*: they are an architectural approach to building applications, in which the software architecture is organized in small independent components that run each application process as a service and communicate with each other via defined APIs. As a result, this architecture allows each service to operate without compromising the others: this distinguish the microservices-based architecture differs from the previous monolithic approaches. Moreover, microservices architectures proposes a reorganizing of the development teams: this approach offers the ability to manage unavoidable criticalities, supports dynamic scalability and facilitates the integration of new features.

3.4.1 Monolithic architecture vs microservices architecture

All processes in monolithic systems are tightly interconnected and run as a single service (Figure 3.9). Therefore, a change in the monolithic application requires a change in the en-

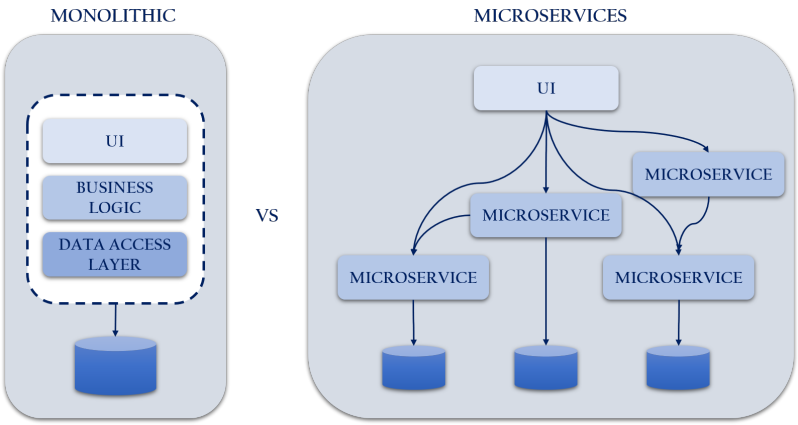

Figure 3.9: Monolithic architecture vs microservices architecture

tire architecture: it leads to an increase in the base code making the service more complex and, in turn, increases the risk of error. By constrast, in a microservices-based architecture each service is:

- is self-contained and perform only one function;
- may be developed, updated, distributed, executed and scaled without breaking down the whole functioning of the other components;
- has its own code, which eliminates the need for the numerous components to share a single huge code and l that the error rate decreases;
- specialized in solving the specific problem for which it was designed.

4 Technologies and services

This chapter presents the technologies used during this project. Amazon Web Service (AWS) is the environment on which the resources are provisioned. The technologies used on this cloud provider, such as lambda functions, API Gateway, DynamoDB, S3 bucket and EC2 instance, are therefore in-depth, explaining their operation. Finally, this chapter mentions the SDK for Java (i.e. how developers can deploy on the AWS environment) and the REST API (how the client communicates with the server).

4.1 Amazon Web Service

Amazon Web Service (AWS) is a global cloud platform used to power infrastructure and applications. It is the most used cloud platform in the world as it provides more services and more features than other *Cloud Service Providers*[1]: AWS provides on-demand pay-as-you-go services, making it faster, easier and more convenient. In addition, AWS lets you experience the latest technologies to quickly transform and innovate your business. The AWS network is global and for each region there are several available locations physically separated and connected by low latency and high throughput, with highly redundant networks. This simplifies the design and operation of applications that are scalable, fault tolerant and highly available. With the largest customer community in any industry, AWS has unmatched experience and operational expertise.

The cloud-based products offered by AWS are several, including lambda, API gateway, dynamo database and other functionality (analyzed in this section) that allow organizations to build sophisticated applications, reducing IT costs, in a scalable and reliable way.

[1]Cloud Providers are companies that provide IT environments such as public clouds or managed private clouds that extract, collect, and share scalable resources across a network. Cloud providers can also offer online services such as Infrastructure-as-a-Service, Platforms-as-a-Service, and Software-as-a-Service.

4.1.1 AWS Lambda

AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS), that allows organizations to run code without having to setup or manage servers. Users of AWS Lambda create functions, writing self-contained applications in one of the supported languages (Node.js, Python, Go, Java, and others), and upload the code to AWS Lambda as a ZIP, Jar file or container image. AWS Lambda creates the instances of the function as they are requested, without worrying about any scale levels or settings to adjust. The Lambda functions can perform any kind of computing task, from serving web pages and processing streams of data to calling APIs and integrating with other AWS services, such as DynamoDB, S3 and API Gateway, allowing companies to build functionally complete applications.

The concept of *serverless* refers to a cloud execution paradigm, in which the Cloud Service Provider allocates machine resources as they are requested and no resources are consumed when the application is not in use. In addition, because AWS Lambda is a fully managed service, it takes care of all the infrastructure for organizations: this architecture allows them to avoid maintaining their own servers to run these functions. This is not to say that there are no servers involved in a serverless environment: it just means that the servers, the operating systems, the network layer and the rest of the infrastructure have already been taken care of, so that organizations can focus on writing just application code.

Main benefits of AWS lambda, so, could be summarized as follow:

- *pay per use*: prices are determined solely by the number of requests made and the amount of time it takes to process the code;
- *fully managed infrastructure*: AWS Lambda takes care of all the infrastructure for code execution in a highly available and fault-tolerant environment, allowing developers to focus on creating backend services. AWS Lambda optimizes code distribution, handles all administration, maintenance, and security patching, and integrates logging and monitoring via Amazon CloudWatch;

- *automatic scaling*: without any user setup, AWS Lambda just invokes the code when it is needed and automatically scales to accommodate the frequency of incoming requests. A limitless amount of requests may be handled by the programming.;
- *tight integration with other AWS services*: on AWS, Lambda serves as the primary compute service. It also interfaces with a variety of other AWS services and, along with Amazon API Gateway, Amazon DynamoDB, or Amazon S3 bucket, forms the foundation for AWS serverless solutions..

However, there are also limitations to be taken into consideration when using lambda functions in production. Some examples are given below:

- *cold start time*, which occurs when a function is not called for some time and there is a latency between the event and the execution of the function, therefore critical for applications that rely on lambdas for latency;
- *execution time/run time*: after 15 minutes of running, a Lambda function will time out;
- *code package size*: the code within the zipped Lambda should not exceed 50MB in size, and the unzipped version should not be larger than 250MB;
- *payload size*: the maximum payload size that API Gateway can handle when triggering Lambda functions in response to HTTP requests is 10MB.

How AWS Lambda works

A lambda function contains the code and some configuration information, such as the function's name and resource requirements. Each Lambda function has its own container in which it executes. Lambda packages the created function into a new container and executes it on a multi-tenant cluster of AWS-managed machines when it is built. First, each function's container is given the required RAM and CPU capacity, then, the execution of functions can start. On completion, the RAM initially allocated is multiplied by the time elapsed by the running function. As previously stated, this is a *pay-per-use* service

and, with this mechanism, the client pays only depending on the allocated memory and the amount of execution time necessary to execute the function (Figure 4.1). Once the Lambda function is created, it remains always in a listening state (i.e. always active and ready to be triggered).

Figure 4.1: How AWS Lambda works

AWS Lambda's whole infrastructure layer is controlled by AWS. Customers do not have much insight into how the system works, but they also don't have to worry about, for example, upgrading underlying machines or avoiding network conflicts, because AWS handles everything for them, allowing them to save time on operational activities. One of the unique architectural features of AWS Lambda is the ability to execute many instances of the same or different functions from the same AWS account at the same time. Concurrency is not a problem for Lambda: clients only pay for the calculations used by their functions. This makes AWS Lambda a good choice for deploying highly scalable cloud computing solutions.

4.1.2 AWS API Gateway

Amazon API Gateway is a fully managed service that allows developers to easily manage the creation, publication, maintenance, monitoring, and security of APIs at any scale. The APIs act as a "gateway" to allow applications to access data, business logic or functionality from back-end services. Developers, through API Gateway, define the HTTP endpoints of a REST API or a WebSocket API and connect them to the corresponding backend business logic. Furthermore, API Gateway is compatible with containerized and serverless

workloads; in fact, many serverless applications use Amazon API Gateway for replacing the API servers with a managed serverless solution.

API Gateway develops RESTful APIs that:

- are based on HTTP;
- allow stateless client-server communication to take place;
- implement standard HTTP methods such as GET, POST, PUT, PATCH, and DELETE.

Within the Serverless ecosystem, API Gateway is a very important element as it acts as a *glue* for serverless functions and API definitions, enabling a full-fledged serverless architecture for web applications. Also, it is able to trigger the execution of a Serverless function directly in response to an HTTP request.

API Gateway is in charge of accepting and processing hundreds of thousands of API calls at the same time (see Figure 4.2). Traffic management, permission and access control, monitoring, and API version management are examples of these tasks.

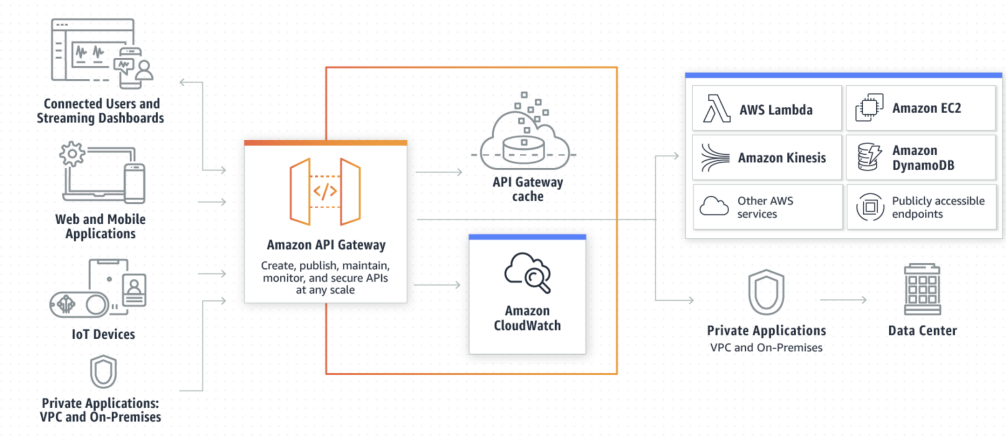

Figure 4.2: AWS API Gateway

The primary benefits of using an API Gateway to construct HTTP APIs in AWS (in addition to those derived from the structure of the serverless paradigm, such as scalability, minimal maintenance, and low cost) are as follows:

- *an API Gateway event may be used to bind HTTP requests to specified functions in*

a Serverless application: API Gateway makes it easy to connect HTTP endpoints to individual functions, eliminating the need for a specialized API server;

- *use several microservices to provide the same top-level API*: API Gateway makes it simple to utilize various Serverless methods to serve various API components. This means that it is possible to better encapsulate functionality in each function and clearly distinguish the business logic of the API's various components. Another advantage is that the function called by a specific API request can be altered transparently: consumers will not notice any changes at the API level;

- *save time with integrations: Authentication, Developer Portal, CloudTrail, CloudWatch*: companies do not need to operate their own authentication systems, API Gateway allows them to build a fully controlled authentication and authorisation level.

By contrast, there are some limitations to keep in mind before putting API Gateway into production:

- *integration timeouts*: the shortest possible timeout for an API Gateway integration is 50 ms, and the longest is 29 seconds;

- *regional APIs*: only 600 regional APIs are allowed per AWS subscription;;

- *payload size*: the maximum payload size that an API endpoint can return is 10MB. In order to return more data, it is need to divide the payload into numerous pages.

How API Gateway works

API Gateway lies between API users and backend services, processing HTTP requests to API endpoints and routing them to the correct backends. It comes with a collection of tools to assist clients in managing API definitions and mappings between endpoints and the corresponding backend services. It can also build API references from definitions and publish them as API documentation for the users.

API Gateway works with a variety of AWS services (AWS Lambda, AWS IAM, etc.) and it serves as a "front door" for applications to provide access to data, business logic, or

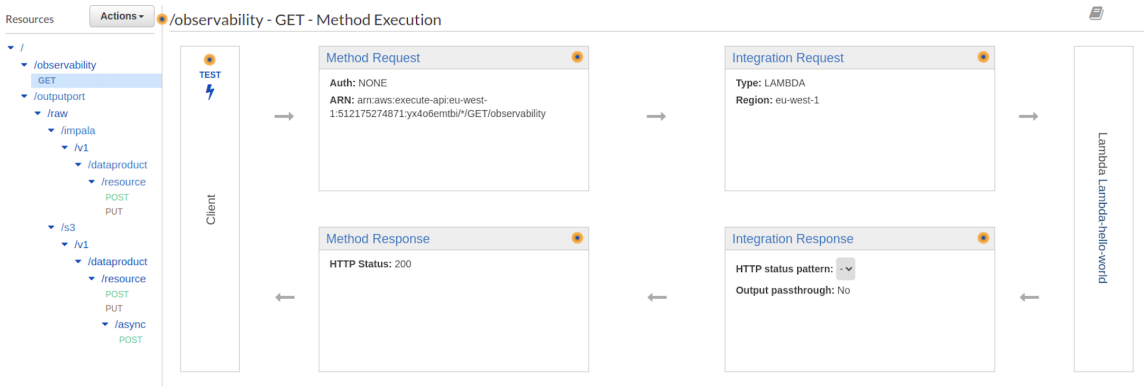

Figure 4.3: API Gateway on AWS environment of the case study explained in Chapter 5

functionality from backend services like Amazon Elastic Compute Cloud (Amazon EC2) workloads, AWS Lambda code, any web site, or real-time communication applications.

As it is shown in Figure 4.3, the graphical user interface of Amazon API Gateway allows users to sketch the layout of an API and observe API flows in graphical form once they have been established. However, it is also possible to create an API Gateway through structured code, such as serverless services or through the SDKs provided by Amazon, in a programming language of the developer's choice.

4.1.3 AWS DynamoDB

Amazon DynamoDB is an Amazon Web Services (AWS) fully managed NoSQL database service that supports document-type and key-value data formats, which means that it does not store data in a structured, relational manner; instead, it saves JSON objects in a simple key-value format. It is a highly scalable, multi-region, multi-active database with built-in security, backup and recovery, in-memory caching for Internet applications, and offers performance of a few milliseconds at any scale.

Serverless application values are matched with DynamoDB: autoscaling based on application load, usage-based pricing, simplicity of setup and tight integration with the AWS serverless solution and AWS Lambda (an example is shown in Figure 4.4). As a result, DynamoDB is a popular choice for AWS Serverless applications.

As a key-value store, DynamoDB is particularly simple to use when a single element

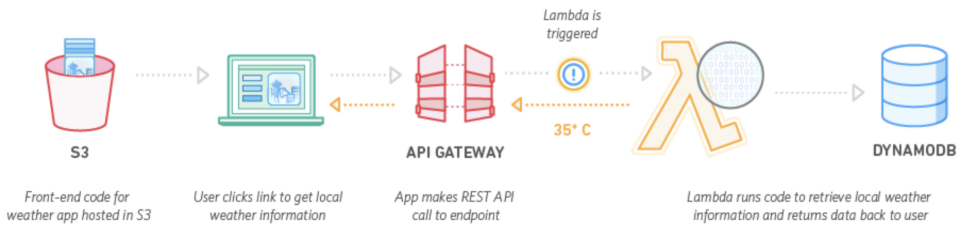

Figure 4.4: An example of a weather application which shows how interacting with DynamoDB

in a single DynamoDB table includes all the data required for a discrete operation in the application. If the application dashboard displays a person and the books they have read, for example, DynamoDB will perform best and cost the least per request if the books are stored in the User object. However, keeping users in one database and books in another, when page loading requires retrieving one user and ten distinct book entries, may make DynamoDB less suited - extra queries are more expensive and slow down the application experience compared to a relational datastore.

DynamoDB's architecture allows it to be used in a variety of situations where conventional databases are ineffective; some of the benefits of DynamoDB are:

- *fully managed*: any operational activity is required to keep the database running, so there are servers to update or kernel fixes to apply. Using this solution cuts down on the amount of time the team spends on operations, enabling them to focus on product development;
- *performance and scalability*: DynamoDB scales automatically table performance or size as the application load increases;
- *automatic replication*: global tables in DynamoDB replicate data across different AWS Regions to give globally distributed applications rapid access to local data;
- *support for streaming*: DynamoDB allows to produce streams of data table changes. These flows can subsequently be used to initiate tasks in other AWS services, such as Lambda functions. This makes it easy to add automation based on DynamoDB data changes.

Before deciding to utilize DynamoDB in production applications, companies should consider the following disadvantages:

- *strong vendor lock-in*: DynamoDB is not open-source and, if the organization decides to stop utilizing DynamoDB, it will have to perform a lot of effort to switch to another database;
- *no join operations*: DynamoDB's design does not support to join data from across multiple tables.

How AWS DynamoDB works

DynamoDB stores data in tables, much like other databases. Each table contains a collection of items, each of which has its own set of fields or attributes. A primary key must be present in each table, present in all elements within the table to uniquely identify each item, and it can be divided in two categories:

- *partition key*: simple primary key, composed of a single attribute. The value of the partition key is used as input to an internal hash function in DynamoDB. The hash function's result defines which partition (physical storage within DynamoDB) the item will be placed in. No two entries in a table with just a partition key can have the same partition key value;
- *partition key and sort key*: called also *composite primary key*, composed of a combination of two attributes (partition key and sort key) for adding flexibility when querying data. The partition key is used as explained in the previous point but, in addition, all items with the same partition key value are sorted by sort key value and stored together. It is conceivable for two entries in a table with a partition key and a sort key to have the same partition key value. Those two entries, however, must have distinct sort key values.

DynamoDB uses also one or more secondary indexes (optionally) to provide more querying flexibility. DynamoDB supports two types of indexes:

- *global secondary index*: an index having partition and sort keys that are not always the same as the table ones;
- *local secondary index*: an index having the same partition key as the table, but a different sort key.

By utilizing the primary key or by building custom indexes and using the keys of those indexes, it is possible to refer to specific elements in a database.

The DynamoDB table is not hosted on a single server. The data, on the other hand, is spread over several servers, ensuring excellent scalability and performance, but also making it impossible to connect to a database host and query the data directly. It must be used the DynamoDB HTTP API to write and read items to and from a DynamoDB table, either directly or through the AWS SDK or AWS CLI. Furthermore, reads and writes on DynamoDB tables may be grouped, even on various tables at the same time. Transactions, automated backups, and cross-region replication are all supported by DynamoDB.

4.1.4 Amazon S3

Amazon Simple Storage Service (Amazon S3) is an object storage cloud service. The primary goal of S3 is to store any type of file in any quantity in the cloud for a variety of use cases (including data lake, *data mesh*, websites, mobile applications, backup and restore, storage, enterprise applications, IoT devices, and big data analytics), while providing industry-leading scalability, data availability, security, and performance. It can be stored static content for serving it directly to the end users, or internal data (configuration, system state, intermediate states and logs).

To store application data that does not fit into a database, serverless applications often employ cloud storage services like Amazon S3. As a result, S3 serves as the foundation for all Serverless solutions that deal with big files such as user-generated data, pictures, or video material. The static components of Serverless systems, such as HTML pages, pictures used on Serverless websites, CSS files, and generated JavaScript code, are frequently hosted on cloud storage. This decreases the time it takes for Serverless websites to load, cuts operational expenses, and keeps Serverless application maintenance to a minimum.

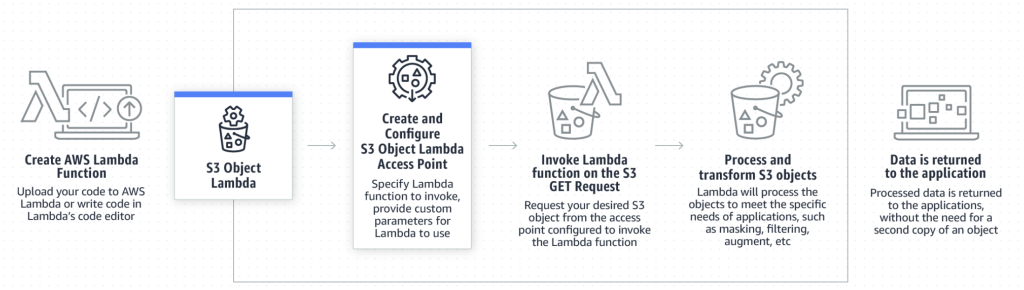

Figure 4.5: How Amazon S3 interacts with AWS Lambda

Amazon S3 interacts with other AWS services by sending alerts when a file is created or modified. AWS Lambda is one service that can act on these messages, and it's the most useful integration for Serverless developers (an example is shown in Figure 4.5). When files in S3 change, serverless functions may be configured to activate, allowing for many types of S3-based business logic. Several AWS services are able to read and write data directly from and to S3, for example, AWS Lambda functions. S3 also works with AWS IAM to control who has access to which files and buckets. In addition, for monitoring and logging, S3 interfaces with AWS CloudWatch and CloudTrail.

Here are some of the most common benefits of Amazon S3 when creating Serverless applications:

- *scalable and fully managed*: users only have to worry about uploading, accessing, and controlling the lifetime of the data they care about, while S3 handles the actual storage. A single AWS account can have hundreds of S3 buckets, each of which can hold several terabytes or petabytes of files while maintaining low-latency access to each one;

- *setup is simple*: users just create a bucket, choose whether or not to make it publicly available, and begin uploading files right away;

- *integrations with other AWS services*: it interacts with other services, as already discussed.

Some of the drawbacks of Amazon S3 (to consider before using it in production) are:

- *high cost at scale*: the S3 pricing plan is based on a pay-per-use approach. While this works well for modest quantities of information, as users utilize S3 in production and amass files in multiple buckets, the overall cost rises over time;

- *unlimited scalability creates difficulties*: while having limitless storage is beneficial for fast developing system, it may also lead to a lack of questions being asked about what data developers are storing and why. As a result, teams may make bad design decisions when it comes to data storage.

How Amazon S3 works

Users must first create an S3 bucket in one of the AWS Regions before uploading objects to Amazon S3; then, users only have to upload and download the files they require. AWS is a managed service and it is in charge of keeping these files, as well as assuring their availability and delivering them to clients when they need them. Once a file has been uploaded to S3, it may be referenced using an S3 path, which includes the bucket name and the path to the file within the bucket. The files in the bucket can be viewed directly using their S3 URL if the bucket is set to be publicly available.

Buckets and objects are AWS resources in terms of implementation, and Amazon S3 provides APIs to manage them: for example, a file can be uploaded to S3 by logging into the AWS Console and using the S3 user interface, or by sending a request to the S3 API programmatically, either directly over HTTP or using one of the AWS SDKs.

A *bucket* is a container for objects. An *object* is a file and any metadata that describes that file.

The S3 service is subdivided into individual storage buckets; each bucket has a bucket name which is globally unique and shared by all AWS accounts. This implies that until a bucket is removed, its name cannot be used by another AWS account in any AWS Region. Moreover, each bucket can include a number of "prefixes" (similar to folders on local drive), which in turn hold the uploaded files.

Objects can be stored in one or more buckets, and each object can be up to 5 TB in size. An object is identified by:

- *key*: object name (used to retrieve the object);
- *version ID*: string generated by Amazon S3 when the object is added to the bucket. The key-vesionID pair uniquely identify the object;
- *value*: content that has been stored;
- *metadata*: a collection of name-value pairs that can be used to hold information about the object;
- *subresources*: object-specific additional information;
- *access control information*: resource-based control information, such as Access Control List (ACL) and bucket policies.

4.1.5 EC2 instance

The Amazon Elastic Computation Cloud (Amazon EC2) is a cloud computing service that offers safe and scalable compute capacity. It's intended to make web-scale cloud computing more accessible to programmers, providing self-service, on-demand, and resiliency infrastructure. It is feasible to get and configure capacity fast and easily using the Amazon EC2 web service interface.

Amazon EC2 is an instance (i.e. a virtual server in the cloud) that reduces the time it takes to build and deploy applications by eliminating the requirement for upfront hardware expenses. Amazon EC2 may be used to create as many virtual servers as needed, configure security and network services, and manage storage.

Amazon EC2 has the following features:

- environments for virtual computing (instances);
- preconfigured templates for instances, known as Amazon Machine Image (AMI), containing the bit packets required for the server (including the operating system and additional software);

- different configurations of CPU, memory, storage, and network capabilities (instance types);
- instance store volumes, which are storage volumes for temporary data that will be destroyed when the instance is suspended or stopped;
- elastic IP addresses are static IPv4 addresses for dynamic cloud computing;
- tags are metadata used to create and apply to Amazon EC2 resources;
- virtual private clouds (VPCs) or virtual networks are conceptually separated from the rest of the AWS cloud, can be built and optionally connected to the network.

Amazon EC2's primary benefits are as follows:

- it cuts the time it takes to spin up a new server from days or weeks to minutes, compared to days or weeks in the on-premise world;
- it can rapidly scale up and down in response to computational demands;
- it provides an easy-to-use interface for configuring capacity;
- it provides full administrative access to the servers, making infrastructure administration simple;
- it includes features like monitoring, security, and support for a variety of instance kinds (wide variety of Operating Systems, Memory, and CPUs).

4.1.6 AWS SDK for Java

The Java API for AWS services is provided through the AWS SDK for Java. By providing a collection of libraries that are standard and well-known among Java developers, the AWS SDK for Java makes it simple to access and integrates with AWS services, such as Amazon S3, Amazon EC2, DynamoDB, and others. It includes credential management, retries, data marshaling, and serialization as part of the API lifecycle support.

4.2 REST API

A REST API, often referred to as a RESTful API, is an application programming interface (API or web API) that adheres to the REST architecture style's restrictions and allows interaction with RESTful web services. Roy Fielding, a computer scientist, invented REST, which stands for REpresentational State Transfer and it is neither a protocol or a standard, but rather a collection of architectural restrictions.

The API (Application Programming Interface) is a collection of definitions and protocols for developing and integrating application software. They may be thought of as a contract between a data supplier and a data recipient: the API defines the content sought by the consumer (the request) and the content demanded by the producer (the response). As a result, the API serves as a bridge between users or customers and the online resources or services they wish to access.

In a REST dialog, the client sends a request to a URL (which represents the requester or the endpoint) via a RESTful API, specifying one of the following methods:

- **GET**: ask for read-only data;
- **POST**: add new information to the database;
- **PUT**: change data that already exists;
- **DELETE**: removes data from the system.

If necessary, the data is given in JSON (Javascript Object Notation) format into the body of the data request in the REST API, since it is the most common and easy format to work with in any programming language. The server executes the operation associated with the URL and responds with a response status code and, if applicable, a JSON object in the body of the request. The information, in addition to the JSON format, can be delivered in several other formats via HTTP, such as HTML, XLT, Python, PHP or plain text.

This can all be done with any language that understands HTTP, a protocol that works through a request/response model.

A very important element is the so-called *status code* which consists of an integer that the server enters in the response. This number alone provides a strong indication of the outcome of the request. Some of the more common examples are as follows:

- **200 "OK"**: operation performed successfully;
- **201 "Created"**: new resource created successfully;
- **400 "Bad Request"**: request not formulated correctly;
- **401 "Unauthorized"**: user not authenticated correctly;
- **404 "Not found"**: resource not found at the specified address;
- **500 "Internal server error"**: generic error in the server;
- **501 "Not implemented"**: the request type is not implemented on the server.

Another key aspect of REST APIs is the *stateless* client-server communication, which does not involve storing client information between get requests and each request is separate and unconnected.

5 | Case study

How to provision a data product

This chapter presents the work done during the internship period by the product team. The analysis of Low Level Design has allowed to identify the resources in the scope of analysis, the services and microservices involved during this first implementation of the Data Mesh solution, the requests and responses to each service and the actions for each of them. The components are analyzed in detail and the workflow, by each component, for the provisioning of a given product and resources. Finally, the steps taken for an end-to-end test are explained (manual test carried out by the product team that verified the operation of all the functions and communications between the various services as a whole).

5.1 Process of provisioning

The process of describing the procedures required to manage access to data and resources, and making them and other technical services available to consumers is known as *Provisioning*. In this context, it refers to the initial configuration of the services. This chapter describes how to provision a data product (Figure 5.1) and its resources. Specifically, the resources analyzed for this case study are three:

- *output ports*: a collection of resources associated to data that are serving the rest of the organization. Some example: files, SQL query interfaces, application programming interfaces (APIs) - anything that makes sense for that domain to represent its data;
- *workloads*: a collection of resources associated with a project;
- *observability*: used to obtain information on the quality of the data product through various metrics.

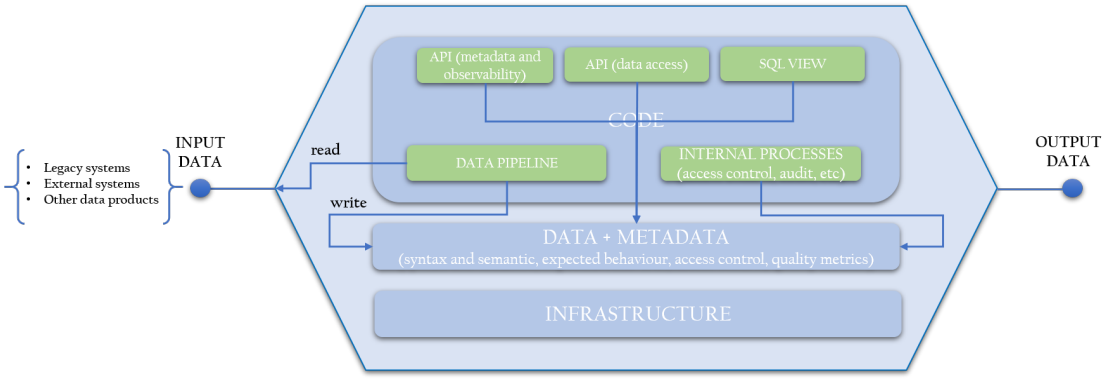

Figure 5.1: How the resources are divided inside a data product

5.2 Actors involved in provisioning process

The actors involved in the provisioning of a data product and its resources (Figure 5.9) are:

- *Provisioning Service*: it is the service that receives a request for data product provisioning, performs the necessary checks and retrieves the information it needs from other services in order to complete the provisioning of the data product and its associated resources.

- *Discovery Service*: it is a service for finding of the URLs of the individual provisioning microservices. For this first implementation, this service is a configuration file containing the mapping between a resource and the environment (in a real-world scenario, however, the provisioning service sends an HTTP request to the discovery service, which has a database with the microservices' URLs).

- *State Service*: it is the service containing the information about the data product and the resources and it is used for keeping track of the state of them. This information are stored in the dynamoDB database.

- *Microservice*: the single microservice manages the provisioning of the resource under its responsibility. For example, the provisioning service will interact with the microservice that controls s3 provisioning to manage a s3 resource; however, for a

spark resource, the provisioning service will select the spark resource component from the data product request and deliver it to the microservice who is responsible for it.

Each actor is invoked through different endpoints (based on the type of request and, therefore, of action). An example of them is defined in figure 5.2 for the Provisioning Service, in figure 5.3 for the microservices and in figure 5.4 for the State Service.

Figure 5.2: Provisioning Service endpoints

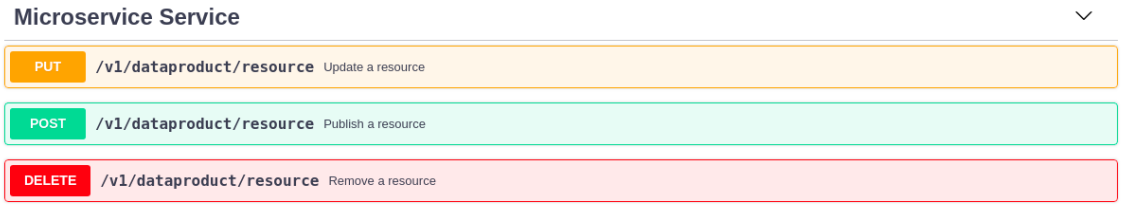

Figure 5.3: Microservice Service endpoints

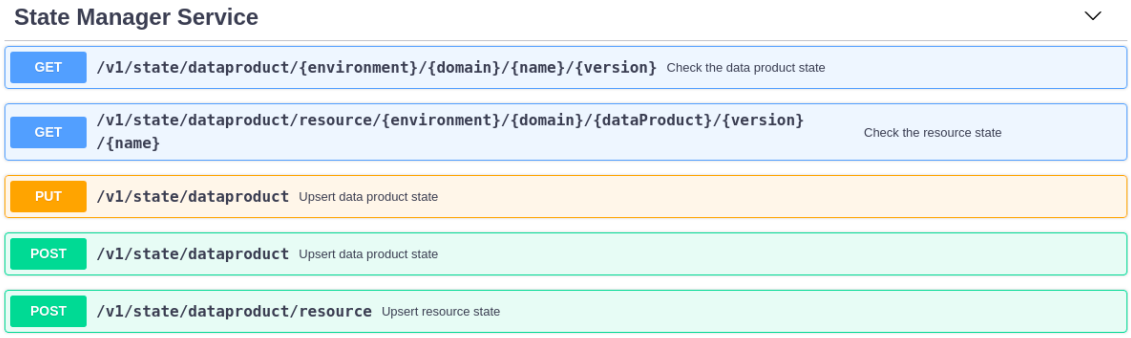

Figure 5.4: State Service endpoints

5.3 Deployment

Deployment is the process through which developers delivers applications, modules, updates, and patches to users. The deployment process can be considered as a phase of the software lifecycle, which concludes the development and related testing and begins maintenance. The processes used by developers to build, test, and release new code have an influence on how quickly and how well a product responds to changes in consumer preferences or requirements and impacts on the quality of the product. As a result, software developers have built procedures that enable software upgrades to be implemented faster and more often in the production environment, where users can access them. Cloud service providers (such as Amazon Web Services) offer Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) IT products help developers to deploy applications in live environments without incurring the additional costs of running their own storage and virtualization servers.

5.3.1 Software deployment methodologies

DevOps is the software development methodology used for new software updates in shorten delivery times, while maintaining high quality. It consists in a set of best practices that emphasizes the use of automation to streamline the software deployment process. Customers frequently use DevOps concepts to deliver serverless applications.

5.3.2 CI/CD pipeline

A *CI/CD pipeline* (Figure 5.5) is a mechanism for frequently deploying applications to customers and is used to automate data product deployment. It is built on the ideas of continuous integration, distribution and, in particular, deployment. The processes of continuous automation and monitoring throughout the life cycle of applications are interrelated and are thus described as *CI/CD flows*, which are supported by operational and development teams that interact in an agile manner.

In CI/CD, "CI" stands for *Continuous Integration*, which is a software development practice in which new code changes are regularly compiled and tested and, if everything is passed, developers merge code changes in a central repository (even multiple times a day), thereby solving conflicts between multiple branches of an application under development.

The abbreviation "CD" is short for *Continuous Distribution* and/or *Continuous Deployment* (two concepts that are sometimes used interchangeably). It adds the technique of automating the whole software release process to Continuous Integration. Engineers would still have to execute these stages manually if there was no automated pipeline, which would be considerably less productive.

Figure 5.5: CI/CD pipeline

Source stage

A source code repository often triggers a pipeline run. When code is changed, the CI/CD tool receives a notification and executes the associated pipeline.

Build stage

To create a runnable instance of the product, the source code and its dependencies are merged. Failure to pass the build stage indicates a serious issue with the project's setup.

Test stage

During this phase, automated tests are run to ensure that the code is correct and that the product behaves as expected. It's critical at this stage to get input to developers as soon as possible.

Deploy stage

The product is ready to be deployed after a runnable instance of the code has been produced and has passed all preset tests.

5.3.3 Deployment workflow

The source code of the data product is located on a GitLab repository and pushed to *GitLab runner*[1], which triggers the CI/CD. The application can be deployed everywhere with GitLab, by referencing the targets in the CI/CD pipeline: in this way, it is possible to interact with the chosen cloud provider (AWS) easily. The GitLab continuous integration fetches the source code from the repository, and then build, package, and deploy to the AWS account. The GitLab AWS provides the *AWS Command Line Interface* (CLI)[2] to run `aws` commands, specified directly from `.gitlab-ci.yml` (inside the Scala project) by specifying the AWS Docker image. The stages specified for this project and for each push are:

- `checkFormatting`: for checking the format of the scala code;
- `test`: test coverage;
- `build`.

When all stages are passed, they become green and the data product request can be POST to the Provisioning Service (Figure 5.6).

[1]GitLab Runner is an application that works with GitLab CI/CD to run jobs in a pipeline.

[2]The AWS Command Line Interface (CLI) is a tool to manage AWS services.

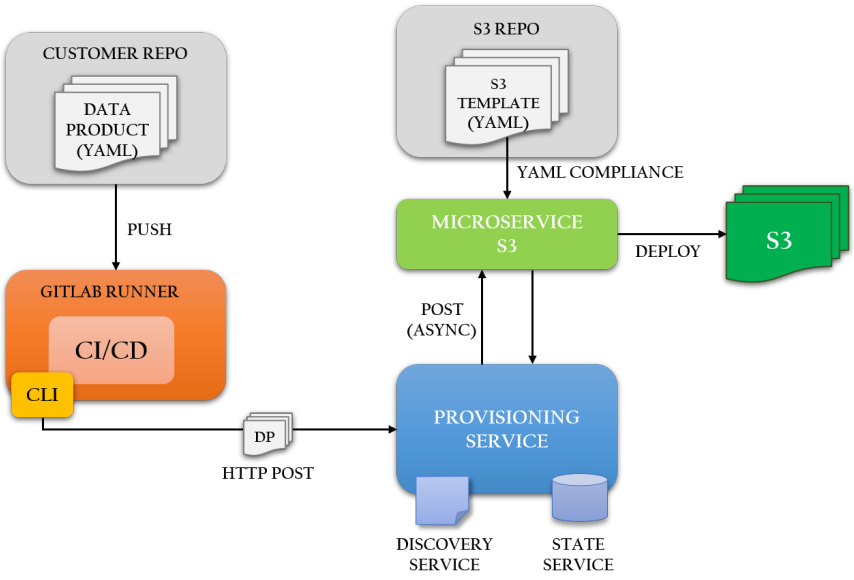

Figure 5.6: Deployment end-to-end of a data product with an s3 resource

5.4 Provisioning Service

The provisioning service receives the data product request in yaml format (Figure 5.7):

1. the yaml is parsed and the request is validated in order to verify that it is compliant with the defined interfaces: the high-level fields must not be empty (while the specific data are microservice-specific - received in json format - and are forwarded to the microservice of interest); there cannot be any resources that are duplicated; at least one resource is required;

2. it extracts the resources from the data product request, in order to create ad-hoc resource requests to submit to the microservice (Figure 5.8);

3. it asks the discovery service which microservice(s) it should contact: if the microservice is registered within the discovery service (in the case in question, in the configuration file), then the discovery service responds with the URL(s) to contact, corresponding to the microservice(s) endpoint(s);

4. once the URLs are obtained, synchronously, the provisioning service contacts the

name: data product name

domain: domain name to which the data product belongs

version: version of the data product

environment:

- ***name***: name of the environment
- ***specific***: additional fields of the environment

outputports: list of output ports

- ***name***: resource category that should be provisioned
- ***type***: type of the resource that should be provisioned
- ***technology***: technology that implements the chosen resource type
- ***description***: optional description of the requested resource
- ***owner***: DPO (IAM) role allowed to r/w access
- ***allow***: list of (IAM) roles allowed to read access the resource
- ***dependsOn***: list of resource names to which current resource depends on
- ***specific***: additional fields required by the resource specific type and technology implementation

workloads: list of workloads

- ***name***: resource category that should be provisioned
- ***type***: type of the resource that should be provisioned
- ***technology***: technology that implements the chosen resource type
- ***description***: optional description of the requested resource
- ***owner***: DPO (IAM) role allowed to r/w access
- ***allow***: list of (IAM) roles allowed to read access the resource
- ***dependsOn***: list of resource names to which current resource depends on
- ***specific***: additional fields required by the resource specific type and technology implementation

observability:

- ***name***: resource category that should be provisioned
- ***type***: type of the resource that should be provisioned
- ***technology***: technology that implements the chosen resource type
- ***description***: optional description of the requested resource
- ***dependsOn***: list of resource names to which current resource depends on
- ***specific***: additional fields required by the resource specific type and technology implementation

Figure 5.7: Structure of the data product yaml request

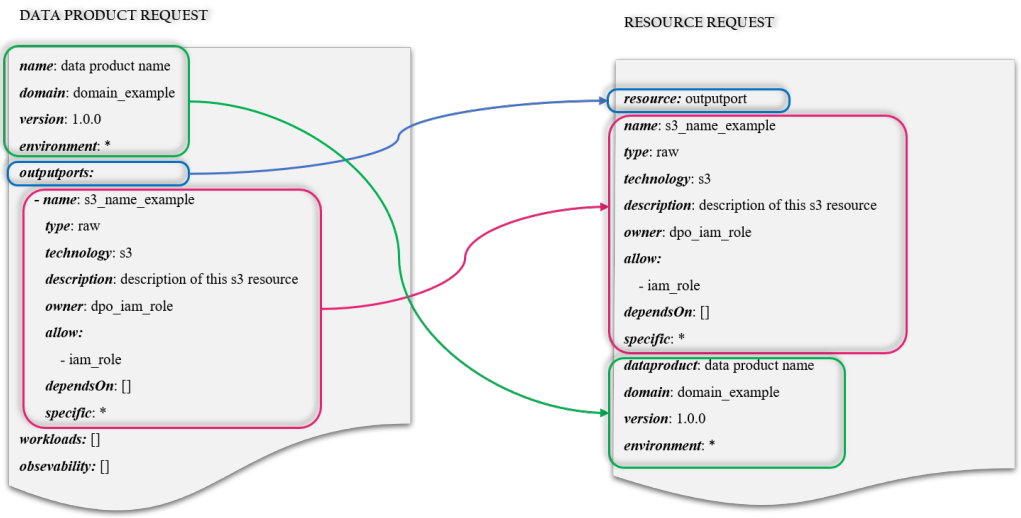

Figure 5.8: An example of how a resource request is created

state service in order to get the status of the data product basing on its linked resources: a data product is in a transit state[3] if at least one resource is in a transit state, and it is in a consistent state[4] if all resources are in a consistent state; all the operations[5] are performed if and only if the data product is in a consistent state. Accordingly, it checks the status of the resource on the associated URL previously retrieved, in consideration of the fact that it is possible to deploy in a case of consistent status of the data product and, in addition, it is also necessary to understand which method to call for the deployment;

5. if the data product is in a consistent state, then it extracts the HTTP method to invoke on the microservices based on the state of resources: if all resources are in the CREATED state, then the resource already exists and the provisioning service should update it, calling the PUT method; otherwise, the resources should be published, calling the POST method: initially, the resources are initialized to NONE, then the provisioning service updates the status of the data product and or the resources to CREATING (for the POST method)/UPDATING (for the PUT method). In particular, as previously stated, the data product is immutable: calling an UPDATE method

[3]Transit states: CREATING, DELETING, UPDATING

[4]Consistent states: CREATED, DELETED, FAILED, NONE, NOT_COMPLETED

[5]Operations: GET, POST, PUT, DELETE

on the resource does not mean modifying the resource itself, but the permissions of that resource (ACLs) or, in the case of a spark job, the jar necessary for the process linked to that resource, which is already defined and not modifiable;

6. for each resource, the provisioning service invokes the microservice endpoint associated: this operation is done asynchronously, due to the fact that some resources may depend on others (field dependsOn in the yaml body request). As a result of the dependencies, the provisioning service polling phase begins. For instance, in a scenario that requires provisioning of an s3 folder and a spark job that must read from and write to an s3 folder, this bucket must have been previously created. If these two resources are deployed in simultaneously, a problem develops since the spark job starts but fails to complete because it cannot locate the s3 folder, which may not yet exist. This implies that all resources that depend on others (in the example, the spark job) must start if and only if everything else from which it depends has been created (in the example, the s3 folder). However, being an asynchronous operation, the provisioning service must know whether the resources have been created or not, this leads to the concept of polling: sending several requests to the state service at regular intervals asking if the microservice has updated the status of the resource that is dependent on another in CREATED: in the example, the provisioning service ask to the state service if s3 is created and, if so, the polling stops and starts provisioning of the spark job which depends on the creation of the s3 bucket, otherwise, it asks later.

An example of the deploy flow is shown in Figure 5.9.

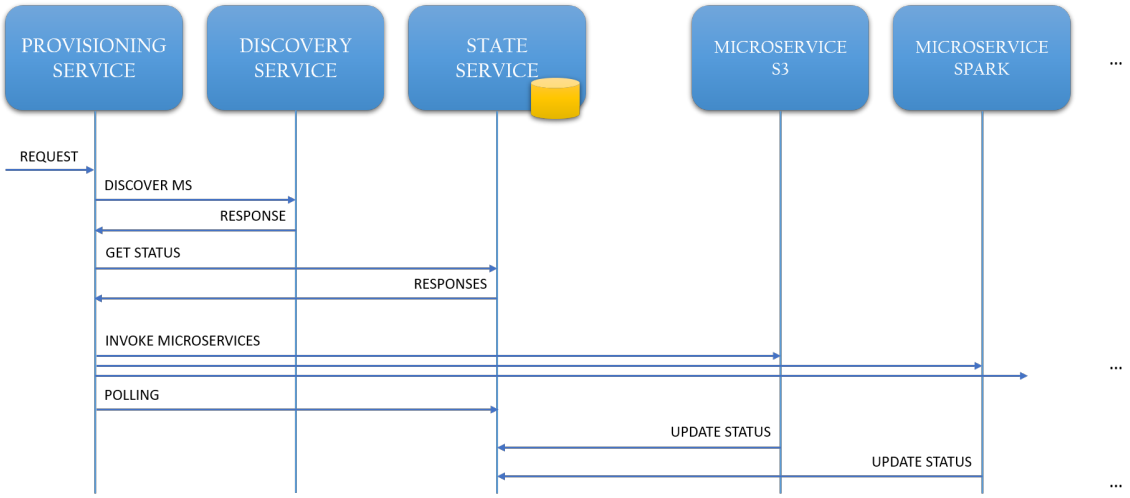

Figure 5.9: An examlpe of deploy flow

The pseudo-code of the deploy function is shown in Code 5.1.

```
1 def deploy(dataProductRequest) {
2   dp = validateRequest(dataProductRequest)
3   resources = getDataProductResources(dp)
4   urls = getResourcesUrls(resources)
5   state = getResourcesState(dp)
6   method = extractResourceMethod(dp, state)
7   deployDP(dp, urls, method, resources)
8 }
```

Code 5.1: Pseudo-code of the deploy flow

5.5 Microservices

Microservice service is the service that exposes the endpoints of that microservice. A microservice is not deployed on a server (as in the case of the Provsioner) but on serverless providers (such as, in this specific case, the lambda functions). In light of this, a microservice is deployed through a lambda: it receives the request from the Provisioning Service for the purpose of deploying a resource and, after extracting the request method, checks if the request is compliant with what it expects using a synchronous call (as each

microservice requires a specific request body for provisioning a certain type of resource). Then, the microservice responses 200 OK to the Provisioning Service (indicating that the first checks were successful) and, hence, proceeds with the next steps of the deployment (which differ according to the resource and, therefore, to the microservice). As soon as provisioning is complete, the microservice updates the status of the resource, unblocking the provisioning service from its polling phase.

5.5.1 Microservice s3

The microservice for the s3 resource receives a first request from the Provisioning Service and manages this request with a synchronous lambda; first of all, this lambda extract the request method, proceeds with the parsing and the check of the compliance of the yaml and, from now on, acts differently according to the method request:

- **POST/DELETE**

  The microservice asks for the status of the resource to the State Service and, if the resource is in a consistent state, it invokes an asynchronous lambda that will handle resource provisioning. As a result, the asynchronous lambda receives a provisioning request from the synchronous lambda and, along the lines of the synchronous lambda, controls the request method. For a POST/DELETE request, the lambda parses the request's yaml and modifies the status of the resource, informing the state service about the provisioning in progress:

  1. if the resource does not exist and, therefore, is published (through a POST method), then the microservice will contact the state service asking to update the status of the resource (currently on NONE) in CREATING; otherwise, in case a resource does exist and it must be deleted, then the status is updated (from any consistent state) to DELETING;
  2. the provisioning of the resource starts: the creation/deletion of the s3 directories on the AWS environment takes into account the *specific* field of the request to the microservice, containing the directories paths to be provisioned and the

name of the bucket s3. After that, the attachment (Code 5.3) or detatchment of the policies is managed: a policy is created / deleted to allow all users of that path to access the objects within the specified bucket and directory. The method by which the policy attachment is handled is the same used to make the policy difference (therefore, during the ACL update). In fact, the same method checks if the role in question has that policy already associated and, if so, the old policy is deleted and the new one is attached; on the contrary, if the policy is not associated with that role, then the put will provide directly with the attachment.

3. if the provisioning operation was successful, the microservice contacts the State Service again, asking to set the status of that resource to CREATED/ DELETED.

- **PUT**

  The microservice proceeds directly to update the ACLs of the resources in the synchronous phase: it is not necessary for the state service to be contacted for a change of the status. The microservice invokes the diffPolicies() method, whose pseudo-code is shown in Code 5.3.

The pseudo-code for the deploy of an s3 resource (specifically, the creation of s3 directories) is shown in Code 5.2.

```
1  def deploy(s3Request) {
2    // create s3 directories
3    dirsToProvision = s3Request.specific.dirs
4    foreach(dir in dirsToProvision) {
5      if(s3Request.bucket not contains dir) {
6        putObject(bucket, dir)
7      }
8    }
9    // attach policy
10   diffPolicies(s3Request)
11 }
```

Code 5.2: Pseudo-code of the deploy function for a s3 resource

```
def diffPolicies(s3Request) {
  policyAllow =     readonly
  policyOwner =     readwrite
  roles = s3Request.allow
  owner = s3Request.owner
  if(roles.exists && owner.exists)  {
    foreach(role in roles) {
      deletePolicyIfExistsWithOtherPermission(role, policyAllow)
      // attach policyAllow to role
      putRolePolicy(role, policyAllow)
    }
    deletePolicyIfExistsWithOtherPermission(role, policyOwner)
    // attach policyOwner to owner
    putRolePolicy(owner, policyOwner)
  }
}
```

Code 5.3: Pseudo-code of the diff function for adding or updating ACLs

Set up the AWS environment

Before provisioning the s3 resource, it is needed to set up the AWS environment:

1. create the lambda functions (sync and async) on the *Lambda* section;
2. associate the API Gateways to those lambdas (on the *API Gateway* section) – create the resource within the REST API of that data product (the path must be compatible with the defined *swagger*[6]);
3. associate the defined methods (POST, PUT, DELETE) to the resource;
4. deploy the API (it allows the gateway to listen on the s3 endpoint);

[6]Swagger: the OpenAPI Specification (or Swagger Specification) is a specification for describing, producing, consuming and displaying RESTful services.

5. update the config file (i.e. the Discovery Service) within the provisioner.

5.5.2 Microservice observability

There is no distinction between synchronous and asynchronous lambda in this sort of observability resource, as it just requires a single (synchronous) lambda. The objective of this resource is to obtain information on the quality of the data product using metrics; this varies from the *auditing* operation, which involves the use of logs to obtain information on the state of the program itself (e.g. server activation, start of the deployment phase of the data product X, etc.). The observability API should standardizes the way each data product exposes its information towards the data mesh. This API gives data product owners access to data product status, workload status and events, data quality and consumer experience metrics, regardless of technology and language used. The observability microservice should deploy a lambda, which provides the requested information.

The microservice receives a request from the Provisioning Service, checks the HTTP method request (in this case, as can be seen from Figure 5.10, it receives a POST request, as it is easy to understand that each time, for provisioning an observability is asked just for quality of the data product: this does not affect other resources through the updating and/or cancellations - the observability provisioner will be called each time new metrics are required); the yaml request is parsed, then it checks if the request is compliant with what it expects and, finally, deploys the resource: it creates a lambda function by extracting the jar (containing the function code) from an s3 bucket and integrates an API Gateway, which will contain the resource and the dedicated HTTP method to invoke the lambda (Figure 5.10).

Create an AWS lambda function

To create the lambda function (if it does not already exist), the microservice retrieves, first, the function code(s) and then, can proceed with the request(s) for the creation of the lambda(s). For the purpose of doing that, the microservice gets the s3 bucket information (i.e. the bucket from which to extract the jar), the artifact (i.e. the path to the jar itself

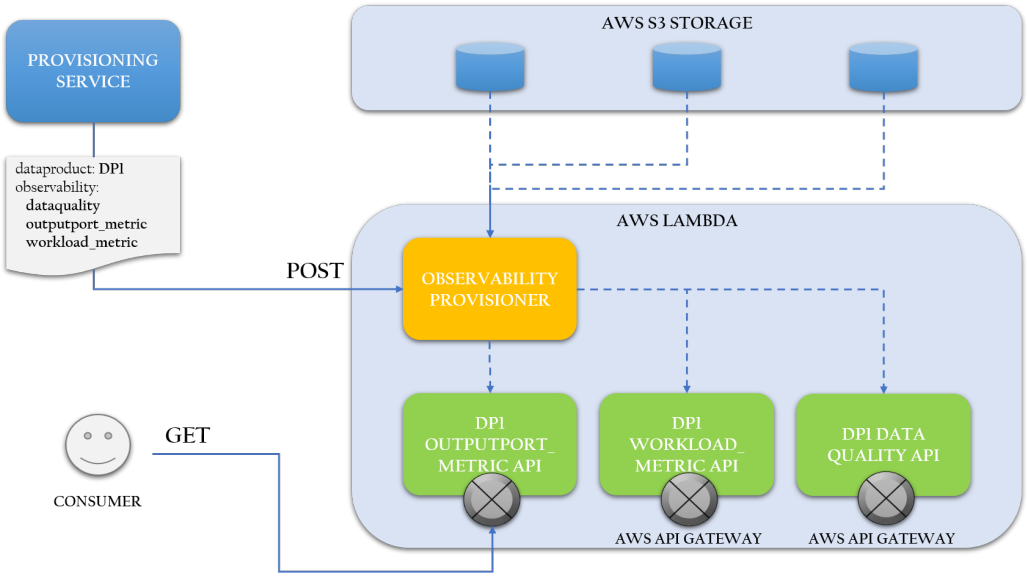

Figure 5.10: An example of the deploy flow for an observability resource

inside the s3 bucket), the name of the lambda function and, finally, the handler url (that is the name of the method that triggers the lambda). This information is extracted from the field *specific* of the observability request (Figure 5.7 - observability section). In particular:

- the *artiact-handler-lambdaName* triple is unique and represents the *endpoint* of the lambda; for each request it is possible to have more triples and, therefore, more lambdas to deploy (Figure 5.11);
- the *artifact-bucket* tuple (or, in the absence of the bucket name specification, *artifact-domain*) identifies the function code to be used to create the request for the creation of the lambda function.

Create an AWS API Gateway

After the creation of the lambda, it is necessary to integrate an API Gateway in front of it, able to invoke the function. The function which creates the API Gateway should:

1. check the existence of the REST API, whose name is extracted from the specific field of the observability request (ideally it is the same REST API as the other resources of the data product, such as s3, impala and so on);

Figure 5.11: Example of observability endpoints

2. create the observability resource under the root of the REST API and attaches the GET method to it (needed for retrieving the metrics and evaluating the quality of the data product), as shown on the top left of Figure 4.3;

3. retrieve the lambda function to integrate and proceeds with the integration of it, setting the method request, the integration request, the method response and the integration response (the main box of Figure 4.3 shows this process on the AWS environment);

4. deploy the API.

The pseudo-code of the deploy function for an observability resource is shown in Code 5.4.

```
1  def deploy(observabilityRequest) {
2    endpoints =  observabilityRequest.specific.observabilityEndpoint
3    foreach(endpoint in endoints) {
4      artifact = endpoint.artifact
5      lambdaName = endpoint.lambdaName;
6      handler = endpoint.handler;
7      restAPIname = observabilityRequest.specific.restAPI
8      bucket = observabilityRequest.specific.bucket
9      if(endpoint.lambdaFunction not exists) {
10       // 1. Create lambda function on the AWS environment
11       functionCode = getFunctionCode(bucket, artifact)
```

```
12          lambdaFunction = createLambdaFunction(functionCode,
13                                                lambdaName,
14                                                handler)
15          // 2. Create API Gateway attached to the lambda
16          restAPI = getRestAPI(restAPIname)
17          resource = createResource(restAPI)
18          attachGetMethod(resource, restAPI)
19          integrateLambdaFunction(lambdaFunction, resource, restAPI)
20          createDeployment(restAPI, resource)
21      }
22    }
23  }
```

Code 5.4: Pseudo-code of the deploy function for an observability resource

5.6 First results: deploy end-to-end

Mock tests were run throughout the project to ensure that all of the Provisioning Service's functionalities, as well as all of the microservices developed up to that point, were working properly. In addition, tests were run on the individual microservices on the AWS environment (without interaction between them or between the provisioner and the microservice). After reaching a stable point (the completion of the provisioning service code and of at least one microservice), it was possible to test the operation of the entire end-to-end deployment process (interaction between the provisioner and the microservice), starting from a HTTP request to the Provisioning Service (which is able to interact with the microservice for provisioning the resource) on the AWS environment.

The Provisioning Service is running on an EC2 instance for these first results; this machine must be started in order to start the Provisioner; to modify the Provisioner (for example, to change the configuration file or update/move a jar file), open a secure `ssh` connection from the terminal, stop the server, make the change, and restart the server. The microservice available for the first end-to-end result was the microservice s3: the endpoint of this microservice is registered in the configuration file (which, as previously explained,

will be a distinct service called Discovery service in the future) and is listened to on the application in order to be called. The database, which contains the *dmp-dataproduct-state* data products table and the *dmp-resource-state* resource table, is hosted on AWS DynamoDB as the initial implementation. The s3 bucket has been called *dmp-datamesh-s3*, and the provisioning folders will be created within it.

Once the instance is ready, Postman[7] may be used to submit an HTTP POST request to the Provisioning Service, including the data product and resource s3 information (for IAM roles, it is assumed that they already exist within AWS).

Here's an example of a request:

```
name: test_name
domain: test_domain
version: 1.0.1
environment:
    name: test
    specific: {}
outputPorts:
    -
        name: test_s3
        resourceType: raw
        technology: s3cdp
        description: my s3 directory
        allow:
            - dmp-test-1
            - dmp-test-2
        owner: dmp-test-owner
        specific:
            directory: test_dir/full
            bucket: ms-datamesh-s3
        dependsOn: []
workloads: []
observability: []
```

[7]Postman is an API developer collaboration tool. By allowing developers to quickly make requests via a user-friendly interface, Postman improves the end-to-end testing experience.

Code 5.5: An example of POST request to the Provisioning Service from Postman

The s3 resource with the *CREATING* status is created within the resource table on DynamoDB as soon as the POST request is sent; remember that the state of the resources is updated by the microservices themselves, and this provides a first indication that the provisioning service and the microservice s3 have communicated correctly. Moving on to the data product table, it will be in the *CREATING* state as well. Both states will be updated correctly to the consistent state within a few seconds (i.e. when the microservice has completed deploying the s3 resource and updated the state of that resource).

The impala microservice was then tested: within the *Cloudera* environment, it creates (if it does not already exist) the database and the required table; the policies on Ranger are added and it is possible to make queries on impala that read from a file saved on s3 (assuming that the PARQUET file already exists within the path specified in the s3 bucket).

Conclusions

To sum up everything that has been stated so far, the data mesh paradigm shift (Figure 3.4) can be summarized according to these key points:

- from centralized ownership and governance to decentralized ownership and federated governance;
- from monolithic to distributed system;
- from pipeline as first-class concern to domain as first class concern;
- from data as a by-product to data as a product.

Other microservices (in addition to those stated) will be implemented, with the ultimate objective of achieving a truly full and functioning implementation of an architecture that will change and enhance the way data is processed.

Sitography

- *AWS Lambda - The Ultimate Guide:* https://www.serverless.com/aws-lambda
- *AWS Lambda Features:* https://aws.amazon.com/lambda/features/?nc1=h_ls
- *Cloud Products:* https://aws.amazon.com/products/?nc1=h_ls
- *Cloud Computing: Cosa sono i provider di servizi cloud?:* https://www.redhat.com/it/topics/cloud-computing/what-are-cloud-providers
- *AWS Lambda:* https://aws.amazon.com/it/lambda/?hp=tile&so-exp=below
- *Amazon API Gateway:* https://aws.amazon.com/api-gateway/?nc1=h_ls
- *Amazon API Gateway - The Ultimate Guide:* https://www.serverless.com/amazon-api-gateway
- *Architecture of API Gateway:* https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html#api-gateway-overview-aws-backbone
- *Amazon DynamoDB:* https://aws.amazon.com/it/dynamodb/
- *Amazon DynamoDB and Serverless - The Ultimate Guide:* https://www.serverless.com/dynamodb#what-is-dynamodb
- *Core Components of Amazon DynamoDB:* https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html#HowItWorks.CoreComponents.PrimaryKey
- *Amazon S3:* https://aws.amazon.com/it/s3/
- *Amazon S3 objects overview:* https://docs.aws.amazon.com/AmazonS3/latest/userguide/UsingObjects.html

- Buckets overview: https://docs.aws.amazon.com/AmazonS3/latest/userguide/UsingBucket.html
- *Amazon S3 - The Ultimate Guide:* https://www.serverless.com/amazon-s3#what
- *Amazon EC2:* https://aws.amazon.com/it/ec2/?ec2-whats-new.sort-by=item.additionalFields.postDateTime&ec2-whats-new.sort-order=desc
- *AWS Lambda vs EC2:* https://lumigo.io/blog/aws-lambda-vs-ec2/#_uncc1111d9mi
- *Amazon Elastic Compute Cloud:* https://docs.aws.amazon.com/it_it/AWSEC2/latest/UserGuide/concepts.html
- *SDK AWS per Java:* https://aws.amazon.com/it/sdk-for-java/
- *Cos'è un'API REST?:* https://www.redhat.com/it/topics/api/what-is-a-rest-api
- *REST API: cosa sono, come funzionano e come progettarle:* https://devacademy.it/rest-api-cosa-sono-come-funzionano-e-come-progettarle/
- *Specifiche OpenAPI*: https://it.wikipedia.org/wiki/Specifiche_OpenAPI
- *Big Data - Cosa sono e perché sono importanti:* https://www.sas.com/it_it/insights/big-data/what-is-big-data.html
- *Le 5V dei Big Data: dal Volume al Valore:* https://blog.osservatori.net/it_it/le-5v-dei-big-data?hsLang=it-it
- *Big Data - Cosa sono e perché è importante investirci:* https://blog.osservatori.net/it_it/big-data-cosa-sono
- *Architetture per Big Data:* https://docs.microsoft.com/it-it/azure/architecture/data-guide/big-data/
- *Agile Lab handbook:* https://handbook.agilelab.it/
- *Linkedin Agile Lab:* https://www.linkedin.com/company/agile-lab/
- *Metodologia agile:* https://it.wikipedia.org/wiki/Metodologia_agile

- *Scrum (informatica):* https://it.wikipedia.org/wiki/Scrum_(informatica)
- *How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh*: https://martinfowler.com/articles/data-monolith-to-mesh.html
- *Interview - Keynote - Data Mesh by Zhamak Dehghani*: https://www.youtube.com/watch?v=L_-fHo0ZkAo
- *Data mesh challenges common data assumptions*: https://sdtimes.com/data/data-mesh-challenges-common-data-assumptions
- *Data Mesh Principles and Logical Architecture*: https://martinfowler.com/articles/data-mesh-principles.html
- *Cosa sono i microservizi?*: https://www.redhat.com/it/topics/microservices/what-are-microservices
- *Cosa sono i microservizi?*: https://aws.amazon.com/it/microservices/
- *Process Of Provisioning*: https://www.webopedia.com/definitions/provisioning
- *Deployment*: https://it.wikipedia.org/wiki/Deployment
- *Software Deployment*: https://www.sumologic.com/glossary/software-deployment/
- *CI/CD Pipeline: A Gentle Introduction*: https://semaphoreci.com/blog/cicd-pipeline