POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Data contracts as a quality enforcement tool under a Data Mesh architecture

Supervisor
Prof. Paolo GARZA

Candidate
Sergio Andres MEJIA TOVAR

Company Tutor
Agile Lab s.r.l.
Emanuele MAFFEO

Academic Year 2022/2023

Abstract

Data contracts are tools that are gaining strength in the data engineering practice, used to enhance data quality during data ingestion. This thesis aims to establish a format specification for contracts and the associated system for enforcing these agreements, particularly targeting source-aligned data products in the context of a data mesh paradigm. This research addresses the pressing need in contemporary data engineering to maintain data quality amid an increasing production of data, proposing a proactive approach where data producers adhere to predefined quality standards at the source. For this purpose, an application was designed to manage and validate contracts, using a push-based data ingestion as the examined scenario.

The study analyzes existing proposals for data contracts and introduces the creation of a contract object that facilitates quality agreements between data producers and consumers. These agreements rely on declarative schema and quality rules, used to computationally generate validations to ensure full alignment and compliance. The system has been designed to incorporate a declarative finite state machine workload management system in order to offer flexibility in adapting data validation processes to diverse use cases.

The research and subsequent software implementation demonstrate the delicate balance in defining a data contract between general standardization and domain-specific quality expectations, which in turn heavily influences the design of the enforcement system to account for this demand of configurability. Nonetheless, by analyzing the implemented system it can be concluded that despite the time overhead as a result of the enforcement, data contract agreement and enforcement enable faster correction producer-side, ultimately elevating data quality for consumers.

This thesis has been developed as part of the Research & Development unit of Agile Lab s.r.l., with plans to integrate it into Witboost, Agile Lab main product.

Acknowledgements

This thesis is the fruit of the endless love and support of my family, who even in the distance have been the greatest pillar in my life and have helped me overcome even the biggest mountain. *Dad, mom, brother and cat*: this thesis is for you.

Thanks are also due to all my friends present in all corners of the world; thanks to my bachelor friends back in Colombia whom I miss fondly; and my master's friends here in Italy and back in their countries. All of you have seen me grow and become the person I am now; this work is the result of it and I couldn't be more grateful.

Finally, I want to thank the *Agile Lab* team, especially my tutor *Emanuele Maffeo*, and also *Nicolò Bidotti*, who opened their doors and offered me the place to grow professionally and meet an amazing set of people.

Esta tesis nació y creció con el infinito amor y apoyo de toda mi familia, que incluso en la lejana distancia me han sostenido y han constituido el principal pilar de mi vida, ayudándome a conquistar hasta el más grande pico. *Papa, mamá, Diego y Maya*: esta tesis es especialmente para ustedes.

Gracias a todos mis amigos y compañeros, presentes en cada rincón inimaginable de esta Tierra. Gracias a mis amigos Javerianos que extraño profundamente, a mis amigos de la maestría en Italia y en todos sus países. Todos ustedes me han visto crecer y convertirme en la persona en la que soy hoy; este trabajo es el resultado de aquello y no podría estar más agradecido.

Finalmente, quiero dar gracias a mis compañeros en *Agile Lab*, especialmente a mi tutor *Emanuele Maffeo* y a *Nicolò Bidotti* que me abrieron las puertas para crecer profesionalmente. En ellos conocí un grupo de increíbles personas.

— *Sergio Andrés Mejía Tovar, 27/10/2023*

Table of Contents

List of Tables

List of Figures

Introduction

Current organizations generate large amounts of raw data, product of their internal processes, interactions with customers and external entities, and so on. This bulk of information can be analyzed to extract knowledge to drive business. However, in order to achieve this, the information should be reliable and possess certain standards of quality, which often are overlooked by the producers of said data, and only considered at later stages, where it might be too late to fix or get around.

Data engineering as a discipline has roots in this necessity, as it focuses on designing the systems to gather and analyze data, responding to the exponential increase in the amount of both the generated data and the data sources inside such organizations. Based on this, different practices have appeared and vary greatly in their structure and foundations, shaping organizations in the process, and fueling growth thanks to data analysis, business intelligence, etc. that spur from this collection of data.

Transforming data into value to generate business growth and ensuring its quality properties, is done through a chain of transformation processes, cleanup tasks, and other kinds of steps, creating what is called a data pipeline, through which data flows and gets converted into knowledge. As a general rule, it always involves a **producer** that generates data that can be interesting to an organization, and a **consumer** that ingests this data and transforms it in a way so that meaning can be extracted. Producers can either belong to the same organization, from daily processes that persist the needed information for the business to work in operational databases, or external owners of information. In addition, ownership of the data and who holds it becomes of great importance in this process, as it affects tasks like pipeline maintenance or improvement, which are transversal to the production of data and are the ones in charge of answering to change and to external requirements.

Through the years, different design guidelines and architecture have been created to connect producers and consumers. Currently, data exchange occurs through a variety of channels, each adhering to its unique set of standards and structures, or sometimes, exhibiting a lack thereof, often tailored to serve specific use cases.

This heterogeneity in the way data is exchanged and handled has created a breach in quality preservation, since the absence of a universal standard for data exchange has allowed data to manifest in various forms, reflecting the design and technical flaws of the systems that produce said data. Most of these systems have been designed not with a data-driven approach, but rather to streamline the acquisition of information to achieve some business goal, and produce data only as a byproduct. And not unlike byproducts in real life, the quality of it can be greatly questioned.

For this reason, both producers and consumers must implement agreements and controls on the information they're handling, especially in order to ensure data quality, which is the main element to satisfy the requirements of the intended purpose of the data [1]. The medium to ensure data quality should thus be inserted into the data architecture of the organization and as such is dependent on the technologies and platforms used to implement it. Some of these agreements are implicit, like the choice of one technology instead of another and their underlying standards (i.e. relational databases and SQL language); but others are done explicitly, either via an agreement between parties, or as an imposition from one of them (usually from the producer). This way, when a producer chooses a specific technology, the protocol of communication, and the schema of the data, they impose the main aspects that govern the interchange of information.

Is in this context that a new way to approach the control on the exchange of information based on Data Contracts has started to gain strength on the last years, as businesses acknowledge that the quality of the data ingested by their analytic processes determines both the value of the output as well as increasing the return of investment by lowering the cost of handling bad data. The concept has existed for more than a decade, taking inspiration from service contracts and data licenses, but modern contract definitions require not only to define the properties of data but also to enforce them automatically as part of any data pipeline.

Following this trend, the purpose of this thesis is to explore the concept of data contracts, culminating in the design and implementation of an effective enforcement mechanism for their utilization in data exchange between two entities. The research gains particular relevance due to the increasing market interest in data contracts observed over the past year, which underscores the industry's recognition of the valuable role data contracts can play in improving data quality, security, and reliability within today's fast-paced data-driven landscape.

The thesis was developed as part of the author's job as a Big Data Engineer on **Agile Lab s.r.l**, an Italian IT consultancy company that focuses on Big Data, Data Mesh, and Research and Development (R&D) in these fields. The company was founded in 2014 and since then it has worked with large enterprises in fields like banking, logistics, manufacturing, insurance, and others.

Founded on the principles of agility, innovation, and cutting-edge solutions, Agile Lab has emerged as a dynamic force in the realm of data engineering and digital transformation. Established with a vision to empower organizations with data-driven insights, the company has continually pushed the boundaries of what's possible in the world of data engineering, being early adopters of the data mesh paradigm, offering consultancy to companies interested in shifting their processes into data mesh, as well as publishing a new product "Witboost" aimed to enterprises to solve data engineering challenges through discovery, enhancement, and automation guided by governance best practices, based on the data mesh paradigm practices, but not forcefully tying data architectures to it.

The current document presents a chapter-structured approach to present the performed analysis and subsequent work. First, an examination of the existing body of research in the field of data engineering, data quality, and data contracts, setting the stage by establishing the current state of affairs.

With a solid foundation in place, the thesis transitions into the problem specification phase. Here, the scope of the problem to be tackled is identified, as well as the work to be done to undertake it.

Subsequently, the document follows the analysis, design and implementation phases done through iterative agile development. Central to this phase is the development of both a standardized data contract format and a robust enforcement system. The approach is guided by the insights gained during the problem specification phase, ensuring that the proposed solution addresses the identified issues while adhering to industry best practices.

Finally, results and conclusions are presented. This section not only showcases the outcomes of the performed work, but also evaluates the impact of the implemented data contract format and enforcement mechanism on data quality and exchange processes. It serves as a platform for drawing meaningful conclusions, discussing implications, and offering valuable insights for future research within this dynamic field.

Chapter 1

Background

In the center of the field of data engineering lies the creation of platforms that host processes built to collect data from different sources, transform it, and perform analyses in order to extract information and knowledge. The meaning of this data and the consequent extracted information and knowledge are used to make decisions. These three concepts: Data, information, and knowledge, work as the core of the practice, and their semantic differences have been extensively discussed. The definition brought by Ackoff [2] has been considered extensively to be one of the most clear, where the difference between data and information is functional rather than structural, and the latter is created by transforming data with the aim of increasing its usefulness. Knowledge instead is extracted from the information and functions as the answer to the questions posed by the original use case and business requirements.

In order to create knowledge from information, and information from data, is necessary to define both the transformations and the platform under which the data is placed and these transformations happen. The discussion on the best practices for the actual implementation of these processes has created along the years several different design guidelines and architectures to achieve the purpose of knowledge extraction from data. This doesn't always mean just the software implementation and integration on the other existing systems in the organization, but could also encompass a complete organizational shift in the way responsibilities are divided.

These guidelines build the baseline from which is possible to introduce further processes like data quality which functions as the ultimate tool to ensure a successful extraction of knowledge, and which should be inserted as part of these systems. The way data quality is performed is dependent on the actual implementation and this might differ on a case-to-case basis, both from the organization's viewpoint and from the data engineering team. Because of this, is fundamental to explain first the different kinds of existing data architectures and paradigms, to then introduce data quality, and how this can be inserted in the context of data transformation and data contracts.

1.1 Data architectures

Currently, the most widely used architectures are Data Warehouses and Data Lakes, so it's important to briefly mention how they approach data and also to highlight the current limitations of these architectures.

1.1.1 Data Warehouse

Data warehousing is a business-driven data architecture, where data is handled based on business goals. Golfarelli and Rizzi [3] define it as a repository for historical, integrated, and consistent data to conduct data analyses that help perform decision-making processes and improve information resources. These goals are specified by requirements that come from stakeholders of the organization and the data is modeled to serve them, allowing the quick creation of Business Intelligence (BI) systems, and other types of business analytics.

A Data Warehouse architecture is designed so there is a clear distinction between transactional (also called operational) and analytical data, where the latter is to be kept in a centralized, standardized, and scalable way. Dehghani [4] makes a great summary of how data is handled in a Data Architecture as follows: Data is extracted from many operational databases and sources that support the day-to-day business of the organization. All of the data is transformed into a universal schema represented in a multidimensional and time-variant tabular format via ETL jobs. It is then loaded into a centralized database called "warehouse" that stores the tables with the information. This warehouse is designed for On-Line Analytical Processing (OLAP) analysis. That is, big queries involving dynamic, multidimensional analyses on a huge amount of records, different from how On-Line Transaction Processing (OLTP) systems are performed, which usually involve only a couple of tables and records and it's aimed to be fast and offer ACID properties. Data is then accessed through SQL-like queries by data analysts for reporting and analytical use cases.

There are several types of actual implementations of data architecture for the warehouse, defined by the number of layers of separation of data and information. The typical base architecture for a Data Warehouse is composed of two layers, physically separating the operational data source and the data warehouse as seen in Figure 1.1.

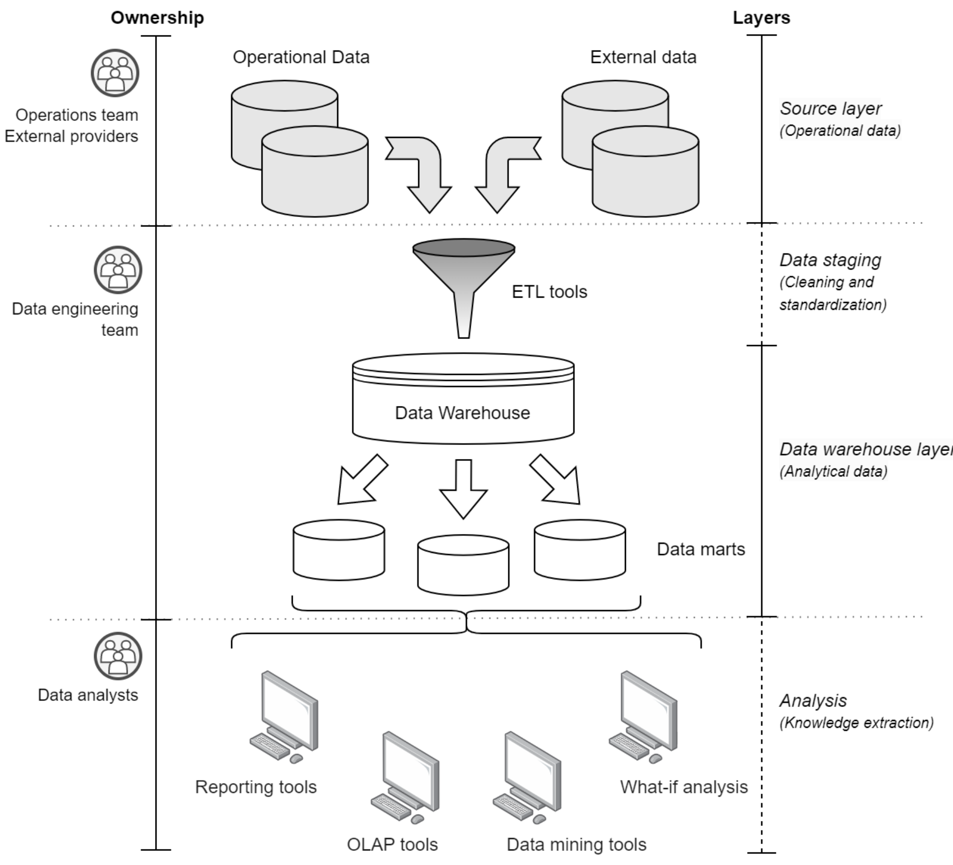


Figure 1.1: Two-layer architecture for a data warehouse. Adapted from Golfarelli and Rizzi, 2007 [3].

On this architecture, the data flow stage is defined by Lechtenbörger [5] as a series of layers with a specific responsibility, operation to be performed and a team that holds the ownership:

---

1. **Source layer**: Data is present on the operational databases
2. **Data staging**: Data is loaded from the operational databases, and it is cleansed and integrated, done through Extraction, Transformation, and Loading tools (ETL), a process that standardizes the format of the operational databases to the data warehouse dimensions.
3. **Data warehouse layer**: Clean Information is stored in one logically centralized single repository, which can be directly accessed or through data marts.

4. **Analysis**: Data is loaded from the data warehouse in an efficient and flexible way to create reports.

In data warehouses, the ownership of the data relies on a team of specialized data engineers and data scientists who maintain the warehouse and generate the queries needed for the analysts and BI. Most, if not all, of the requests for pushing/pulling data from the warehouse, pass through this team. Any change needed to the dimensions of the information, regardless of the business-oriented goal or context of the data, needs to pass through the data engineering team. This is a known potential source of problems as will be explained later.

1.1.2 Data Lake

Data lakes appeared as an evolution of data warehouses, although in reality they work side by side with them, as they're designed for slightly different use cases. They also consist of a centralized repository, albeit allowing to store both structured and unstructured data at any scale [6]. The key fact that differentiates the Data Lake and the Data Warehouse is how the data transformation is being performed. As explained above, the data warehouse uses ETL jobs, so the Transformation is done before saving the information on the warehouse to standardize the format, whereas on a Data Lake, an ELT process is more common, so the Transformation process is done after Loading the data on the lake.

This changes the fundamental way of saving and ingesting information. In a data warehouse, having the transformation process before data is loaded into the data warehouse means that the whole set of data is already present in a standard schema of set dimensions that can be directly queried. On a Data Lake however, the data is saved not in a standard format, but directly on the format of the source (be it structured or unstructured), improving the availability and timeliness of data but introducing new processes to be performed at a later point in time. In fact, to access the data or perform analyses, it is necessary to carry out the appropriate processing of the data (Transformation). Nonetheless, the advantage is seen in the fact that the data to be ingested is already readily present, and different Business Units can access newer information in a quicker way, not needing to wait for the Data Warehouse ETL pipeline.

In both these architectures, data passes through two clear main steps. The first step is delineated by the initial Extraction step of the pipeline to push data into the warehouse or lake, where these systems act as the consumer, ingesting data from external sources to an internal platform that is then considered as a data domain.

That is, a perimeter where the information is handled and the ownership of it belongs to a well-defined set of maintainers or a team, becoming the domain team. Upstream on the ingestion, both operational databases and external providers are considered producers in this step, acting outside of the domain of the lake or warehouse.

By ingesting data, it is now transferred inside the domain ownership and handled by their maintainers in order to allow businesses and data analysts to consume what has been exposed in the warehouse and lake systems. This acts as the second main step, where ownership is contained inside the architecture, and further requests to access data are handled by the domain team. This differentiation of ownership will be key to further quality checks and requirements that will be analyzed further in the document.

1.2 Data Mesh

Data warehouses and lakes solve the problem of collecting data from different sources and saving it in a reconciled format for the case of warehouses. However, this centralization of data brings further issues. As organizations grow in complexity and expect more data and in more instances, the systems are put under stress and the data engineering team that tenders for the warehouse or lake starts to fall behind the analysts' and stakeholders' requirements. Even if the systems are built to scale with the increasing amount of consumed information and access to it, the maintenance of these systems and the extension of them to provide new views on data, data marts, etc. doesn't scale as easily. In these scenarios, shortcuts are taken and data quality and reliance on the data is the first thing to be lost. When this happens, is necessary to not only reorganize the way these systems are implemented, but also how they are handled, and the teams behind them. The Data Mesh paradigm aims to solve this, defined as a decentralized approach to data management proposed by Zhamak Dehghani and formalized in her 2022 book [4].

Is important to remark that Data Mesh is not only a data architecture, but it also entails a "sociotechnical approach to share, access and manage data on complex and large-scale environments" [4]. Its approach aims to manage large-scale data in a decentralized way, where the ownership of the data is not centralized but rather distributed among **domains**: Teams aligned to specific business domains and the information handled by it, being in possess of the knowledge about the real-life concepts and use cases of the data it produces. In this manner, it is able to offer better data that they then expose to the rest of the mesh and to the exterior, treating data as a company product with quality standards and expectations, "shifting the value system from data as an asset to a product to serve and delight

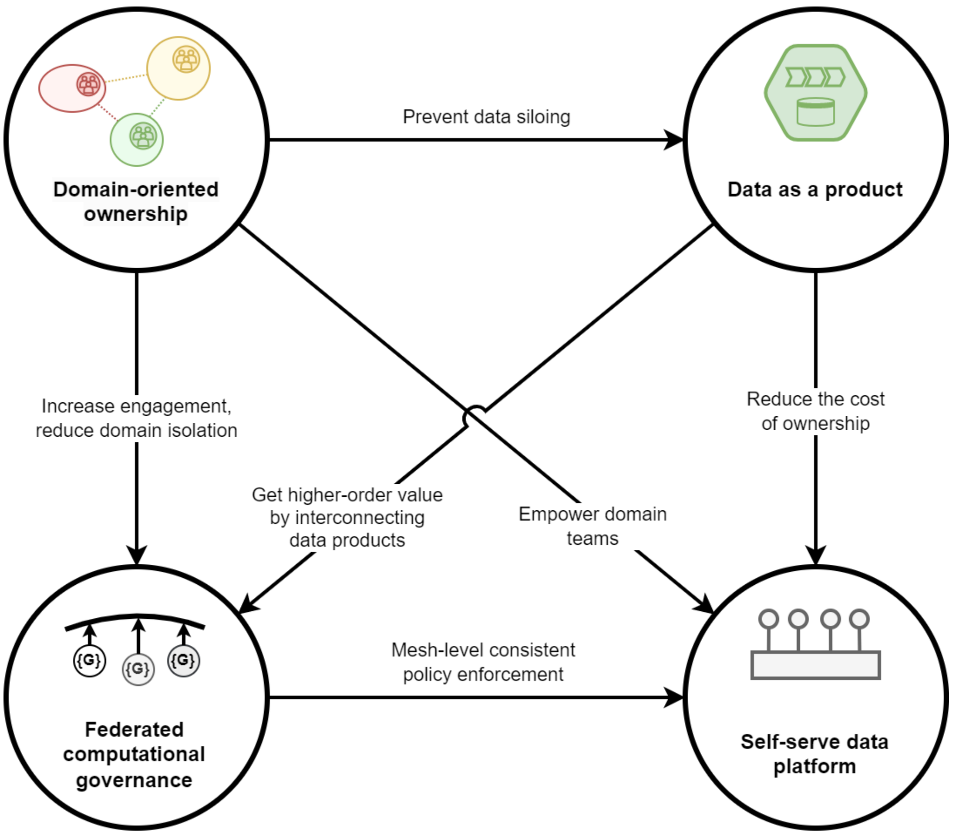

Domain-oriented ownership
Data as a product
Prevent data siloing
Increase engagement, reduce domain isolation
Reduce the cost of ownership
Get higher-order value by interconnecting data products
Empower domain teams
Mesh-level consistent policy enforcement
{G}
Federated computational governance
Self-serve data platform

Figure 1.2: Four principles of data mesh and their interactions. Adapted from Dehghani, 2022 [4].

the data users (internal and external to the organization).” [4]. Each domain team is independent from the others and in charge of handling their own data, ensuring their quality.

Architecturally, data mesh transitions the way of consolidating data, passing from monolithic data warehouses and lakes to a distributed mesh of communicating Data Products accessed via standardized ports. This requires strong data governance, a mechanism that ensures the system is secure, trusted, and delivers value through the interaction of the parts [4]. In Data Mesh, this mechanism is modeled through a federated model with automatic computation policies that are embedded in the mesh and that monitor the overall health of the environment.

The whole operation of a data mesh is supported under four principles defined by Dehghani that capture the logical architecture, operating model, and organizational shifts. These principles work together as seen in Figure 1.2 to serve high-quality data under a healthy mesh, empowering data teams to care for the produced knowledge. The principles are the following:

- Principle of Domain Ownership
- Principle of Data as a Product
- Principle of the Self-Serve Data Platform
- Principle of Federated Computational Governance

Operationally, these principles in action can work to create a data architecture that allows for decentralized ownership, while offering the interfaces to access data, manage governance, service data products on a platform, etc. Figure 1.3 summarizes this approach, where domains communicate with each other to access and share data while having transversal federated governance and a data platform to serve and discover the mesh data products.

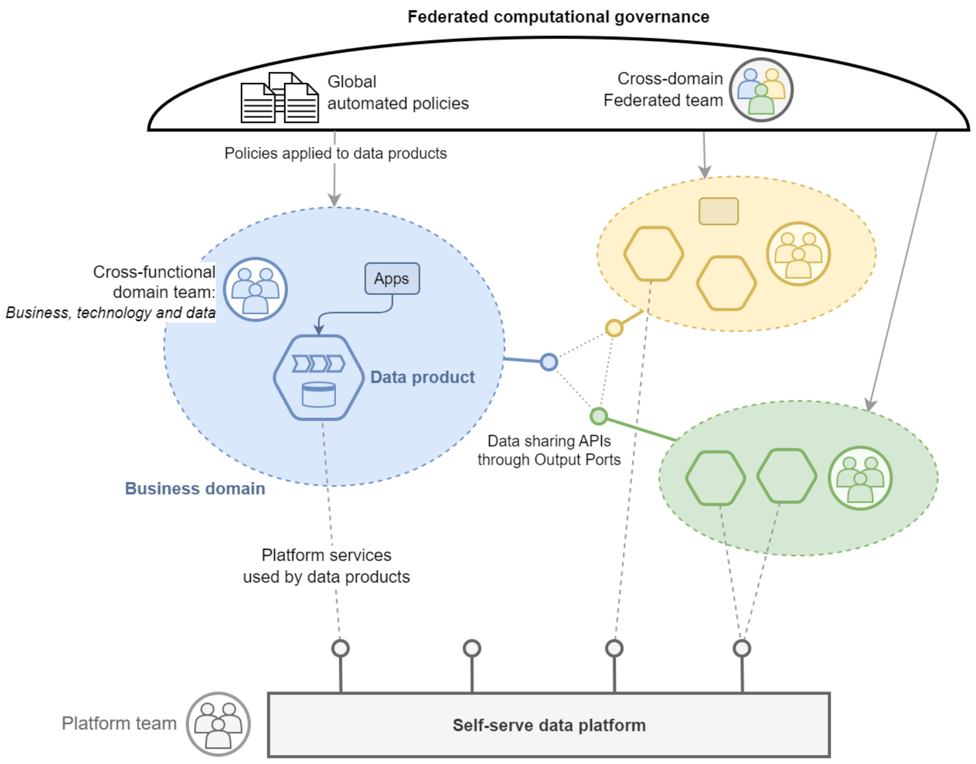

Federated computational governance
Global automated policies
Cross-domain Federated team
Policies applied to data products
Cross-functional domain team: Business, technology and data
Apps
Data product
Data sharing APIs through Output Ports
Business domain
Platform services used by data products
Platform team
Self-serve data platform

Figure 1.3: Operating model of data mesh principles. Adapted from Dehghani, 2022 [4].

1.2.1 Domain Ownership

As outlined before, Data Mesh proposes a division of the organization into domain teams that are parallel to the data transformation process, thus being accountable for the whole end-to-end data lifecycle and having ownership of the whole process. In this context, ownership refers to the responsibilities to create, model, maintain, evolve, and share data as a product to meet the needs of data users [4]. Ownership thus means End-to-end data life cycle, quality responsibility, access control, and security, ensuring compliance with standards and regulations, as well as metadata management which is necessary for a successful Data Mesh integration.

The effective shift-left of the ownership closer to the source of the data achieves more control over the data with the benefits of quality, scaling out, optimization for continuous change, agility in the process by reducing bottlenecks, and cross-team synchronization. It also improves the truthfulness and accuracy of the data by closing the gap between the production and consumption of data, with improved resiliency of analytics. This is key since Data Warehouses or Data Lakes would rely on the centralized team in charge of the consolidation of data, being responsible for the transformation of the data and the whole pipeline in general, overloading the team with responsibilities and spreading thin the knowledge about the stored data. Data marts may have given a small division of representation of the data loosely compared to a data product, but the ownership of them still belongs to the data warehouse team.

This shift into decentralized teams helps improve the approach to data compared to the monolithic architectures, providing enhancement from the two sides of the production of information: The first, from the workers and systems that produce Operational Data and were not aware of the process of Data Warehouse and Data Lake, being disengaged in the creation of added value but that now are part of the mesh. The second, the disappearance of the single, centralized team that ingests the data into the warehouse or the lake, which previously needed to struggle with data quality based on the requirements given to them by all of the different downstream consumers, and that could also be heterogeneous among themselves. By decentralizing the domain teams, these won't need to be aware of all the business domains and change management driven by the source that inevitably slowed down the process, and eventually gave distrust to the data consumers that wouldn't rely on the availability and quality of data.

By creating data domains based on the decomposition of business domains that organizations today perform, Data Mesh pushes this decomposition all the way down to the data and its ownership. This ownership will now possess a well-defined boundary. To identify this boundary an approach of Domain Driven Design is used, where these business domains are first identified and then based on these, map the boundaries to the data, creating the data domains and having the business act as the underlying structure. This results in data domains holding datasets with the same lifecycle and high cohesion among them, facilitating ease of use and atomic integrity.

1.2.2 Data as a Product

Creating data domains enhances the care given to data, treating it now as a product. On a data mesh, handling data as a product carries with it the same attributes as a physical product manufactured by a company. It needs to delight the end customer, provide quality and measure success, be advertised and made available on different channels or representations, and so on. According to Marty Cagan [7], a product needs to be *feasible*, *valuable* and *usable*. Thus, a Data Product must be a well-defined entity that encompasses these attributes by itself. This means having the responsibility of creating and transforming quality data, pushing it upstream on the pipeline, and effectively inverting the model of responsibility compared to a Data Warehouse or Lake. All the processes that are needed to build, expose, and maintain the data must be inside the data product.

Following the same considerations, a data product must also be discoverable, exposing all its information by itself (metadata, schema, syntax, etc.) and thus being self-described. A central catalog is usually leveraged for this purpose but is not strictly necessary. Furthermore, the data product also needs to be addressable, understandable, trustworthy (encompassing attributes of quality, transparency, completeness, etc.), natively accessible, interoperable with the rest of the mesh and the exterior, valuable on its own, and secure, offering measures of access control. The data it holds is usually denormalized, providing relevance by itself and not having the need to consume other sources in order to extract knowledge.

Data products should be designed to have the ability to expose information in many formats, allowing a polyglot approach. Nonetheless, this must be standardized to allow only certain types of exposure to a certain customer but to allow the possibility of having multiple representations of the same data in different formats. The data products are read-only, as the only entity allowed to write in the storage is the data product itself. To expose the produced information and its respective metadata, the data product uses what is called *Output Ports*, acting as interfaces or points of contact from which the consumers can consume the data, but also learn about it thanks to the exposed metadata, offering as well properties of security by access control lists to limit the access to the information.

Other ports that may be present on a data product and are worth noting are Observability Ports, which expose information used to for example measure service level objectives (SLOs), like information about timeliness, interval of change, completeness, shape of data, lineage, etc. This allows consumers to read it and gain trust in the offered data. By including this port along with others, like control ports to configure data governance policies on a data product, the Data Product achieves the accountability that a product should have.

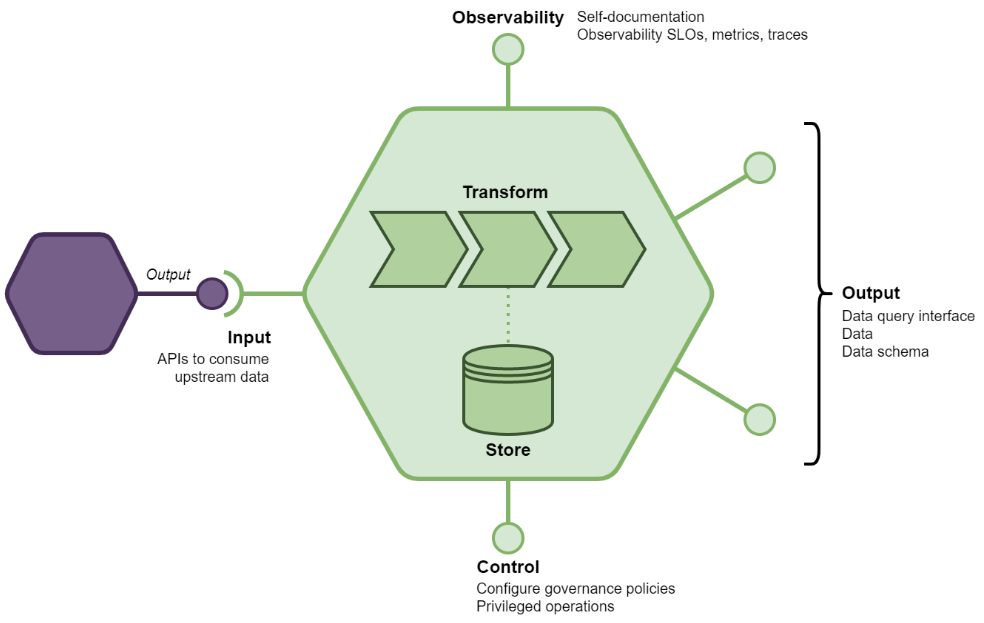


Figure 1.4: A unit of architecture for a Data (Product) Quantum. Adapted from Dehghani, 2022 [4].

The inclusion of the necessary ports to the data product creates a unit of architecture that can be independently deployed, containing all necessary elements for it to function and which Dehghani refers to as a Data Quantum. A high-level view of the unit of architecture for a Data Quantum can be seen in Figure 1.4, but for the purpose of this explanation, the concepts “Data Product” and “Data Quantum” can be used interchangeably. The domain teams that design and deploy data products must work to carry through these requirements in order to have a successful Data Product.

1.2.3 Self-Serve Data Platform

By creating a mesh of data products, the requirement for a centralized platform doesn’t disappear but rather changes purpose. It passes from handling the whole data lifecycle and storage to the creation of a data-sharing platform that manages the full lifecycle of individual data products, empowering domains’ cross-functional teams to share information, helping to use it from source to consumption, and effectively creating a reliable mesh rather than independent data products that live by themselves. It eases both the domain teams and end users to discover, access and use data products.

The main benefits of having the platform that ties the mesh together are, according to Dehghani [4], the reduction of the cost of decentralized ownership of data; the abstraction of data management complexity, facilitating a load of domain teams to manage the data product lifecycle; the reduction of the need of specialization helping to open the data product development to more developers; and the automation of governance policies.

1.2.4 Federated Computational Governance

Decentralizing the management of data introduces several new issues, some of them solved by the self-serve data platform, like discoverability. But even if data products interconnect and aim to offer data with high standards of quality, it's still necessary to balance the autonomy and agility of domains with the global interoperability of the mesh. The Federated Computational Governance principle proposes an operating model that creates an incentive and accountability structure for this. It is a mechanism that ensures the system is secure, trusted, and that delivers value through the interaction of the parts by addressing global issues, such as interoperability, security, compliance, etc. The responsibility of modeling the meaning of the quality of data still belongs to the domain, but automating computation policies are added on top and enforced on the platform that assures the data is secure, compliant, of quality, and usable [4].

Federated computational governance represents the evolution step after the previous manual intervention and central processes of data validation and certification that don't scale quickly and offer minimal support for change. Even though most of the responsibilities of these validations are delegated to the domain teams that maintain the data products, this delegation is partial, since now the need for a centralized and global set of rules that are applied to all data products, with their interfaces and ports appears and it's laid down by a cross-domain team, thus "federated".

By allowing the inclusion of policies embedded as-a-code, data mesh reduces governance coordination friction among domains. The manual certification and validation of policies by the federated governance team can now be automated in the continuous delivery (CD) of data products. In this way, it is a practice that creates global computation policies, arising from concerns on the mesh, defining standards or policies to manage and secure the mesh with a team composed mainly of domain data owners, rather than having general data experts.

The definition of these policies, both at the federated level with global policies and at the domain level with the specification of Output and Observability ports, rely on what the mesh and organization, through a team composed by members of the different business domains, decide are the requirements for it to function correctly and generate knowledge, as this is the goal regardless of the actual implementation. These requirements are usually based on mesh homogeneity for interoperability, data quality for the correctness of the processes and outputs of the platform, and other business and compliance requirements.

1.3 Data Quality

As briefly mentioned before, data is valuable when it's possible to extract meaningful knowledge from it that ultimately helps drive business decisions and provide insights and new services. The degree to which data satisfies these requirements of its intended purpose is what is known as Data Quality (DQ) [1]. This definition, as general as it is, requests a way to delineate this level of "satisfaction" desired from data in order to build processes upon it, and effectively generate valuable knowledge. A model to assess the level of quality of a system or process has been delineated under the standard ISO/IEC 25102 [1]. This standard proposes that organizations must define quality goals that they expect their data to have based on a set of analyses under a set of quality dimensions. However, as Jarke et al. explain in [8], the outlining of these goals is characterized by two issues:

- **Quality is subjective**: Quality goals must be organized according to the stakeholder groups, taking into account research results in data and software quality.
- **Quality goals are diverse in nature**: These goals can be neither assessed nor achieved directly but require complex measurement, prediction, and design techniques, often in the form of an interactive process.

For this reason, even with standards to define data quality in an organization, the ultimate technical aspects of what to define as quality may only have meaning within the organization that sets them. Moreover, according to Golfarelli and Rizzi [3], the corporate of an organization also plays a fundamental role in reaching data quality, as they can create a role hierarchy where only a limited group of users are in charge of data, and these people must be chosen by an appropriate and accurate certification system (that can be specific for every enterprise area). The centralization of these decisions carries possible setbacks that can cause bottlenecks in the creation of knowledge, as already discussed in Section 1.2, so it always needs to be assessed with care.

1.3.1 Data Quality Model

In order to measure, ensure, and ultimately enforce quality on the data, it's necessary to define a measuring frame based on properties or dimensions. As already stated, quality is subjective, so several proposals exist as to what these dimensions should be, all using ISO/IEC 25012 as a baseline. This standard proposes a model to measure data quality based on dimensions that data and the systems used to manage it should hold. These dimensions are:

- Inherent: They're intrinsic to the data and the values it contains
  - Completeness
  - Accuracy
  - Consistency
  - Currency
  - Credibility
- Inherent and system dependent: They're based both on the intrinsic data and the system that stores and exposes it
  - Precision
  - Understandability
  - Accessibility
  - Compliance
  - Efficiency
  - Traceability
- System dependent: They're mainly based on the system that stores and exposes the data
  - Availability
  - Portability
  - Recoverability

The standard provides a wide baseline for the identification of data quality dimensions, but in modern systems where these quality checks are actually used and implemented, usually a subset of them is actively monitored while ignoring some of the dimensions, or passively providing them by the underlying technology guarantees. For example, the Data Management Association (DAMA) of the United Kingdom, a community of data professionals that acts as consultants for the country's government, proposes the following six dimensions as the core data quality dimensions [9]:

- **Completeness**: Are all datasets and data items recorded?
- **Consistency**: Can we match the data set across data stores?
- **Uniqueness**: Is there a single view of the data set?

- **Validity**: Does the data match the rules?
- **Accuracy**: Does the data reflect the data set?
- **Timeliness**: Is data representing the reality of the required point in time?

These core dimensions that DAMA proposes include other measures not included in the standard ISO, like uniqueness and timeliness. Indeed, the decision of which frame to use to measure quality data is effectively chosen according to the business requirements and how the organization works.

Some examples of data quality checks that occur frequently in practice on datasets might span from null-checking (checking that a certain dataset or column contains at most $X\%$ of null values or not at all), the presence of only a closed and known set of values for a certain attribute, number ranges that a numeric column must follow, etc. Depending on the granularity of the evaluation and its specificity, more or less complex checks can be performed. However, as already stated several times, this is dependent on the business environment and the domain the data lives in.

Some consider these metrics as part of a bigger set of attributes that Dehghani [4] calls Data Guarantees. These define whether the data meets the guarantees that their particular use case requires, not only on objective measures of data quality, but also levels on levels of maturity, conformance to standards, temporal characteristics, and so on.

Briefly, the categories of guarantees that Dehghani proposes are: Data Quality metrics which encompass the dimensions already discussed, the Data Maturity metrics which, in contrast to data quality metrics, indicate the point where an organization is located in their roadmap towards a data-driven operating model by measuring properties like degree of usage, diversity, evolving lifecycle, etc. Dehghani also considers the guarantee of Data standards conformance, like industry standards for external interoperability, or internal ones for interoperability inside the organization, as well as legal standards on data. Lastly, the author defines temporality metrics that illustrate the temporal shape of the data, assessing the suitability and availability of data over time.

1.3.2 Data Quality Tools

Since Data Quality is fundamental to the implementation of pipelines for data-driven organizations, quality checks should be created and applied to the desired data. Desirably, these validations should be done in an automatic manner attached to the different steps of a pipeline, executed both at the source level and wherever the data is transformed in order to ensure a high user-based standard of quality.

In a typical ELT pipeline, for example, quality checks should be run at the Load stage, after the ingestion, to check if the source data contains bad data; and at the Transform stage, to check if the transformation hindered the quality of the data as well.

Considering the discussed importance of quality, is not a surprise that an infinity of tools have been created and are used for checking data quality at various dimensions and stages in the typical lifecycle of data. Some of these tools will now be explained.

Great Expectations

Great Expectations (GX) is a tool for the validation of data to ensure user-defined standards of quality while providing as well mechanisms for documenting and profiling the analyzed data [10]. The tool is based on *Expectations*, which are defined by the tool as "verifiable assertions about source data". An expectation provides a high-level declarative abstraction of a quality rule that a user would want to verify on a Data Source, which in turn is an abstraction of a specific set of stored information or data.

Thus, by abstracting both the process to enforce quality rules and the access to the information, Great Expectations provides a flexible tool that adapts to the specifics of a data architecture. This way, the data analyst or engineer needs only to define the set of Expectations (or Expectation Suite) and the Data Source connector to access the information, and the tool will manage the language or technology-specific configurations to act on said data. Moreover, by providing an extensible Expectation interface, custom validation rules can be created for domain-specific validations, or published on the Expectation Gallery, a public marketplace where contributions are made by members of the tool's community.

Architecturally speaking (as seen in Figure 1.5), Great Expectations works under a core component called Data Context, which stores metadata about a specific project and contains the information about the defined Expectation Suites, Data Sources, documentation, etc. that belong to said project, and being themselves the other core components of the tool. Furthermore, the Data Context serves as the entry point for the tool's API, serving the requests and managing the internal state and workflow of a validation, storing results, and interfacing with alerting or notification services.

Using the Data Context it's possible to create Data Sources which configure the physical location of the data to be analyzed. Through this abstraction, it's possible to read data located not only on a local file system but also from cloud storage, streams, etc. From these sources, batches of data can be configured using

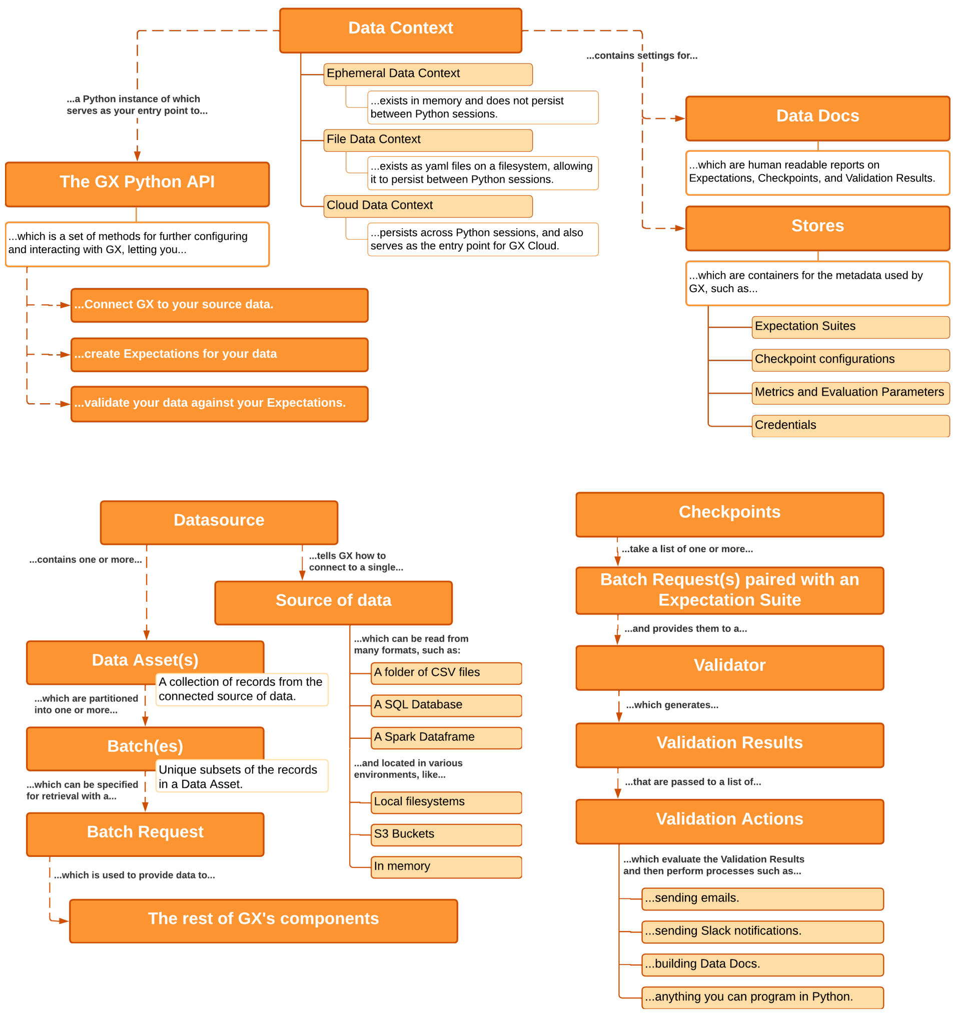

Data Context
...a Python instance of which serves as your entry point to...
...contains settings for...
Ephemeral Data Context
...exists in memory and does not persist between Python sessions.
File Data Context
...exists as yaml files on a filesystem, allowing it to persist between Python sessions.
Cloud Data Context
...persists across Python sessions, and also serves as the entry point for GX Cloud.
The GX Python API
...which is a set of methods for further configuring and interacting with GX, letting you...
...Connect GX to your source data.
...create Expectations for your data
...validate your data against your Expectations.
Data Docs
...which are human readable reports on Expectations, Checkpoints, and Validation Results.
Stores
...which are containers for the metadata used by GX, such as...
Expectation Suites
Checkpoint configurations
Metrics and Evaluation Parameters
Credentials
Datasource
...contains one or more...
...tells GX how to connect to a single...
Source of data
...which can be read from many formats, such as:
A folder of CSV files
A SQL Database
A Spark Dataframe
...and located in various environments, like...
Local filesystems
S3 Buckets
In memory
Data Asset(s)
A collection of records from the connected source of data.
...which are partitioned into one or more...
Batch(es)
Unique subsets of the records in a Data Asset.
...which can be specified for retrieval with a...
Batch Request
...which is used to provide data to...
The rest of GX's components
Checkpoints
...take a list of one or more...
Batch Request(s) paired with an Expectation Suite
...and provides them to a...
Validator
...which generates...
Validation Results
...that are passed to a list of...
Validation Actions
...which evaluate the Validation Results and then perform processes such as...
...sending emails.
...sending Slack notifications.
...building Data Docs.
...anything you can program in Python.

Figure 1.5: Great Expectations core components. Taken from [10]

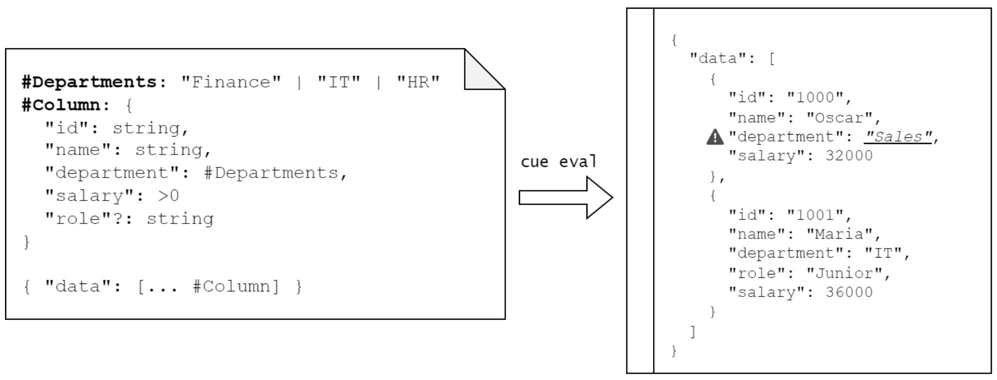
the Batch Request object, and finally, Checkpoints can be built to execute and save validations and their results. By querying the outcome of a Checkpoint, it's possible to know all the details about the execution of a suite of expectations and their outcome.

By using this tool, it is possible to define validations both manually and, more especially, in a computational manner so that these quality rules can be applied to any set of information, receiving detailed reports or alerts that inform the user about its outcome, and allowing the data ingestion process to stop as soon as a quality inconsistency or issue arise.

CUElang

In the context of validation, it is possible to find tools like CUE (also known as CUElang) which take a different approach to validation. CUE is an open-source language working on top of Golang, created as a tool for defining, generating, and validating data [11]. It has a base on logical programming languages (Prolog), and thus it allows the creation of type definitions that work as values. In fact, inside this programming language, types and values are merged into a single concept and this gives the potential to perform data validation based on an expected shape or structure, while also enforcing value constraints.

CUE is not a general-purpose programming language, since its main objective lies in data and object schema validation; and although it doesn't offer more sophisticated validation checks like Great Expectations, it has a strong type definition mechanism to validate data schemas and configurations. An example of a CUE script defining a schema can be seen in Figure 1.6, where it is possible to appreciate the different levels of constraints that CUE offers.



```
#Departments: "Finance" | "IT" | "HR"
#Column: {
  "id": string,
  "name": string,
  "department": #Departments,
  "salary": >0
  "role"?: string
}

{ "data": [... #Column] }
```

```
{
  "data": [
    {
      "id": "1000",
      "name": "Oscar",
    ⚠ "department": "Sales",
      "salary": 32000
    },
    {
      "id": "1001",
      "name": "Maria",
      "department": "IT",
      "role": "Junior",
      "salary": 36000
    }
  ]
}
```

Figure 1.6: CUElang example for a schema definition and validation. By constraining the allowed Department name values, it's possible to easily identify wrong data.

On AgileLab, CUE is especially used for schema definitions and validations for YAML objects and configuration files.

1.4 Data Contract

As it has been explained, the importance of the quality of the data being used by organizations can't be overlooked, and this has led to the integration of quality checks in the steps of a typical data pipeline. Furthermore, new controls on the data and how it's handled by the involved users have been proposed, among which the concept of a Data Contract, a tool to define beforehand the expectations and promises both producer and consumer of the data must follow and comply.

1.4.1 The problem with Data Quality

To understand why it has become necessary to establish contracts on data and its attributes, it's crucial to take a look at the current status of the insertion of data quality checks in a typical pipeline.

Nowadays, when a consumer ingests data, it needs to implement quality checks as part of the pipeline, sometimes on several points of the process. These redundant checks are being done not only for thoroughness but also because in general data tends to degrade as it passes through the pipeline transformations. This degradation happens not only because of faulty transformations or generalizations but especially since the quality of the data on the source determines the potential maximum quality of a pipeline. We can say that the quality of the source is the asymptotic maximum value for the quality that data can achieve, and this is considered uncontrollable as it lies outside the data analyst's responsibility reach (or in the case of a data product, outside of the data product team ownership). It becomes difficult to maintain data quality, as sources constantly evolve, making it hard to trust even if the source was considered trustworthy at an earlier point in time.

As the quality of the output of a data transformation or pipeline is determined by the quality of its inputs, it's nearly impossible to extract knowledge from information that has been deemed as garbage from the start, even on high-quality pipelines where the ETL process has been carefully designed to account for possible different degrees of quality. In the data engineering practice, this is coined as the "Garbage In / Garbage Out" principle (GIGO). As obvious and known as this issue may seem in the practice, data engineers continue to struggle with poor quality data [12], where is reported that 31% of revenue might be impacted by poor data quality, and in 74% of occasions, data consumers are the ones to identify the data issues, downstream on the pipeline [13].

Some of the roots of this problem can be found in the separation of concerns that is established in architectures like Data Warehouses or Data Lakes, previously mentioned in Section 1.1. The layer separation of these architectures has abstracted the engineering team that handles the creation of operational data, and the platform team that manages the centralized warehouse or lake from the processes that data scientists actually perform on the data. Operational teams are only focused on building software and architectures to handle the day-to-day operations, and platform teams are given the overwhelming responsibility to understand the bulk of data the business is producing.

Furthermore, the concept of quality might also differ on each analyst's use case, and can also be understood differently on each side. What might be considered as good data for one team, could be garbage to the other. Thus, it becomes complex for teams to hold knowledge of the different domains the data is being used on, and being unaware of this, they lack the motivation to achieve certain standards of quality (not only in the general context of a dataset but rather at more specific levels where the detailed requirements and nuances of quality might get easily lost). The cognitive struggle to work on a beginning-to-end quality consistency increases, and fails to build an incentive to produce quality data.

Data Mesh proposes a new approach by treating data as a product and enforcing governance on it, ensuring that teams tender for the quality of the data they're exposing and selling. However, both on Data Mesh and other data architectures, enforcement of these quality properties is poorly handled, scarce, or non-existent. Data schemas are treated as non-consensual Application Programming Interfaces (APIs) that could easily mutate rendering downstream pipelines useless. This was identified by Chad Anderson [12], where he explains that:

> While connecting ELT/CDC tools to a production database seems like a great idea to easily and quickly load important data, this inevitably treats a database schema as a *non-consensual API.* Engineers often never agreed (or even want) to provide this data to consumers. Without that agreement in place, the data becomes untrustworthy. Engineers want to be free to change their data to serve whatever operational use cases they need. No warning is given because the engineer *doesn't know that a warning* ***should*** *be given or why.*

1.4.2 The concept of a Data Contract

A solution that has emerged in the last years to solve the problem of data quality is based on the concept of "Data Contract", an agreement between data producer and consumer that has roots in data licenses or service contracts for other kinds of services. Here, both parties agree beforehand on an established set of expectations and promises about the schema and quality of the information to be produced by one and ingested by the other, with an important focus on making this an automatic and auditable process that can be integrated into the pipeline as upstream as possible, hopefully directly at ingestion time, allowing for problems to be identified before the data is used on any analysis.

The overview purpose of a data contract can be seen summarized in Figure 1.7. By introducing a data contract, it is possible not only to set a level of quality a producer must provide but also to introduce enforcement before the transformation pipeline. In this way, early errors that would've been irrecoverable or costly to fix downstream on the pipeline can be automatically identified and the related data rejected. By this means, the quality of the input in the consumer pipeline is guaranteed and consequently, its output has the potential to hold higher value.

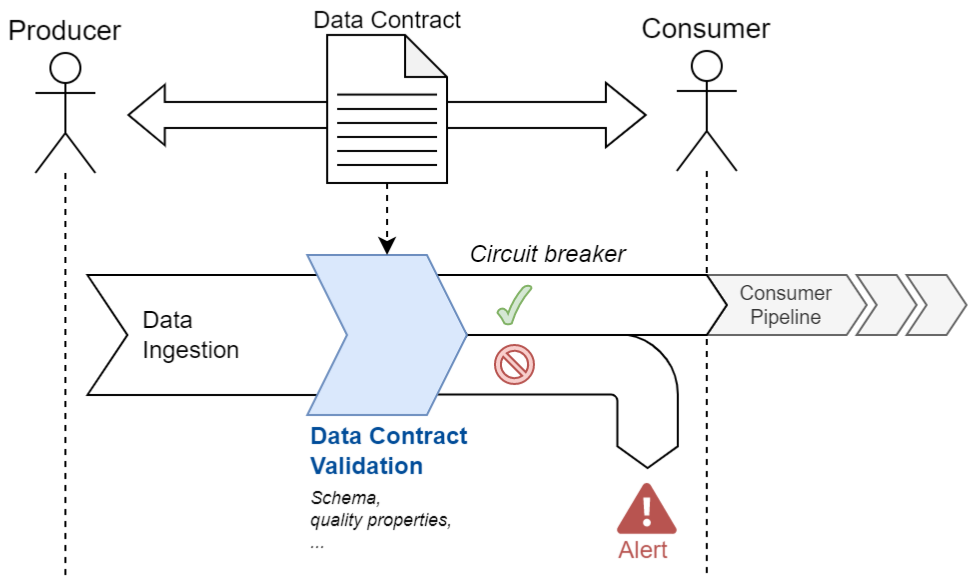


Figure 1.7: Data contract as the mediator of data ingestion.

As Data Contracts administer the communication between producer and consumer, the definition of these two actors will impact how and where the management and enforcement of the contracts shall be done. Thus, it's necessary to approach the definition of these two participants in the context of data ingestion and contract validation.

In general, there are two instances that are tackled by the design of data contracts, based on the control of the ownership of the data source (the producer). If the ownership lies outside the organization, be it an external data provider or any other type of source that falls outside the organization's data architecture or the related data domain, the contract will probably be agreed upon at a business level. Although both parties are incentivized to monitor and enforce the data stream to be compliant with agreed properties, it is necessary to enforce at the consumer level the data that is being ingested, as the contract, which was defined based on business requirements, may not be actually aligned with what the producer is actually providing and the consumer ingesting. The provided data quality cannot be implicitly trusted, and thus the enforcement must be done at a consumer level.

If the ownership lies inside the organization, the scenario becomes more complex, as it becomes possible to manage at both the producer and consumer level that the contract requirements are being satisfied, and that handled correctly, the output of the process will be of high quality and satisfaction for both parties. The way that this enforcement shall be done depends on the context and the existing data architecture.

For example, in a Data Warehouse architecture, the contract might be agreed between the data engineering team that handles the warehouse and data marts, and the data analysts that ingest the data and build reports, train ML models, etc. It is an internal agreement where agents of both parties make an agreement on the required schema, semantics, and other data quality metrics, but ultimately is the consumers that impose the rules to accept the information provided by the warehouse or data marts, as there is usually a lack of metadata that explains the data quality dimensions of the information present on the warehouse or data marts.

1.4.3 Data Contracts in a Data Mesh

In a Data Mesh platform, the difference between the control of the ownership and the consequent establishment of the data contract is made clearer compared to other data architectures. In fact, inside the mesh, where the ownership is handled entirely inside the different domain teams, contracts that govern the communication between data products can be easily built based on the ports they expose. As data is treated as a product, each self-contained data product provides a set of ports that provide valuable information about the data, its shape, quality, etc. Observability ports are designed to produce data quality metrics, and output ports to expose metadata containing in it the schema, service level agreements (SLAs), etc.

From the available information, it is possible to construct and apply the data contract. In particular, on a Data Mesh, a data contract can be used at two points inside the mesh: On Output ports where the data product is exposing the attributes of the data is producing; and on Input ports of source-aligned data products, which work as the entry point of the mesh. When the data originates inside the mesh, data contracts are created by using the provided observability and metadata exposed by the data product itself, and then the contract is exposed by the data product as part of the Output and Observability ports as shown in Figure 1.8, so monitoring of the produced data and its quality can be easily performed in a pull like fashion by querying the ports of the data product to be validated. Furthermore, the versioning of the data contract is tied to the data product one, and the enforcement of supporting different versions of the contract falls directly on the management of the lifecycle of the data product.

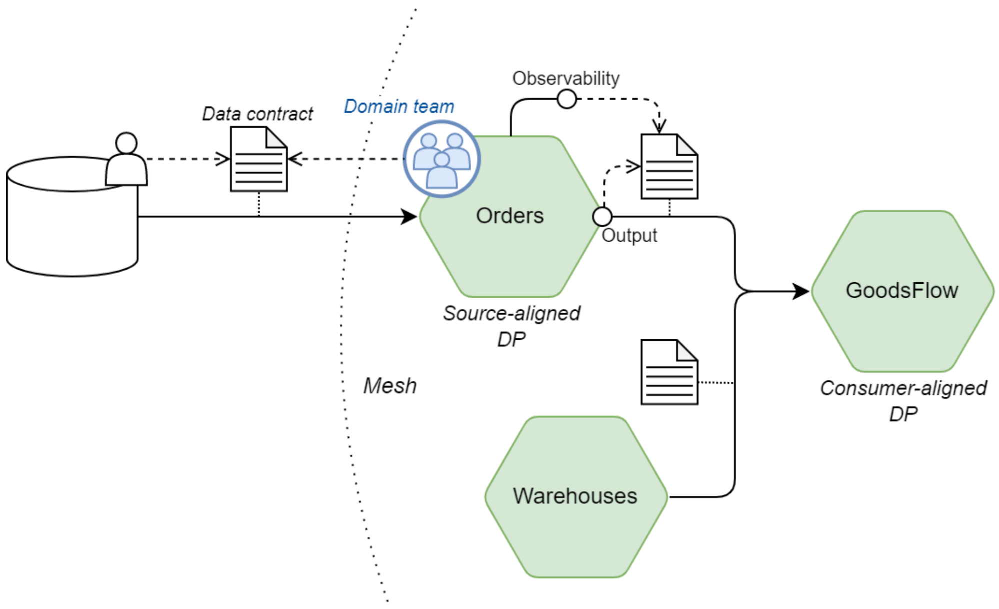


Figure 1.8: Data contracts on a Data Mesh architecture. Source-aligned data products create data contracts by agreeing with a data producer on quality properties. Inside the mesh, contracts can be automatically built and enforced using data product ports.

A slightly different situation is verified on the edge of the mesh, on source-aligned data products that consume data from external providers outside of the mesh. Here, the contract is defined on the Input port and must be constructed and agreed upon between the external provider and the related domain team (see Figure 1.8). Moreover, as explained before, in these occasions where data is coming from an external source and ownership is not held by the domain team, the enforcement of the contract lies on the consumer and thus the data product team should apply the appropriate controls to the input of the data product to ensure compliance checks are in place.

Furthermore, source-aligned data products in their role of points of contact of the mesh with the exterior become the source of the asymptotic maximum of the quality on the mesh, as the value of the information that will be produced and consumed inside of the organization environment will be capped by the quality at the source (GIGO principle). For this reason, the implementation and enforcement of data contracts on this point of the architecture is considered to be of the highest priority.

1.4.4 Data Contract implementation

By acknowledging the importance of a data contract in modern data architecture environments, the need to implement the mechanisms to define and enforce said contracts arises. The implementation of a data contract is a fuzzy task that hasn't been standardized yet, mainly since the concept is recent and it's still in the initial adoption process by companies. It is not only the formalization of a structure that encloses the information of the contract (data quality properties, data schema, service levels, pricing, etc.) but also the utilization of the formal contract to actually mediate the consumption of data, enforcing the agreed specifics of the data.

Early examples of structures called "data contracts" and used for the purpose of mediating between producers and consumers can be found as early as 2007 as part of the architecture of Windows Communication Foundation (WCF) [14] for handling the communication of service-oriented applications in the operating system. However, the domain of application for this concept was far from the data engineering field.

The concept was later revisited in the following years by other authors like Truong et al. [15] in 2011, as a proposal in a context closer to data management, to act as a tool mediating Data-as-a-Service systems, defining the structure and quality of exposed data based on "terms" and five main categories, although easily extensible to others: Data Rights, Quality of Data, Compliance, Pricing Model, and Control Relationship). Their contracts are defined in Extensible Markup Language (XML), Resource Description Framework (RDF), or JavaScript Object Notation (JSON).

More modern examples of Data Contract structure definitions have been proposed both by organizations that handle large amounts of data and by data quality tools as part of the features they offer, following the common practices of the data engineering practice in the current years. Agile Lab itself, for example, has defined a basic specification for a Data Contract as part of the Data Product specification created by the company and made open-source in 2021. The data contract was included in the specification in 2022 and is embedded in the definition of the Output Port, allowing the definition of a schema with basic field checks (nullable, data type, uniqueness), as well as SLAs, basic terms and conditions, pricing, security, and the endpoint from where to access the data. However, most of these fields are free-form plain text and are not defined in a way that automatic processes can be built on top.

Another Data Contract definition is given by Open Data Mesh, an open specification that defines a data product declaration using a JSON or YAML descriptor document, released under Apache 2.0 license and developed by Quantyca s.r.l for a data mesh architecture. Similar to Agile Lab implementation, it defines data contracts as part of the data product specification, but this time as part of a more generalized “Service Agreements” concept which includes three different services called “Promises”, “Expectations” and “Contracts” that are declared on a Control port rather than on the Output port [16].

As defined in their specification, Promises are used to declare the intent of a port in the data product. They are not guarantees of the outcome, but the data product will behave with the goal of realizing their intent. Service APIs, Service Level Objectives (SLO), and deprecation policies are examples of Promises. Expectations on the other hand declare how the data product wants its port to be used by consumers, by declaring the desired audience and usage patterns. Lastly, Contracts describe promises and expectations that must be respected both by themselves and the consumers. It includes terms of conditions, SLA, billing policies, etc. The enforcement of these service agreements is left to the implementers of the architecture to decide, although Open Data Mesh states that for Promises “this verification should be automated by the underlying platform and synthesized in a *trust score* shared with all potential consumers”.

More remarkably, PayPal, an international digital payment company, recently developed and published a data contract format on April 2023 as an open-source format under the Apache 2.0 license [17]. It is defined in YAML and is being used in the implementation of Data Mesh at the company. The version 2.1.1 of the format is composed of several sections shown in Figure 1.9 and defined as follows:

- Fundamentals: Contains general information about the contract, like name, version, support contact, physical access, etc.
- Schema: Describes the dataset that the producer is exposing in the form of a schema. The data quality properties are integrated with the schema at the dataset or column level.
- Data quality: Quality rules and parameters that are included in the dataset. It uses a tool called Elevate to perform the data quality check, although it provides support for other data quality tools.
- Pricing: Describes the way that access to the dataset is billed.
- Service-level agreement (SLA): Describes the SLAs that the producer should provide on the dataset. This enforces properties like latency, retention, time of availability, etc.
- Security & stakeholders: Provides a space to list the stakeholders of the data contract, as well as security measures to control the access to the dataset via the definition of roles and their allowed actions.
- Custom properties: Covers other properties that the contract may include.

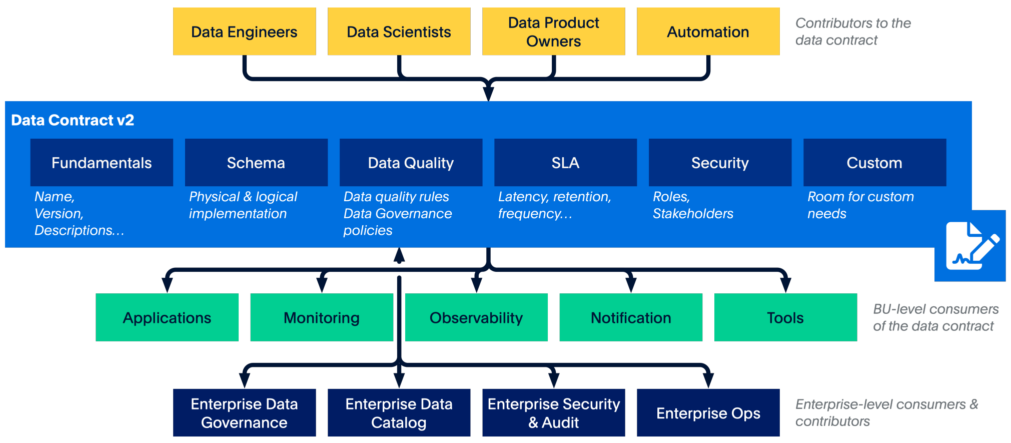

Data Engineers
Data Scientists
Data Product Owners
Automation
Contributors to the data contract
Data Contract v2
Fundamentals
Name, Version, Descriptions...
Schema
Physical & logical implementation
Data Quality
Data quality rules Data Governance policies
SLA
Latency, retention, frequency...
Security
Roles, Stakeholders
Custom
Room for custom needs
Applications
Monitoring
Observability
Notification
Tools
BU-level consumers of the data contract
Enterprise Data Governance
Enterprise Data Catalog
Enterprise Security & Audit
Enterprise Ops
Enterprise-level consumers & contributors

Figure 1.9: Diagram showing the PayPal data contract format proposal. Taken from [17]

Based on the PayPal data contract format proposal, more data contract formats have been proposed, each of them with slight modifications on how to define schema, quality rules, etc. The AIDA User Group followed shortly by publishing the "Open Data Contract Standard" as a fork of the PayPal data contract format specification [18]. However, PayPal or these groups have yet to offer a tool for enforcing this kind of contracts. PayPal mentions briefly the use of an internal service that uses itself a data quality tool named "Elevate", but no other details were given about these platforms.

Nonetheless, all of the data contract specifications agree and emphasize that any specification should be made in a generic and declarative way, to allow ease of integration into other data quality tool languages and data architectures. The same reasoning can be made to the rest of the format specification, as it shouldn't bring any language-specific or tool-specific declarations that bind it to a technology stack, rendering it technology agnostic and pushing the responsibility of contract enforcement to the data contract user.

Lastly, a more practical but more reduced in scope proposal for a data contract was made by dbt, a tool for defining transformation workflows to centralize analytics code, which on April 2023 released in version 1.5 the possibility to add their own definition of "contracts" to their pipelines [19]. Since dbt uses YAML and Structured Query Language (SQL)-like syntax to build projects that define data transformations, it's possible to build data schema and basic quality checks on top of the data transformations. The platform includes already a well-defined data transformation pipeline and quality validations upon which they built their own concept of contracts.

This way, dbt contracts are introduced as a step in the model transformation pipeline where quality tests are run before a model is built, in order to check for compliance with a well-defined set of constraints at both model and column level of a dataset. This way, instead of running tests on data that has already been pushed downstream and might cause breaking changes, it is run before and the model will fail if this fails. However, as these contract checks are executed before the load and transformation are executed, it can only perform checks on the shape (schema) of the returned dataset, and only on constraints that can be applied to the columns at the platform or database level (where most platform only enforce simple constraints like `NOT NULL`, or `PRIMARY|FOREIGN KEY`).

The fact that several proposals for data contracts have been created in a short span of time shows the acknowledgment by the community of the importance of said contracts, but the actual enforcement of said contracts and their agreements is still not present in any tool. Although there are several potential mechanisms to enforce data contracts, there is no one agreement or standard, especially since the concept has started to be taken into consideration only in recent years, and the solutions are business-oriented, following closely the technology stack that an organization might be working with, or by the data engineering most popular or used frameworks. For this, the design of a data contract enforcement system must be designed having in mind the context of the organization, and the products they might be using for data management, especially on data governance and observability.

1.5 Witboost

When discussing data engineering architectures, it is important to mention the platforms on which these architectures are implemented. On this matter, Witboost is the main product of Agile Lab, designed as a modular platform to productize data under built automated data platforms driven by data governance best practices [20]. It is technology agnostic and its structure helps build data architectures for medium to large enterprises. The most common use of Witboost currently is to facilitate the implementation of a Data Mesh platform on organizations that follow the paradigm or are transitioning into it from more standard approaches. This transition is leveraged thanks to the almost one-to-one correspondence of its main components with the logical principles and architecture of the paradigm. In fact, the product was designed with these principles in mind to leverage the advantages that domain-driven design and data mesh practices provide to the management of data.

Due to the microservice architecture of Witboost, it is an extensible platform composed mainly of four components that can be seen in Figure 1.10: A *Data*

Product Builder to leverage templates and blueprints to build automated data pipelines and storage, a *Data Product Marketplace* to make discoverable the data products inside the organization environment enhancing transparency and trust around them, as well as a component for handling federated data governance called *Computational Governance Platform*. Finally, it includes the *Data Product Provisioner* module, which automates end-to-end provisioning and deployment of data initiatives with policy compliance.

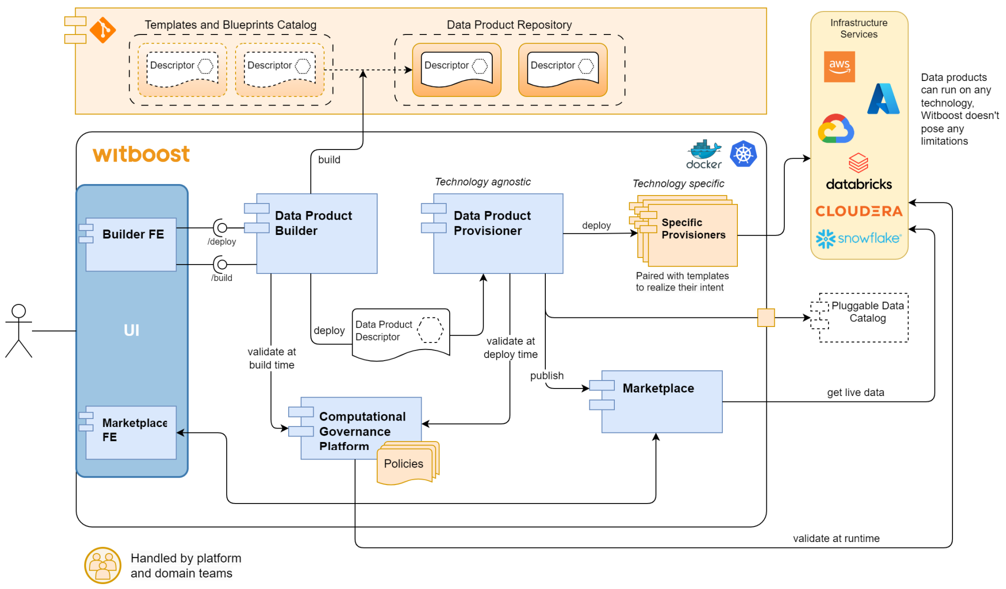


Figure 1.10: Overview of the Witboost architecture showing the main services and components.

Witboost is designed to be entirely technology agnostic, which means that is not tied to a specific stack of technologies but it allows, through the introduction of templates and technology-specific provisioners, to support any technology. Thus, the customer has the ability to include a new multi-cloud strategy or add multiple technologies to an existing Data Mesh implementation. The Data Product Provisioner module orchestrates the deployment of data products by choosing automatically the infrastructure template that completely automates the deployment tasks of the modules that compose a data product, each one implemented by a specific, user-defined technology. These infrastructure templates, also called Specific Provisioners due to their nature of being able to provision a component of a certain specific technology, are implemented as independent microservices either by the customers following an interface specification common to all specific provisioners, or used already built ones made open-source by Agile Lab.

The platform also introduces, through its *Computational Governance Platform*, the ability to enforce standards and quality checks across all resources, defined within the organization by the governance platform team. With this, policy compliance becomes a concrete development step inside the whole lifecycle of a data engineering project, as a way to avoid the degradation of quality in the data products of the organization.

Chapter 2

Problem Specification

Based on the current push on the market to increase quality levels on data transfer, ingestion and transformation, a problem arises for a way to enforce data contracts in a flexible but standard way. Early adopters like PayPal or dbt have created their own solutions to the problem, but they all lack some part of a generally accepted data contract workflow. Paypal open-source project specified a way to declaratively define the necessary user requirements on both schema and data quality rules but lacks an open-source system that uses this format to ingest and control data. dbt on the other hand takes advantage of its declarative schema definition and rule enforcement on top of Database Management Systems (DBMS) features, but this lacks the flexibility to enforce more complex rules and data quality dimensions.

2.1 Scope of this work

Following the previous analysis, it becomes necessary to define a proposal that offers both a rigorous standard for data contracts to define quality rules that are also extensible and flexible, plus a way to enforce them using data quality tools. This last one is heavily dependent on the way data is being sent by the producer, or handled by the consumer, and it may be one of the reasons that no solution is offered off the shelf but they are in all likelihood internally developed and tailored to a specific scenario.

Following this logic, and having in mind the limitations of this thesis, it is necessary to define a scope that is complete enough to show a robust proof of concept, but that should serve as a starting point for future developments. For this, a push-based scenario on the edge of a data mesh has been chosen, where the ownership of data belongs to the producer, and thus is not controllable by the end data user, who is conceptually forced to perform the checks based on the agreed data contract. The data mesh environment is chosen as it is central to Agile Lab culture and strategy, but it should be applicable to other similar environments as well.

2.1.1 Main objectives

1. Create a robust, computable, and flexible data contract format that allows the definition of data quality rules in a declarative way to be applied to datasets.
2. Design and implement a software solution that enforces the data quality rules of a data contract on push-based data ingestions at the edge of a data mesh environment and that can be integrated into Agile Lab's Witboost.

2.2 Project phases

Study case outlining

As source-aligned data products on the data mesh are the point of contact of the mesh with the exterior, is critical to ensure high standards of quality are followed and enforced in order to guarantee high quality output. In these cases, data might be queried or ingested from the external source based on the consumer processes and more importantly, triggered by the consumer on what is called a pull-based scenario. On the other hand, push-based scenarios are triggered by the producer normally in accordance with previous agreements between the producer and consumer for the dispatch of the data at regular intervals or as part of an event-driven pipeline.

More critically, since push-based scenarios are triggered by the producer, is difficult for the consumer to control the flow of data it is receiving and the quality of it. In architectures like Data Warehouse, a stage environment is used as a buffer between the warehouse and the external environment, in which data can be validated and transformed. On Data Mesh, however, there is no centralized stage buffer where to push the information and validate it, as each data product holds the responsibility for its own information. This prohibits the duplication of data, as the ownership would be divided between the data product and the hypothetical stage buffer, meaning that the producer must push directly into the data product storage (although this doesn't necessarily mean that it will be pushing data into the production-ready storage).

However, by internally analyzing the data product, it becomes evident that it could use the appropriate tools needed for validating the received data and have control, albeit limited, of the flow of the data into itself to elevate to a higher level the controls done to the data. Is in these cases that the data product can take advantage of a service that provides the appropriate functionality given the data contract that governs the communication between said data product and a specific data source.

Data contract format specification

After delineating the scope and study case of this work, the first phase consisted in the fulfillment of the first main objective, designing the data contract format specification by analyzing the current proposals in the market, the definitions and implementations of data quality, and the general requirements that it should possess in order to function as the input of a service that implements the required data quality checks based on its contents. For this purpose, a requirements-based analysis and design was performed.

Software solution development

The development of the software service, based on the data contract format and the analyzed study case was implemented after a high-level requirement phase followed by an Agile development lifecycle based on the identified requirements. The software architecture was designed using the hexagonal architecture pattern and based on a declarative workload management state machine which allows for the service to cover more diverse use cases in the future, handling pull-based scenarios, different data formats, streaming, etc.

For the scope of the current work as a proof of concept, the software solution was implemented with the capability of handling push-based scenarios on simple storage objects. It was developed as a hexagonal micro-service architecture, where the management of the Data Contract is done on a Scala micro-service mainly following functional programming guidelines, and the data validation on a Great Expectations Python micro-service.

As this software is intended to work as a minimum viable product for the company, the final development contains the basic workflow to create, manage, and enforce basic data contracts, and perform validation on comma-separated values (`csv`) files stored in local storage.

Chapter 3

Analysis and Implementation

The following chapter will describe the process of analysis, design, and implementation of the specification and software products that realize the main objectives of this thesis. In regards to the software development, it was done following an Agile methodology, starting from a high-level requirements analysis using stories, and then following a process of high-level and low-level design for the software product. Following the iterative approach that is at the core of Agile processes, these two kinds of designs were performed for each major feature to be implemented, rather than an initial large design that would accompany the whole project from start to end. This way, the project could be adjusted during its development process based on the observed desired functionalities, manifested errors, etc. This was especially useful on low-level design features, as the high-level architecture of the software product would not suffer big changes, but the more detailed internal processes defined on the low-level design steps could change swiftly without accumulating technical debt and responding to changes in the decision process.

The following sections explain the analysis, design, and subsequent implementation of the Data Contract format specification, and of the Data Contract Manager with its microservice architecture. The process, as already explained, started from the high-level requirements identified inside the company by communicating with the stakeholders and involved people, to then settle these requirements via a high-level design refined using a low-level design, that finally gets implemented via software and tested against the initial requirements.

3.1 Data Contract Format

Based on the analysis done in Chapter 1 about the current implementation of Data Contracts, it is possible to define a set of high-level requirements in the form of user stories, which are the smallest unit of work in an agile framework and function as an end goal for the developed product. On the data contract format specification scope, the following set of user stories was defined, taking into account stakeholders and the current state of the art.

- As a data consumer, I want to explicitly define the schema of the data I will consume, so that I can ingest it without having issues with versioning, updating queries, or losing relevant information.
- As a data consumer, I want to define data quality metrics that the data I will consume should achieve, so that I can extract valuable knowledge from the data.
- As a data consumer, I want to define service levels for the data that I will consume, so that I can extract valuable knowledge from the data.
- As both a data consumer and data producer, I want the data contract to have a structure from which I can build automatic and computable processes.
- As a data consumer, I want to evolve the data requirements I enforce on the Data Contract based on the improvements of my analyses, so that I can enhance the processes I make to extract knowledge.
- As a data producer, I want to have a clear understanding of the metric values I need to provide so that I don't enter into conflict with my consumers
- As a data producer, I want to define the pricing model of the ingestion of the data, so that I can bill correctly my customers.

From this, and from the current format specifications already in the market, it is possible to define a formal specification of a data contract to be used by Witboost in the context of a Data Mesh. Using PayPal specification as a baseline provides several benefits since it provides a well-established format that covers the desired requirements and is released under an open-source license. However, as this format was defined not to be as generic as possible, but to work on the Data mesh implementation of PayPal and then modified to be open source, it has some incompatibilities that may provide problems when used under the Witboost platform.

First of all, the data quality tool being used by PayPal and mentioned briefly in the specification is an internal tool called Elevate, and the format slightly depends on it and the required rule name needed to execute a specific validation. This way, the data contract loses its declarative approach and depends heavily on the underlying implementation of the software that manages it. It is agreed upon that any data contract format should be able to be enforced by any platform and thus should hold a high level of technology-agnostic features. A more generic and declarative approach can be followed for more general rules that are present in most data quality use cases, but always leaving the specification open for extension for more technology-specific custom rules.

Other incompatibilities between PayPal format and Witboost are less impactful but need to be considered as well, like naming conventions that differ for several fields, and that PayPal specification is also very broad on all the use cases that intend to cover, and many of the fields defined by PayPal are not designed to have automatic computations on top of them, which is essential to a Data Contract enforcement. In the same fashion, some fields already in use on the Agile Lab data contract should be included in order to maintain the same minimum level of information offered by it.

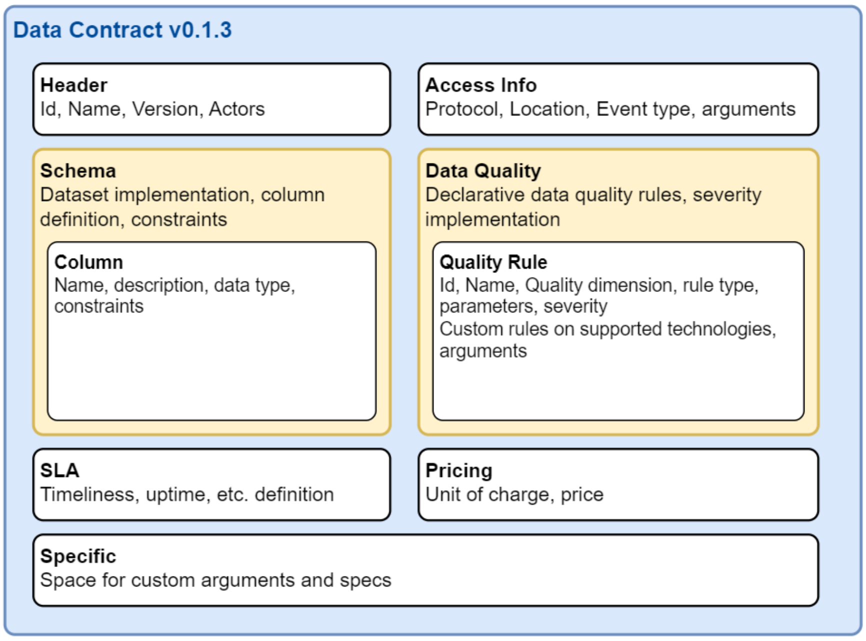

Data Contract v0.1.3
Header
Id, Name, Version, Actors
Access Info
Protocol, Location, Event type, arguments
Schema
Dataset implementation, column definition, constraints
Column
Name, description, data type, constraints
Data Quality
Declarative data quality rules, severity implementation
Quality Rule
Id, Name, Quality dimension, rule type, parameters, severity
Custom rules on supported technologies, arguments
SLA
Timeliness, uptime, etc. definition
Pricing
Unit of charge, price
Specific
Space for custom arguments and specs

Figure 3.1: Example of the proposed Data Contract format

Having all of this in mind, the first version of the proposed contract can be seen in Figure 3.1, and the specification definition for each field, and a corresponding example YAML file can be found in Appendix A. The overall structure of the format takes inspiration from the PayPal specification, but the specific field definitions are modified to fit the aforementioned requirements and incompatibilities. Thus, the proposed sections are the following seven:

1. **General information**: Contains the header of the contract showing metadata about the contract and its versioning, as well as data sharing agreement information.
2. **Access Information**: Defines the way data should be transferred and where. Includes properties to describe the protocol, data location, and security information, as well as extra arguments that would be useful to characterize the data to be read and can be leveraged by future validators to access the correct information.
3. **Schema**: Defines the shape of the dataset, as well as basic constraints on the data columns.
4. **Data Quality**: Declares the quality rules to be met by the information. These are defined in a declarative way following inspiration on a configuration designed by the Analytics Platform Engineering (APE) team at Ibotta [21] for their data quality checks, as well as the format for warning and error definitions used by the tool SodaCL [22]. The quality rules are defined within three categories, following the Data Quality dimensions analyzed in Chapter 1: Completeness, Uniqueness, and Validity rules. Table 3.1 shows the summary of the supported data quality rules. A quality rule can be defined for one or several columns, or none if it is defined at the dataset level.

Category	**Rule**	**Description**	**Threshold check**
Completeness	`size`	Checks the number of rows in the dataset	$=$ `threshold`
	`complete`	Checks the percentage of `NULL` values in a column	$\geq$ `threshold`
Uniqueness	`unique`	Checks the uniqueness of a column	$= 1.0$
Validity	`min`	Checks that the values of a numeric column respect a minimum threshold	$\geq$ `threshold`
	`max`	Checks that the values of a numeric column respect a maximum threshold	$\leq$ `threshold`
	`allowedValues`	Checks that the values of a column belong to a closed set of values	
	`mean`	Checks that the arithmetic mean of a numeric column is equal to the expected value	$=$ `threshold`
	`stdev`	Checks that the standard deviation of a numeric column is equal to the expected value	$=$ `threshold`
	`custom`	Provides fields for a specific data quality rule based on a specific supported `technology` and a quality rule name to `call`, plus the needed parameters for the rule check	

Table 3.1: Supported declarative quality rules on the proposed Data Contract format specification

In addition to the parameter provided by the user, the threshold is modified based on the warning and fail parameters for each rule, as these provide a way to relax the constraints, especially the equalities, where an error tolerance might be acceptable under the business requirements. For this purpose, the warning and failure tolerance are defined as a percent error on the provided user parameter, and control the threshold check, raising an error if and only if the actual values are outside of the acceptability range.

5. **Pricing**: Provides a section to specify the cost of ingesting the provided data based on the data size or the number of data scans. It is important to notice that the enforcement of payment and cost calculations is outside of the scope of the implemented data contract manager micro-service.
6. **Service level agreements (SLAs)**: Defines the values for SLAs that the data producer has to meet. The enforcement of the SLAs is outside the scope of the implemented data contract manager micro-service.
7. **Specifics (Other properties)**: Defines other information that might be specific to the data contract and doesn't fit in any of the other sections.

As Witboost works mainly with YAML files to store component information, is logical to use the same format, as it also offers more flexibility than other file formats like JSON (despite YAML files being prone to syntax errors as it uses indentation as the main nesting tool).

For several fields, the OpenMetadata standard for format specifications is leveraged, as it is widely used in the sector and provides a common format language. Agile Lab already uses several of their schemas on data product specifications, on fields such as dataset schema, tags, and data profiling; so the same structure can be done on the Data Contract, allowing interoperability between components inside the platform.

3.2 Data Contract Manager

After designing the data contract format to declare the agreements between producers and consumers, it becomes necessary to use it to ensure the quality on the data related to this agreement. The contract in itself is already a valuable tool as a metadata annotation on the expected properties of the data to be ingested by a certain producer, but it gains even more value when used to build automatic validation processes on top of the contract definition.

For this reason, it becomes necessary to analyze how a system for this purpose should be created. A Software solution that realizes this purpose can have different forms, and as explained in Section 1.4.4, no one solution is standard. For this purpose, it is necessary to perform an analysis starting with the user requirements to understand the possible design and implementation solutions.

3.2.1 Analysis

1. As a data consumer, I want to validate the data contract format and requirements in an automatic way, so that is compatible with the standard and the requirements have a logical sense.
2. As a data consumer, I want to continuously monitor the ingested data, so that it remains consistent and compliant to the contract.
3. As a data consumer, I want to evaluate the proposed data by a producer before accepting the transfer of it, so that I don't waste resources if the data is not compliant.
4. As a data consumer, I want to receive warnings or alerts when the received data is not compliant with the contract.

Based on the high-level requirements, two initial proposals were evaluated. The first proposal defines a microservice system that manages the data contract transaction, data ingestion workload, and inherent validation based on the agreed contract. The second one is to design a library to be integrated into data ingestion workloads of a specific technology (e.g. a wrapper of a Spark job that validates data while it's being ingested) and that shortcuts the pipeline if any error is identified.

The first option would provide a robust service that not only performs validation, but also keeps a record of the processes being performed, allowing persistence and versioning of data contracts, and being technology-agnostic, thus offering the possibility to be configurable and work under different technologies. However, it would provide a bigger overhead on a data transaction and would require existing pipelines on both the producer and consumer side to be rewritten to include the management and validation of the data contract in them.

The second option would offer a more light-weight, easily integrable solution that can be included in existing pipelines with lower overhead, but with the drawback of being ephemeral, not allowing to store metadata or records about the performed processes, and being tied to the single technology for it was conceived, having to be re-done for each type of data pipeline framework used inside the mesh.

3.2.2 Design

Having these tradeoffs in consideration, the first option was chosen as the high-level solution to be designed and implemented. This was further refined via a high-level design process as a microservice system at the mesh level that will handle the data contracts of the platform. At this level of design, the actual validation process is not defined, but it must take into account the diversity of validations and data that might exist, allowing the platform to be used by the Producer and Consumer to mediate the exchange of data in different scenarios as seen in a high-level infrastructure diagram in Figure 3.2. Both consumer and producer will interface with a Manager that will handle the contract lifecycle and validation processes, interfacing itself with the necessary tools to perform quality checks.

At a mesh level, it is expected for this system to be used as part of the Input ports' functionality of a Data Product to ingest and validate the information ingested by it.

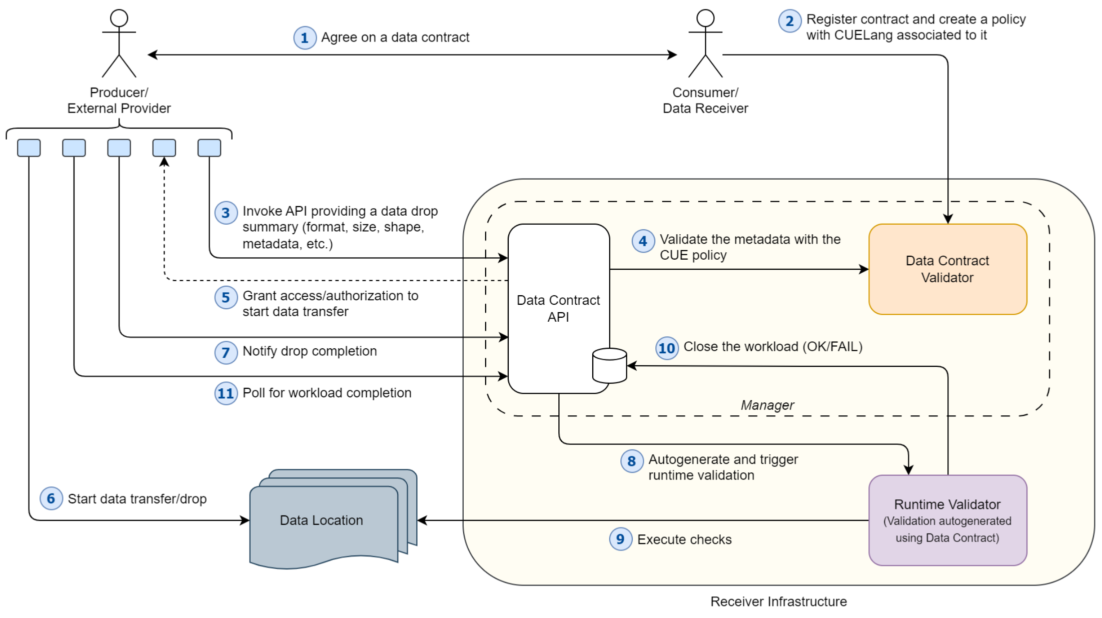
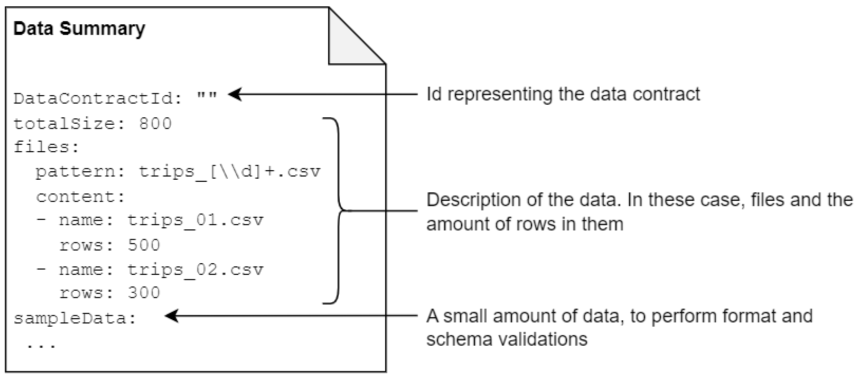

1 Agree on a data contract
2 Register contract and create a policy with CUELang associated to it
Producer/ External Provider
Consumer/ Data Receiver
3 Invoke API providing a data drop summary (format, size, shape, metadata, etc.)
4 Validate the metadata with the CUE policy
Data Contract Validator
5 Grant access/authorization to start data transfer
Data Contract API
7 Notify drop completion
10 Close the workload (OK/FAIL)
11 Poll for workload completion
Manager
8 Autogenerate and trigger runtime validation
6 Start data transfer/drop
Data Location
Runtime Validator (Validation autogenerated using Data Contract)
9 Execute checks
Receiver Infrastructure

Figure 3.2: High-level design infrastructure of the system that will handle data contract lifecycle and enforcement.

The chosen basic scenario for the implementation of the first version of the service would be the ingestion of data initially located outside the mesh on a push-based scenario (Figure 3.2 shows a basic operation flow of this use-case scenario). In this context, the data location is owned by the consumer, and the producer will push information into it. This location will be a type of storage, not in production, that will receive the data and will have its quality and format evaluated based on the registered data contract to verify that is compliant with what was agreed upon. The ownership of said location belongs to the consumer domain, likely inside a Data Product.

Furthermore, it's important to highlight that the high-level design foresees the implementation of a data summary in order to achieve the high-level requirement number three *"As a data consumer, I want to evaluate the proposed data by a producer before accepting the transfer of it so that I don't waste resources if the data is not compliant."*. This data summary would include properties of the data to be sent, like format, size, shape, metadata, and row examples, in order to make an early validation of the proposed data. An initial example of how a data summary could be designed can be seen in Figure 3.3. However, for the scope of this thesis, this requirement was not implemented in the software solution and was left for future improvements of the system to accurately design and integrate it into the validation workflow (where the Manager, as will be seen later, has been designed to be easily extensible and configurable for processes like this one).


Data Summary
Id representing the data contract
Description of the data. In these case, files and the amount of rows in them
A small amount of data, to perform format and schema validations

```
DataContractId: ""
totalSize: 800
files:
  pattern: trips_[\\d]+.csv
  content:
  - name: trips_01.csv
    rows: 500
  - name: trips_02.csv
    rows: 300
sampleData:
 ...
```

Figure 3.3: Small example showing a data summary object, used by producers to send initial metadata about the data to be sent.

Validations in the system will be performed by two different kinds of tools: CUElang and Data Quality tools. These are focused on different aspects of the process and are defined as follows:

- **CUE Validation**: Format and structure validation for the Data Contract object registered by the consumer, plus the data summary sent by the producer and the eventual dataset schema.
- **Runtime Validation**: Data quality validation using a specific data quality tool or framework, like Great Expectations.

Based on the high-level solution, the microservice architecture is defined by a Data Contract Manager that handles the lifecycle of the data contract and the workload of the data validation, leaving the validation itself to a second microservice named Data Contract Validator. Since the validation workload must be flexible, the Manager is designed to handle declarative workloads based on a series of steps that may vary depending on the user requirements. This microservice will expose a defined interface that will allow the integration of different technologies in the future, leaving the Manager technology agnostic and pushing the responsibility to implement the quality checks to the Validator services based on the specifics of said tool.

Through a low-level design process, the Data Contract Manager was designed with a layered architecture composed of three layers: Application, Domain, and Infrastructure layer, following a design pattern called “Ports and Adapters”, also known as “Hexagonal architecture”. This pattern defines domains that encapsulate the business behavior of different units of the application under Ports: Well-defined interfaces that are then implemented by Adapters units. This abstraction helps to seamlessly integrate different technologies by implementing different adapters, allowing to have in future different tool integrations for Data Contract format validations, Data Summary validations, Quality Runtime Validations, and more.

A summary of the three layers architecture can be seen in the diagrams shown in Figures 3.4 and 3.5 where they show the Manager under two abstractions: The first, as system domains, containing the ports they define and the dependencies between them. Five domains compose the Manager, these being the *Data Contract Lifecycle*, the *Data Contract Validation*, the *Data Runtime Validation*, the *Data Transfer* domain, and the core *Workload* domain that handles the contract enforcement. The second image is a Unified Modeling Language (UML) component diagram showing the interfaces and components that compose the domains across the three layers. In the diagram is worth noting the “soft dependencies” between the Workload Manager component and the Workload Operation interfaces, as these are defined only at runtime based on the declarative state machine that defines the data contract workload, which will be explained in further detail later in this document.

Application layer

Handles API requests from producer and consumer actors. Contacts the domain layer for the application logic. The main component is the **Data Contract API**. It receives the request, maps them into data transfer objects, and sends them to the domain layer. It is defined by the API port which in practice is declared as the API interface specification.

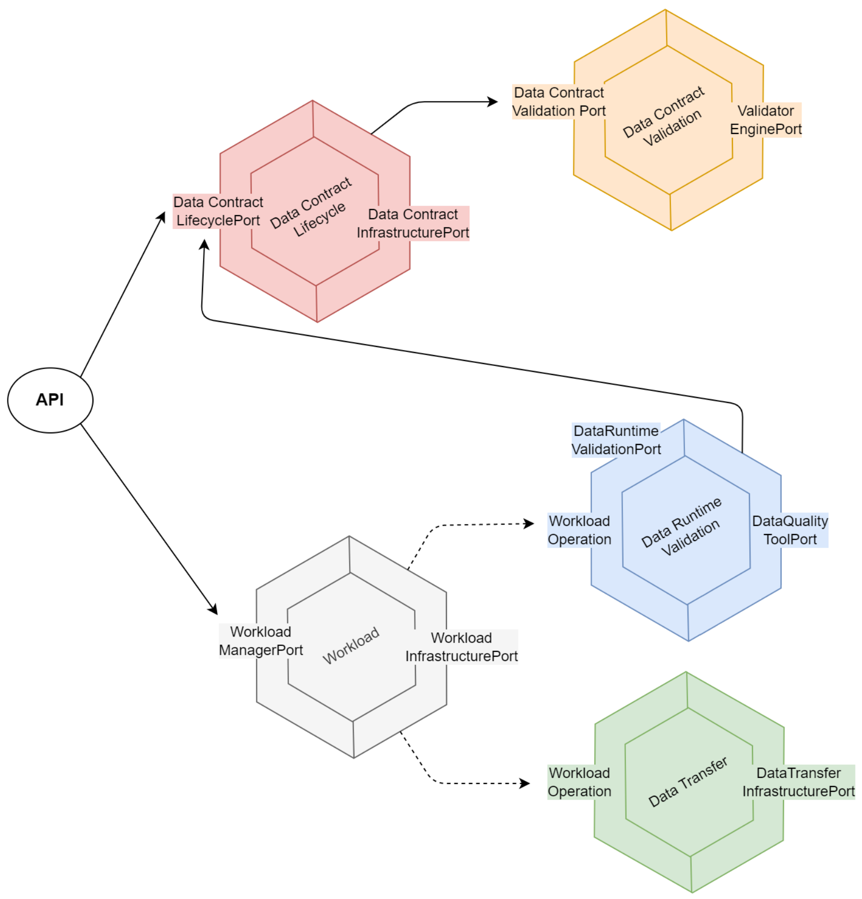


Figure 3.4: Data Contract Manager domain diagram. It shows the relationships between application domains and the ports exposed by each one of them.

Domain Layer

Handles the business logic of the data contract enforcement. Creates the entities, triggers validations in the context of a transaction, and based on the validation evaluates the goodness of the workload result. It is the layer where the "Application" entity of the hexagonal architecture resides. The different Application domains are based on the entities and the responsibilities they handle and are defined as follows:

Data Contract Lifecycle: Domain that is defined by the Data Contract Lifecycle Port and the Data Contract Infrastructure Port. In charge of handling the Creation, Retrieval, Update, and Deletion (CRUD) of Data Contracts. It

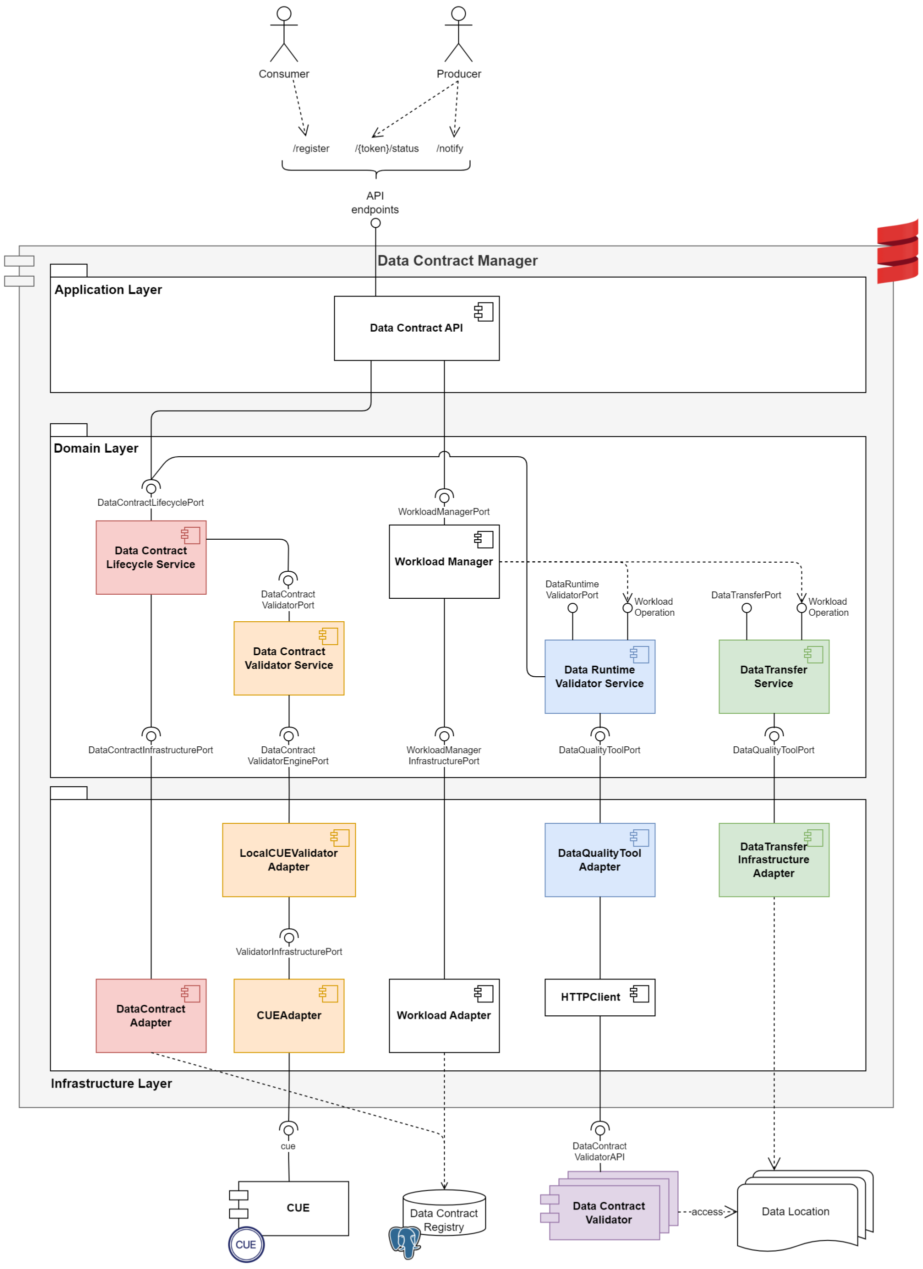


Figure 3.5: Data Contract Manager component diagram. The three layers can be seen from top to bottom: Application, Domain, and Infrastructure. All the domain ports are defined in the Domain layer as per the design pattern, avoiding business logic leaking to other layers.

consumes the port of the Data Contract Validation and is consumed by the Runtime Validation domain to retrieve the contract information and trigger the validations based on it.

Workload: A workload is a unit of work that contains the lifecycle of the intention of pushing data into the mesh and validating it. It can be declaratively defined via a state machine, but in general, starts when the Manager is notified whether by the producer or the consumer that a data ingestion is to be done, and ends when the quality runtime validations have finished. The outcome of the workload (OK/WARNING/NOK) is dependent on the outcome of the quality runtime validations and other checks the Manager might have performed. This domain is defined by the Workload Manager Port and the Workload Infrastructure Port; and is in charge of starting, retrieving, and updating workloads, usually called in an asynchronous way.

Data Contract Validation: Application that encompasses the Data Contract format validations and the Data Summary validations, since both are based on a common format. Data Contract format validation validates the format and contents of a Data Contract with respect to the specification. It autogenerates data summary validations based on the contents of a contract, which is a validation used to validate data summaries belonging to a specific data contract. It is defined by the Data Contract Validator Port, Data Summary Validator Port, and Data Contract Validator Engine Port.

Runtime Validation: A (Quality) Runtime Validation is a validation done on the data that has been pushed by the producer into the consumer's mesh using the appropriate medium provided by the contract. This domain is in charge of handling the business logic of deciding the tools to use to enforce the quality rules using the appropriate port, and further determine the outcome based on the received validation results. It is defined by the Data Runtime Validator Port and the Data Quality Tool Port.

The Data Quality Tool Port is the interface of the tool that takes the application runtime validation and triggers the appropriate validation of the data, handling the communication with the external services.

Data Transfer: Often is necessary to transfer data from locations in order to perform validations. As Validators might only have access to a specific location to read and perform validations, is the responsibility of the Manager to handle the transfer of information between locations. This domain is heavily related to the storage technology, so several adapters that implement the infrastructure port can be defined. The domain is defined by the Data Transfer Port and the Data Infrastructure Port.

Infrastructure Layer

The infrastructure layer contains the access to the registry of information for the service, it contains the Adapter implementations of the Infrastructure Ports needed by the domain application. It also contains the adapters to contact the external tools that may be used, such as the Data Contract Validators for the Runtime Validation, external storage like the file system or cloud storage for data transfer, or other tools like CUElang.

3.2.3 Implementation

As mentioned before, the design of the system is composed of two microservices, the Data Contract Manager for data contract validation and workload management, and the Data Contract Validators for data quality validations. This section will explain the technical aspects of the implementation of the Manager and will go into the details of implementation, both its development process and the resulting system.

The Data Contract Manager has been developed as a microservice in Scala 2.13 following pure functional programming practices, using the libraries `cats` and `cats-effect` to handle Input/Output (IO) and asynchronous operation side effects. The Application layer is implemented using the libraries `http4s` and `tapir` for handling the API endpoints, both as a server receiving requests from consumers and producers, and as a client for contacting the Data Contract Validators when needed. On the infrastructure level, it uses a Docker containerized PostgreSQL database for persisting data contracts, workloads, and their configurations; and to work with the Java Database Connectivity (JDBC) layer to access the database the application uses the library `doobie`.

The exposed API interfaces of the Data Contract Manager have been defined as follows:

- `POST /api/v1/data-contract`: Registers a data contract, performing parsing and validation before accepting it.
  - **Request body**: application/json containing the data contract object as a plain string. For an example of a data contract, see Appendix A.
  - **Response**: `200 OK` (success) with the registered data contract ID, or `400 Bad Request` if a validation error occurred.
  - **Response body example**:
    ```
    "data-contract-finance-dataproduct1-1.0.0"
    ```

- `GET /api/v1/data-contract/{dataContractId}`: Queries a Data Contract based on its unique data contract ID
  - **Request body**: None
  - **Response**: `200 OK` (success) with the data contract, `404 Not Found` if the specified data contract wasn't found, `400 Bad Request` if an error occurred.
  - **Response body example**:
    ```
    {
      "id": "data-contract-finance-dataproduct1-1.0.0",
      "version": "1.0.0",
      "producer": {
        "name": "finance",
        "group": "finance_othercorp.com"
      },
      "consumer": {
        "name": "dataproduct1",
        "group": "dataproduct1_corp.com"
      },
      "content": "..."
    }
    ```
- `POST /api/v1/workload`: Creates a data transfer workload of a specific data contract.
  - **Request body**: application/json containing the data contract ID as a plain string.
  - **Response**: `200 OK` (success) with the created workload and its status, `404 Not Found` if the specified data contract wasn't found, or `400 Bad Request` if a validation error occurred.
  - **Response body example**:
    ```
    {
      "workloadId": "93a324c1-f923-451c-8a88-4d93d7669d60",
      "dataContractId": "data-contract-finance-dataproduct1-1.0.0",
      "status": "Created",
      "result": {
        "result": "OK",
        "info": { ... }
      }
    }
    ```
- `POST /api/v1/workload/notify/{workloadId}`: Resumes a non-Running workload, used to notify that the data is available to validate.
  - **Request body**: None
  - **Response**: `200 OK` (success) with the workload and its current status and result, `404 Not Found` if the specified workload wasn't found, or `400 Bad Request` if a validation error occurred.
  - **Response body example**:
    ```
    {
        "workloadId": "93a324c1-f923-451c-8a88-4d93d7669d60",
    ```

```
        "dataContractId": "data-contract-finance-dataproduct1-1.0.0",
        "status": "RuntimeValidationInProgress",
        "result": {
            "result": "Running",
            "info": null
        }
}
```

- `GET /api/v1/workload/{workloadId}`: Polls the status of a workload using its workload ID
  - **Request body**: None
  - **Response**: `200 OK` (success) with the workload and its current status and result, `404 Not Found` if the specified workload wasn't found, or `400 Bad Request` if a validation error occurred.
  - **Response body example**:

```
{
  "workloadId": "93a324c1-f923-451c-8a88-4d93d7669d60",
  "dataContractId": "data-contract-finance-dataproduct1-1.0.0",
  "status": "Rejected",
  "result": {
    "result": "NOK",
    "info": {
      "dataSummary": "...",
      "errors": [],
      "results": [
        {
          "id": "minimum_fare_amount_rule",
          "column": "fare_amount",
          "metric": -52,
          "severity": {
            "fail": {
              "success": false, "exceptionInfoModel": null
            },
            "warn": {
              "success": false, "exceptionInfoModel": null
            }
          }
        }
      ],
      "success": "fail"
    }
  }
}
```

Workload management

The Workload domain is the core of the Data Contract Manager. In the context of the system, a workload is the concept of the work that has to be done and the result that it produces, living inside a well-defined finite state machine (FSM) that dictates the set of processes a workload has to go through to produce a final result. A low-level design was performed to design said domain, having in mind this definition and with the focus of keeping the generalization of possible workloads and their FSM.

The workload, aside from holding the specific information each performing task needs to work, also holds the current State inside the state machine, dictating the action to be executed and producing an outcome that will determine the transition to trigger and thus next state in the FSM. For this purpose, operations are chained via a step function, which for each state may call an interface `WorkloadOperation` bound for that specific state in the FSM. The work is done by a domain implementing the interface, performing the unit of work, modifying the workload, and returning the outcome so the step function transitions the workload accordingly. The design for the management of workloads can be seen in the class diagram of Figure 3.6.

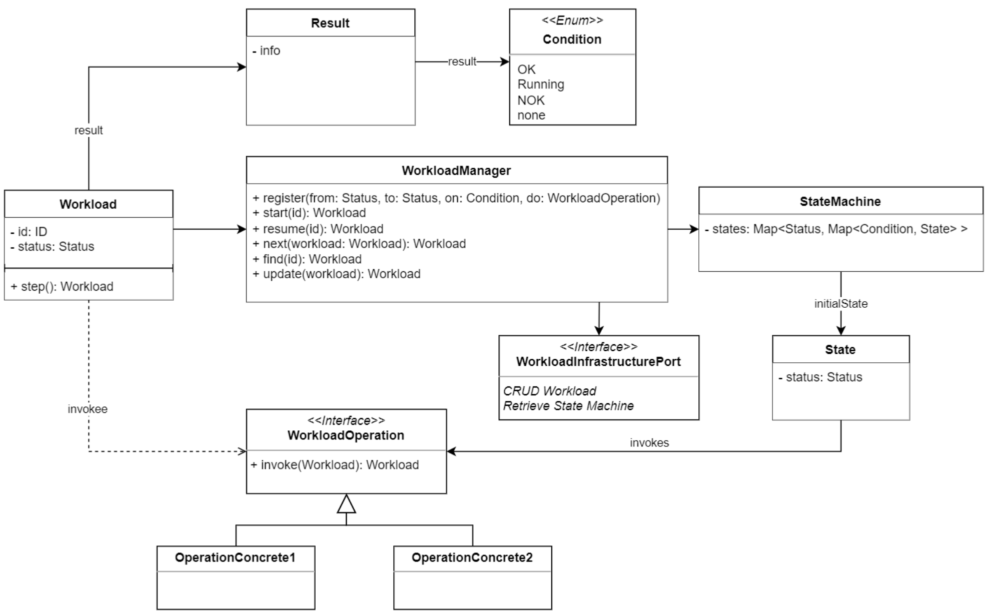


Figure 3.6: Workload Manager class diagram. The Workload Manager handles the Finite State Machine and transitions workloads between states based on the outcome of their operation.

The Workload domain is designed in a way that its components are generic on the realization of a process. On one hand, workload operations are wrapped under the `WorkloadOperation` interface, so it's important that the management is designed to consider operations that could be not only synchronous but asynchronous as well. This introduces a new challenge, as a workload now could either need to wait for a blocking synchronous operation (i.e. an IO operation, or the response of an HTTP call) to advance, or to be resumed after an asynchronous operation finishes. To solve this, the workload is persisted using an infrastructure port and saving the necessary information to resume it at a later point in time (which, in terms of the workload alone, consists of the current state, transition condition, and the current workload information), and to use a callback system on asynchronous calls to resume the workload when needed.

Moreover, to support multiple use cases on the way data contracts are enforced, the Finite State Machine used by the workload Manager has to be both generic and configurable, not ingrained in the software flow, but defined as a set of states and transitions, which in turn can be persisted in a database and read at the startup of the service, allowing for multiple FSMs to exist and to be configured without affecting the codebase with big changes. A State, as seen in Figure 3.6 is based on a *Status* which is just the State name, and the `WorkloadOperation` that must be performed when a workload is present on said status.

Using this information, the workload manager creates workloads, setting them on the initial state, performs the transitions of State based on the Result of the Workload and the registered State Machine, and resumes workloads when needed. For this last one, to resume a specific workload, the manager confronts the state machine with the persisted status of the workload to be resumed. This generalization means that is possible to define in the future multiple FSMs designed to implement diverse use cases using the same infrastructure and cohabiting in the system. The workload manager would possess the responsibility of updating the workload inside the appropriate FSM based on its corresponding type.

Based on this, the following types of transitions can exist in the FSM. Furthermore, an example of how to define it on SQL is given for each one.

Transition types

- **Initial transition**: Transition that executes when a workload is created. Is characterized by a `NULL` in the `status_id` column and a Running value as the `transition_condition`.

  ```
  INSERT INTO workload_machine_state (
    status_id, transition_condition, transition_to_status_id,
  ↪  operationid
  ) VALUES (NULL, 'Running', 1,
  ↪  '_operation_setup_runtimevalidation');
  ```

- **Active transitions**: Transitions that execute an action when they're activated. They contain a not null value in the `operationid` column. When they are activated, the `WorkloadOperation.invoke` method bound to the specified `operationid` value is called. A user can trigger via endpoint a Transition with a Running value set as the `transition_condition`.

  ```
  INSERT INTO workload_machine_state (
  ```

```
  status_id, transition_condition, transition_to_status_id,
↪  operationid
) VALUES (2, 'Running', 3, '_operation_start_validation');
```

- **Passive transitions**: Transitions that do not execute any action when they're activated. They are used for final outcomes (e.g. transitioning to the last State of the FSM), or when the following Transition to be activated is triggered by a finished asynchronous operation or by the user.

```
INSERT INTO workload_machine_state (
  status_id, transition_condition, transition_to_status_id,
↪  operationid
) VALUES (3, 'OK', 4, NULL);
```

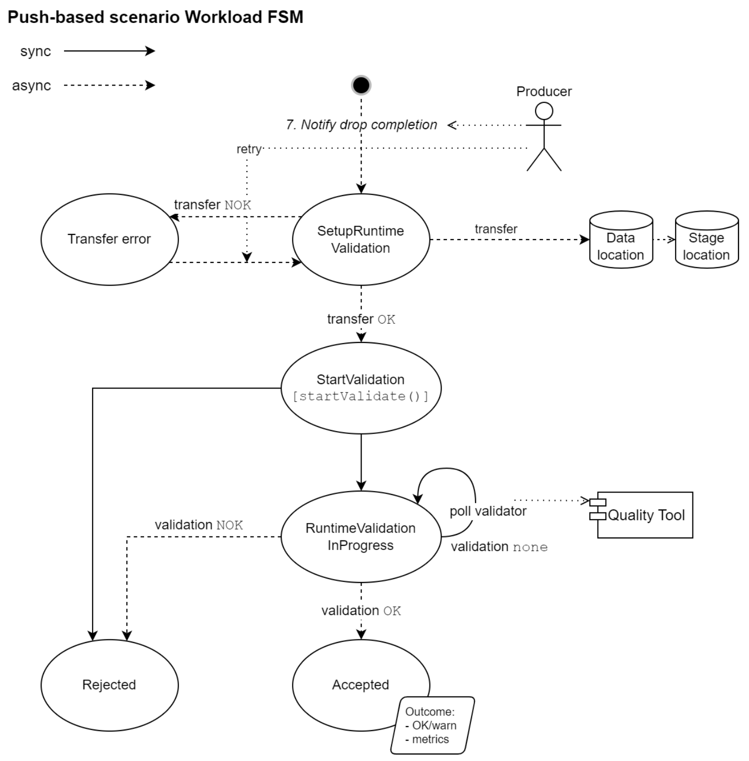
- **Self transitions**: Transitions where the "from" and "to" States are the same state. They may include an action or not. It's useful for repetitive actions that may be performed until another Condition is achieved. When defining self transitions is important to be careful and implement a base case on the operation to break self transitions, as currently the Manager doesn't have a way to interrupt infinite loop executions.

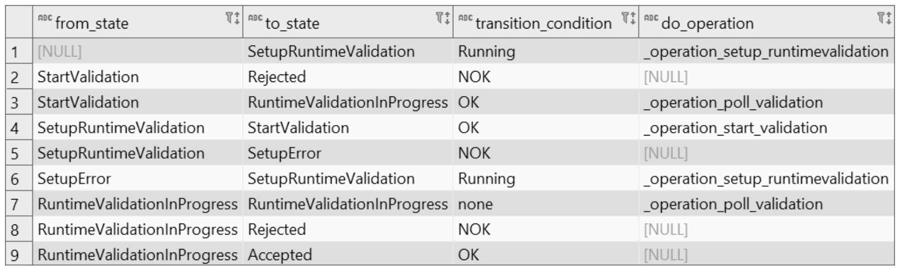
```
INSERT INTO workload_machine_state (
status_id, transition_condition, transition_to_status_id,
↪  operationid
) VALUES (3, 'none', 3, '_operation_poll_validation');
```

As a result of this design, Figure 3.7 shows an example of the Finite State Machine that implements the proposed push-based scenario, handling the states of workload creation, data transfer, and validation. Transitions can be triggered either by the system itself when a task finishes with a specific result, by the execution of an asynchronous callback, or by a user via service endpoints. Furthermore, there are two end states, Accepted and Rejected, which model the outcomes of the data validation and can be used by the downstream users to check the final result of the process.

Lastly, it is possible to see how a recurring task is implemented using this system. The runtime data validation is modeled in this state machine as an asynchronous task that polls the quality tool results until the validation finishes and returns a valid status. Internally, since the transition goes from and to itself, it translates into the workload calling recursively itself until it receives the answer. This is important, as appropriate timeout or failure mechanisms should be implemented to avoid infinite recursion.

A Workload contains the Result of itself, specific to each state the workload passes. The Result is used to determine the transition on the state machine and will be what ultimately defines the success/failure of the workload. Furthermore, it





	from_state	to_state	transition_condition	do_operation
1	[NULL]	SetupRuntimeValidation	Running	_operation_setup_runtimevalidation
2	StartValidation	Rejected	NOK	[NULL]
3	StartValidation	RuntimeValidationInProgress	OK	_operation_poll_validation
4	SetupRuntimeValidation	StartValidation	OK	_operation_start_validation
5	SetupRuntimeValidation	SetupError	NOK	[NULL]
6	SetupError	SetupRuntimeValidation	Running	_operation_setup_runtimevalidation
7	RuntimeValidationInProgress	RuntimeValidationInProgress	none	_operation_poll_validation
8	RuntimeValidationInProgress	Rejected	NOK	[NULL]
9	RuntimeValidationInProgress	Accepted	OK	[NULL]

Figure 3.7: Finite State Machine for a Push-based data ingestion scenario, where synchronous operations are defined using a solid line, and asynchronous operations using dotted lines. Some states interface with external services, so they are inherently async.
The table below is the representation of the FSM in the database where the foreign keys have been resolved as names for readability. The initial state is represented as the entry with a `NULL` value in the `from_state` column

includes the final outcome of the validation, based on the quality rules enforcement performed by the workload, so it's possible to understand, along with the workload State, if the workload outcome was Accepted, Accepted with warning, or Rejected.

Data Contract format validation

Data quality validation is not only applicable to the dataset, but also to its surrounding metadata, and in this same fashion, to the data contract object as is itself a way of metadata to express the expectations and promises agreed between the two parties. The data contract enforcement would fail if the data contract itself doesn't attain certain levels of quality and structure. For this, a validation of the format of the contract is necessary to identify errors even before it is registered into the system, blocking the operation if this validation fails.

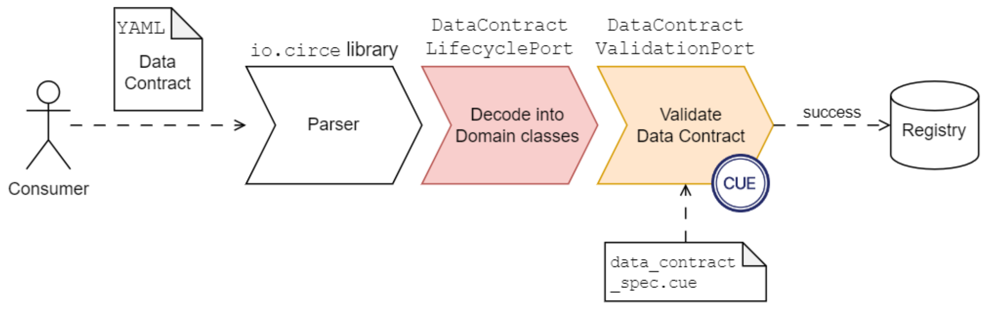


Figure 3.8: Data contract format validation done when a contract is being registered on the Manager. It parses, decodes, and validates the input to guarantee that is in the expected form and identifies many potential errors even before the contract is persisted in the system.

For this purpose, a two-step operation is done when the request to register a data contract arrives at the Manager. As can be seen in Figure 3.8, the Manager first tries to parse and decode the incoming request into domain objects, an operation where both the structure and some data type validations are performed; and then validates the input using CUElang using a crafted script for that specific format version of the data contract. As parsing is safer than validation and easier to implement with the help of specialized libraries, it is the first transformation done to the input. If the received data contract doesn't adjust to the expected domain objects, further operations in the Manager would simply fail and thus the request would be rejected. At the implementation level, this is done using Scala libraries like `circe` to parse and decode the YAML input into Scala classes.

Furthermore, after the data contract has been decoded into domain objects, an additional validation is executed against the YAML input of the contract using CUElang. Taking advantage of the strengths of this language to perform schema validations, the format of the contract is validated against a cue file including the restrictions to it. This cue file has the further advantage that it allows anyone to validate their contracts against the expectations of the Manager, avoiding the necessity to contact the system to verify its validity. The cue script used to validate the data contract can be seen in Appendix B.

Infrastructure layer

At the infrastructure level, the database schema of the Manager was designed to support basic queries on data contracts based on ID, tags, and actors, and queries on workloads based on ID and related data contract. The schema also allows the definition of the workload management Finite State Machine that is then loaded at runtime, defining the Manager processes. The Entity Relationship (ER) diagram of the database can be seen in Figure 3.9.

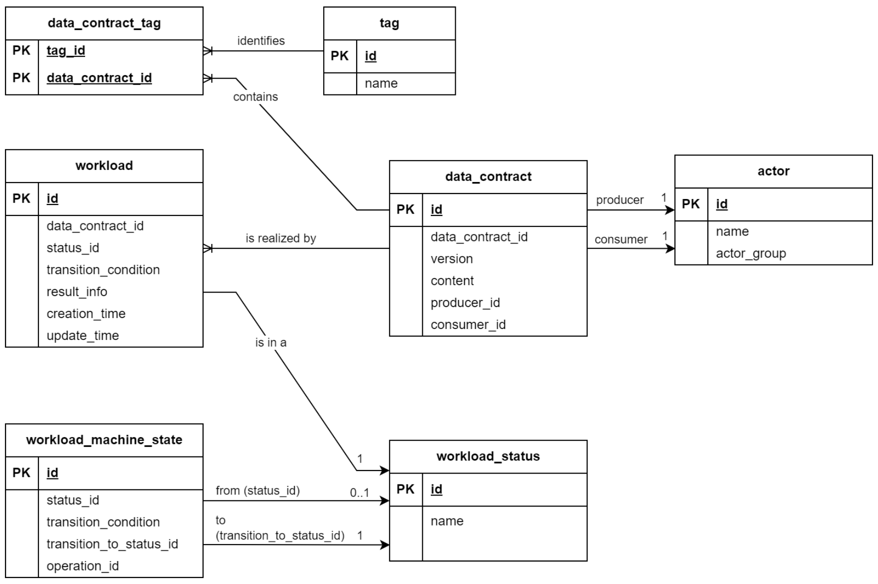


Figure 3.9: Data Contract Manager Entity Relationship diagram

Another important task that the Manager performs at the infrastructure level is the capacity to transfer data from one location to another by means of the Data Transfer domain. This allows workloads to move information as needed and allows the Validators to access the necessary information to perform their tasks. Currently, the implementation of the Data Transfer Infrastructure Port interface is used to transfer files inside a local filesystem, specifically between the location where the producer would drop the data on push-based data ingestion, and the location owned by the Validators to access and validate the quality. The specifics of this behavior will be explained later in the document.

Integration with Data Contract Validator

Since the data contract defines schema and quality rules in a declarative way, and the Manager is implemented to remain technology agnostic related to the way these data quality checks are executed, this responsibility is pushed down to a Validator service. For this motive, the Manager needs to interface with these services. The data contract quality rules, defined in Section 3.1, are basic rules that are simply propagated to a Validator that implements them. Moreover, the possibility to add `custom` rules allows bypassing the declarativeness of the quality rules to offer greater flexibility and strength, but it also means that the Data Contract Manager should be open to integrate different technologies and select the appropriate Validator to realize a certain quality rule.

The current design of the Manager solves this in the Domain layer, where the Data Runtime Validator Service groups the rules based on the technology that implements them or assigns a default technology for the declarative rules that are not tied to any specific one, as all Validators should be designed and implemented to understand the data contract rule format. Each group of rules is then transformed into a validation configuration object that shall be sent to the appropriate endpoint of each of the Validators, and that works as the starting point for them to build the quality checks. This way, every time the Manager interfaces with the Validators, it must wait until all of them answer in order to have a complete outcome, both at the kick-off of the validation and when performing polling of the result. This behavior is summarized in the diagram in Figure 3.10.

Since the Manager remains technology-agnostic, the design of the API interface specification demands the same definition, allowing validation configurations to be sent to the Validator, and eventually receiving validation status from them that the Manager can understand and handle. The interface in its most basic form defines two endpoints: The first one is used to start the validation using the validation configuration object including the quality rules and other parameters required to access the data. This endpoint is designed to just start asynchronously the validation and won't return the quality check results, as these checks might take an undefined amount of time to complete. This is also consistent with the provided example of the FSM. The second endpoint is used to poll the validation results of a specific workload until the final outcome is calculated, moment in which the Manager will accept or reject the transfer and save the result. Further details on how the validation is performed will be provided in the following section.

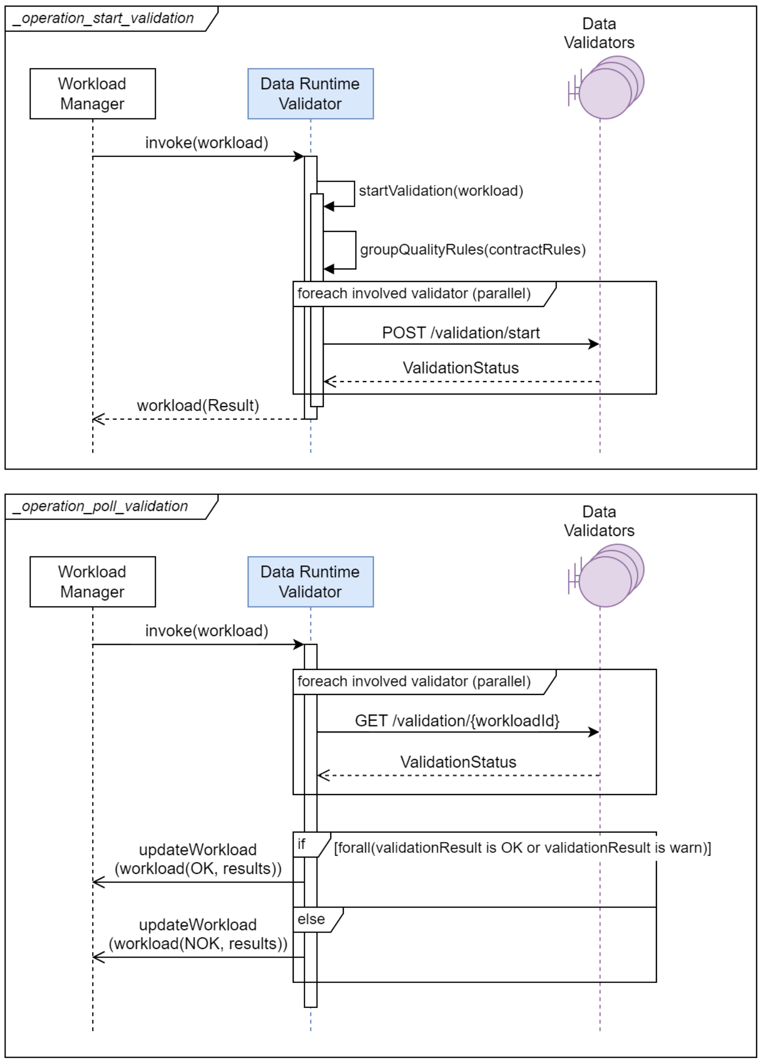

_operation_start_validation
Workload Manager
Data Runtime Validator
Data Validators
invoke(workload)
startValidation(workload)
groupQualityRules(contractRules)
foreach involved validator (parallel)
POST /validation/start
ValidationStatus
workload(Result)
_operation_poll_validation
Workload Manager
Data Runtime Validator
Data Validators
invoke(workload)
foreach involved validator (parallel)
GET /validation/{workloadId}
ValidationStatus
if
[forall(validationResult is OK or validationResult is warn)]
updateWorkload (workload(OK, results))
else
updateWorkload (workload(NOK, results))

Figure 3.10: Sequence Diagram for the communication between Manager and Validator. On the first moment, the Runtime Validator Service triggers the validation where it groups the quality rules and communicates with the Validators. On a second moment, asynchronously, it polls the Validators, gathering the results and updating the workload outcome.

3.3 Runtime Validation

While the Data Contract Manager is designed as a microservice to abstract the complexity of handling a data contract, it doesn't perform any validation on the data itself, only on the contract itself and its lifecycle. This is the task of the Data Contract Validator that at runtime will computationally generate and perform validations on data based on the quality rules agreed between the interested parties.

3.3.1 Analysis

As a starting point, to evaluate the architecture of the Validator inside the system, an initial high-level design was performed. Specifically, a focus was made on the aspects related to the software architecture of the service, the exposed interface by the Validator for the Manager to communicate with it, and the way the service should access the information to be validated. This last aspect was the most pressing one, as the different possibilities for doing so could change drastically how the system behaves and performs.

In a push-based scenario, the producer transfers the data into a predefined location, which is agreed on this context inside the data contract specification. The issue arises when the system, both the Manager and the Validator, but especially this last one, need to access this location to perform their intended job.

The first analyzed possible solution was to provide the Validator with the appropriate access control to the location where data had been pushed. The advantage of this approach is having direct access to the source of data, so no extra data transfer needs to be done, since the validations would be performed directly on the target location. The disadvantage of this approach lies in the fact that the Validator would need to receive not only the validation configuration but also access control to every location to which data will be pushed, which might lie in different domains under different ownerships, each one with possibly their own access protocol or formats, increasing the complexity of the Validator in its task.

Taking this into account, the chosen solution is based on a “stage” location owned by the Validator, thus not needing extra access control management, meaning it can easily access the data present on it. The added complexity lies now in the new responsibility of the Manager to transfer the data from the location defined in the contract (whether it’s a remote/local or source/target location) to the Validator stage location before triggering the quality checks, thus introducing the Data Transfer domain into the application. By doing this, the Validator is left use-case independent and with the single responsibility of translating quality rules and executing them.

It’s important to note that a further disadvantage of this approach is that it might constrain the size of data that can be analyzed, as transferring large quantities of information could be expensive both in time and cost. However, as the workloads executed by the Manager are designed in a modular and declarative way, better solutions can be designed and implemented in the future.

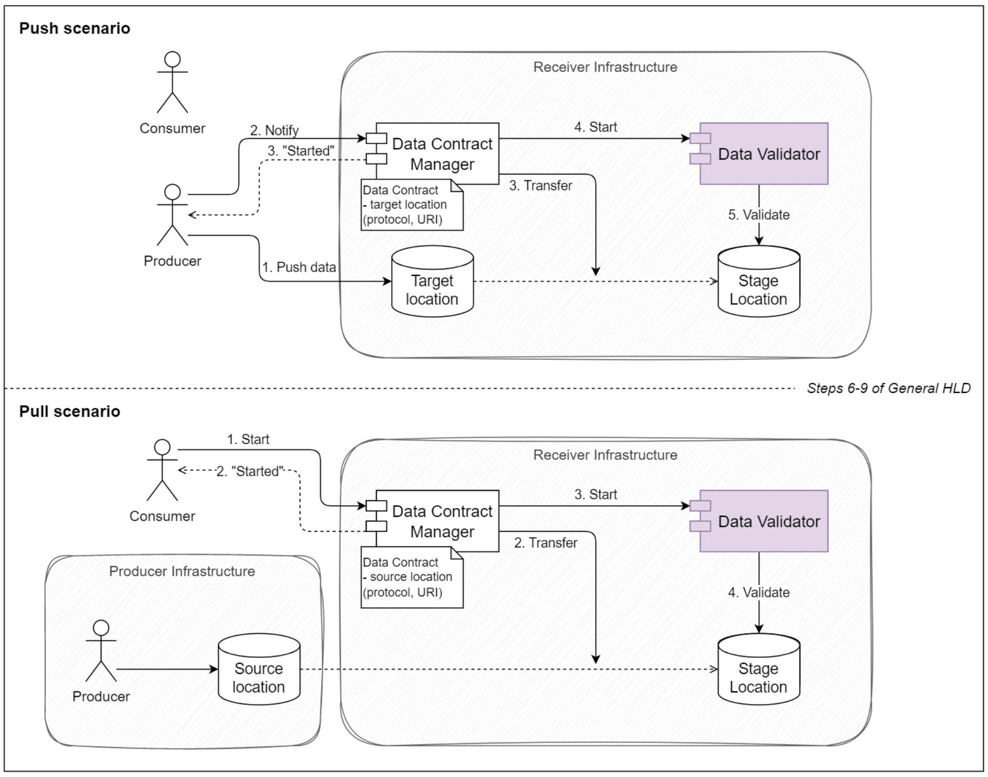


Figure 3.11: High-Level Design for the start of the data validation process. Externally, the process is different based on the user that triggers the process and the data location. Internally, the process remains the same.

In consequence, even if this transfer flow will slightly differ depending on whether the scenario is pull-based or push-based —as the user who triggers the workload will change, and potentially the source location will as well—, for the Manager and Validator system, at a high level they will follow a consistent pattern which can be seen in Figure 3.11: An HTTP request signals the availability of data, prompting the Manager to initiate a transfer to the staging location, returning immediately with an OK status and the Validator will schedule asynchronously the quality checks execution, making this step asynchronous. While the quality checks are being executed, the Manager will contact the Validator using a poll strategy to update the workload status, which in turn will be polled by the user to retrieve the outcome of the process.

3.3.2 Design

The following design refers specifically to the Validator designed as part of the implemented proof of concept. The design process is similar to the one performed on the Data Contract Manager, done in parallel with this one. As the responsibilities of the Validator are reduced compared to the Manager, the architecture of the software is smaller and simpler. Nonetheless, the same three-layer architecture was chosen as the system still needs to expose API interfaces, run business logic, and communicate with infrastructure services, especially the library or framework that performs the validation. The same Ports and Adapters architecture was followed, but in this case, its advantages aren't as striking, as this component is smaller and more technology-oriented. As the Data Quality tool, Great Expectations was chosen as the first Validator. Figure 3.12 shows the proposed domain and component diagram.

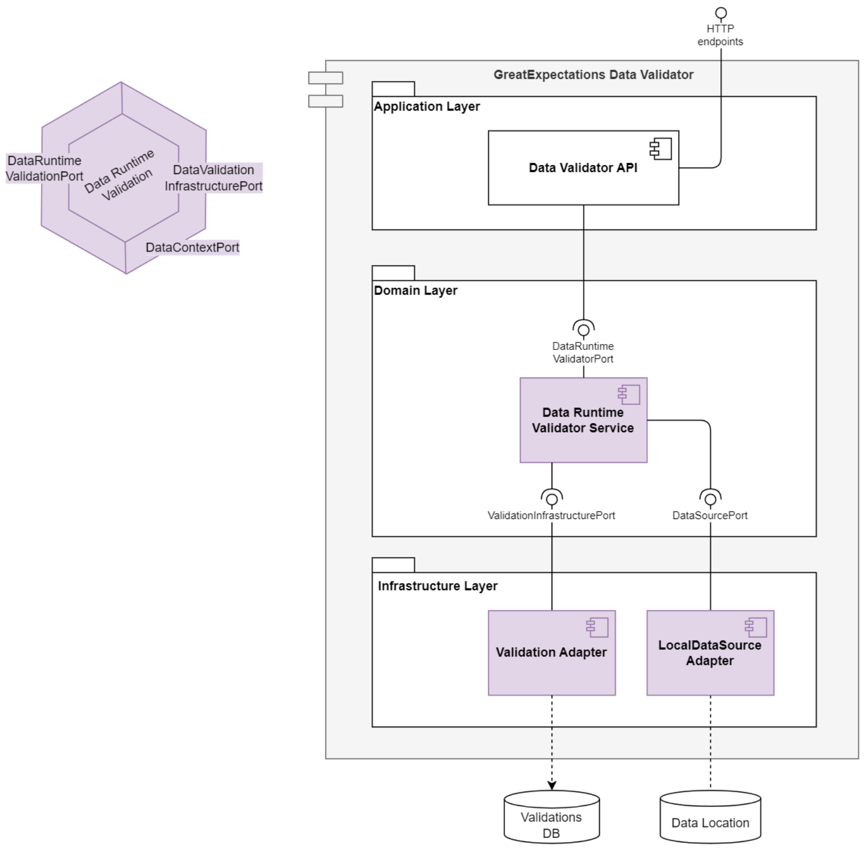


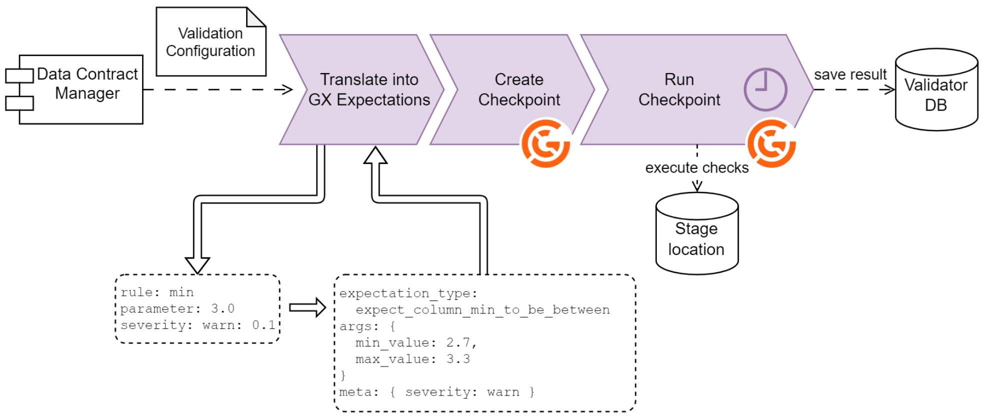
Figure 3.12: Domain and Component diagram for a Validator. The three layers can be seen from top to bottom: Application, Domain and Infrastructure.

As already discussed, quality checks can be time-consuming so this process is designed to be asynchronous. For this, the Validator must persist the results on a database in order to serve requests for the outcome of any workload. These validation outcomes, even if they're generated by a specific technology, must conform to a technology-agnostic business schema that the Manager knows how to handle, in the same fashion as the difference between the received quality rules and the implementation of these rules in the data quality tool domain-specific language. Thus, the Validator must hold the responsibility of translation between the Data Contract Manager's business domain and the data quality language.

3.3.3 Implementation

For the development of the first Validator microservice, Great Expectations was used as the underlying data quality tool, developing the microservice in Python 3.11. The service runs under the Asynchronous Server Gateway Interface (ASGI) standard using the `uvicorn` library, allowing for asynchronous operations to be executed under Python. As a potential addition, since the validation checks might be time-consuming and might block the main thread, it has been considered to utilize the library `gunicorn` that provides the baseline to work with multiple worker instances of the service, orchestrated by a master node. Along with this, the library `FastAPI` was used to implement the application layer, and following the Manager technology stack, a containerized PostgreSQL accessed via the `psycopg2` library was used for the database infrastructure. This database instance is not the same as the Manager's, and the decision to use different instances for the two services was taken to maintain independence between these two environments and on their deployments.

The general process performed by the Validator, comprising rule translation and quality check execution, can be summarized as seen in Figure 3.13. The execution is based both on Great Expectation Checkpoints, as well as the Validator database, where results are stored in domain objects to be polled by the Manager and updated asynchronously when the Checkpoint execution finishes. The Validator stores only the metric associated with a certain rule ID, column, and severity level, but does not perform business logic regarding the success or failure of the whole validation. Even if it possesses the information to do so by examining the outcome of each expectation with the defined severity level, is a responsibility that lies on the Data Contract Manager as part of the Data Runtime Validation domain. Specifically, since the success or failure of a workload can depend on more criteria than the success of an expectation alone, and the acceptance logic based on a severity level may trigger further actions down the workload state machine, it's important to keep this behavior encapsulated on the Manager.



```
rule: min
parameter: 3.0
severity: warn: 0.1
```

```
expectation_type:
  expect_column_min_to_be_between
args: {
  min_value: 2.7,
  max_value: 3.3
}
meta: { severity: warn }
```

Figure 3.13: Data Validation process on a Great Expectations Validator.

Application layer

The exposed API interfaces of the Data Contract Manager have been defined as follows:

- `POST /v1/validation/start`: Starts a data validation using the settings in the validation configuration
  - **Request body example**:
    ```
    {
      "workloadId": "64f92252-a20d-45a2-b09c-c61f472a6023",
      "validations": [
        {
          "id": "row_size_rule_id",
          "dimension": "completeness",
          "ruleType": "size",
          "parameter": 10000,
          "severity": {
            "warn": { "tolerance": 0.1 },
            "fail": { "tolerance": 0.15 }
          }
        },
        {
          "id": "allowed_values_id",
          "dimension": "validity",
          "ruleType": "allowedValues",
          "columns": [ "payment_type" ],
          "parameter": [1, 2, 3, 4, 5, 6]
        }
      ],
      "access": {
        "path": "trips_data/",
        "pattern": "trips_[\\d]+.csv"
      }
    }
    ```
  - **Response**: `200 OK` (success) with the validation status model with status

`OK` if the validations started correctly, or `400 Bad Request` if a validation error occurred while creating the data quality checks.

  - **Response body example**:

```
{
  "workloadId": "64f92252-a20d-45a2-b09c-c61f472a6023",
  "status": "OK",
  "result": []
}
```

- `GET /v1/validation/{workloadId}`: Polls the status of a workload validation using a workload ID
  - **Request body**: None
  - **Response**: `200 OK` (success) with the validation status model with status `OK` if the validations started correctly, or `400 Bad Request` if a validation error occurred while creating the data quality checks.
  - **Response body example**:

```
{
  "workloadId": "64f92252-a20d-45a2-b09c-c61f472a6023",
  "status": "OK",
  "result": [
    {
      "id": "num_rows_rule",
      "column": null,
      "metric": 10000,
      "severity": {
        "fail": {
           "success": true,
           "exception_info": { "raised_exception": false }
        },
        "warn": {
          "success": true,
          "exception_info": { "raised_exception": false }
        }
      }
    },
    {
      "id": "allowed_values_id",
      "column": "payment_type",
      "metric": [1, 2, 3, 4],
      "severity": {
        "fail": {
          "success": true,
          "exception_info": { "raised_exception": false }
        },
        "warn": null
      }
    },
  ]
}
```

Domain layer

The technology implementation of the declarative quality rules into Great Expectations expectations was done using the core expectations provided by the library in its Contributions Gallery. For each of the declarative rules, the appropriate expectation name was chosen, and the parameters were calculated based on the affected columns and the severity tolerance values. This means that the correspondence between data contract quality rules and expectations is not a one-to-one correspondence, but rather one-to-many, as each quality rule, following the specification defined in Table A.3, provides a possible list of columns to be analyzed with the same rule, as well as two types of possible severities (warning and failure) that impact the parameters to be used in the expectation. Table 3.2 shows the translation between the quality rule types and the expectations provided by the data quality tool.

If the translation succeeds, the Great Expectations checkpoint object is created, containing the Expectation suite and the Data Source object to access the data. At this point, a successful response is returned to the Manager, where this success characterizes not the success of the quality checks, but of the setup and kickoff of the validation. Then, the data quality executions are started asynchronously and the checkpoint is executed. During this execution, the Validator is open to more requests, either new validations to be created for other contracts, or polling for workload results. When the execution finishes, the Expectation results are translated into business domain objects and persisted, ready for queries requested by the Manager.

Infrastructure layer

The infrastructure layer of the Validator is divided into two main aspects, the Great Expectations API interface and the PostgreSQL database. The first one is provided by the Great Expectation Python library, whereas the second consists of a small containerized database instance containing a single table to store the validations, its current status (OK/NOK/Running), and the eventual list of results. As mentioned above, the OK/NOK status doesn't imply the success or failure of the data quality rules, but the execution of the process. The outcome of the quality rules is calculated by the Data Contract Manager with the received results.

With the outcome of the Validator and the calculation of the success or failure of the data contract enforcement, is possible to analyze if the received results are what was expected when the contract was agreed. By implementing these two services, is possible to validate the goodness of the data and avoid that faulty data can be accepted by a domain.

Rule	Severity	Expectation name	Expectation arguments
`size`	None	`expect_table_row_count_to_equal`	`{value: parameter}`
	warn/fail	`expect_table_row_count_to_be_between`	`{min_value: parameter*(1-tolerance), max_value: parameter*(1+tolerance)}`
`complete`	None	`expect_column_values_to_not_be_null`	`{column: column, mostly: parameter}`
	warn/fail	`expect_column_values_to_not_be_null`	`{column: column, mostly: parameter-tolerance}`
`unique`	None	`expect_column_proportion_of_unique _values_to_be_between`	`{column: column, min_value: 1.0}`
	warn/fail	`expect_column_proportion_of_unique _values_to_be_between`	`{column: column, min_value: 1.0-tolerance}`
`min`	None	`expect_column_min_to_be_between`	`{column: column, min_value: parameter}`
	warn/fail	`expect_column_min_to_be_between`	`{column: column, min_value: parameter*(1.0-tolerance)}`
`max`	None	`expect_column_max_to_be_between`	`{column: column, max_value: parameter}`
	warn/fail	`expect_column_max_to_be_between`	`{column: column, max_value: parameter*(1.0+tolerance)}`
`allowedValues`	-	`expect_column_distinct_values_to_be _in_set`	`{column: column, value_set: parameter }`
`mean`	None	`expect_column_mean_to_be_between`	`{column: column, min_value: parameter, max_value: parameter}`
	warn/fail	`expect_column_mean_to_be_between`	`{column: column, min_value: parameter*(1.0-tolerance), max_value: parameter*(1.0+tolerance)}`
`stdev`	None	`expect_column_stdev_to_be_between`	`{column: column, min_value: parameter, max_value: parameter}`
	warn/fail	`expect_column_stdev_to_be_between`	`{column: column, min_value: parameter*(1.0-tolerance), max_value: parameter*(1.0+tolerance)}`
`custom`	-	`call` field in the rule	`{column: column, **args}`

Table 3.2: Translation table between data contract quality rules and Great Expectations expectation configurations. For each quality rule, one expectation is created. The total number of expectations created by a single quality rule is dependent on the number of columns defined on it and the presence or absence of a severity tolerance.

Chapter 4

Results

By having implemented the two services, Data Contract Manager and Great Expectations Data Contract Validator, containing the features mentioned in the previous chapter, is now possible to tackle real-life use cases. In order to test the application, a dataset where its quality could be deemed as "average" was chosen. For this, and following the arguments exposed in this thesis, is necessary to first define a business domain so that we could define a desired level of data quality and thus be able to quantify what "average" would mean in the business context.

For testing the actual usability of the system, a weather domain was used. Specifically, related to hourly weather measures of the Southeast region of Brazil in 2019. A dataset was chosen following this, where the business interpretation of the dataset fields is explained in Table 4.1. Overall, the dataset contains measurements on precipitation, temperature, solar radiation humidity, wind, and other measurements, taken by the hour by several weather stations in the evaluated region. The data is provided by Instituto Nacional de Meteorologia (INMET), the national institute of meteorology of the country.

In general, the dataset contains physical measurements, so it is expected that measurement columns may contain missing values due to sensor errors, downtime, etc., although it is still expected that the majority of the values should be present. On the other hand, fields referring to the identification of the record, like timestamp, weather station and its location, are expected to always be present as these do not rely on physical measurements but rather on record identification and are part of the metadata of the record.

column	type	description	example
index	number	Index of the record among each weather station measurements	1
date	string	Date of the measurement in format `YYYY-MM-DD`	2019-05-12
hour	string	Hour of the measurement in format `hh:00`	04:00
pcpn_mm	number	Amount of precipitation in millimeters in the last hour	5.1
station_pressure_mb	number	Atmospheric pressure at station level in millibars	928.6
pressure_max_mb	number	Maximum air pressure for the last hour	928.6
pressure_min_mb	number	Minimum air pressure for the last hour	928.3
radiation_kj_m2	number	Solar radiation in Kilojoules per squared meter.	1784
temp_c	number	Air temperature at the moment of measurement in celsius	25.4
dewpoint_c	number	Dew point temperature at the moment of measurement in celsius	16.2
temp_max_c	number	Maximum temperature for the last hour	28.6
temp_min_c	number	Minimum temperature for the last hour	27.3
dewpoint_max_c	number	Maximum dew point temperature for the last hour	17.9
dewpoint_min_c	number	Minimum dew point temperature for the last hour	17.1
humidity_max_perc	number	Maximum relative humid temperature for the last hour in percentage from 0 to 100	45
humidity_min_perc	number	Minimum relative humid temperature for the last hour in percentage from 0 to 100	40
humidity_perc	number	Relative humid at the moment of measurement in percentage from 0 to 100	43
wind_direct_grad	number	Wind direction in radius-degrees (0-360)	152
gust_max_m_s	number	Wind gust in meters per second	11.4
wind_speed_m_s	number	Wind speed in meters per second	1.7
region	string	Brazilian geopolitical regions. The dataset contains only the Southeast region (SE)	SE
state	string	Brazilian state two-letter code. The Southeast region is composed by the states of São Paulo (SP), Rio de Janeiro (RJ), Minas Gerais (MG) and Espírito Santo (ES)	SP
station	string	Station name	Nova Venecia
station_code	string	INMET Station code	A623
lat	number	Geographical latitude	-18.695265
lon	number	Geographical longitude	-40.390572
elev	number	Geographical elevation	156.02

Table 4.1: INMET Weather Dataset specification

After defining the domain and context of the data, it's possible to set a data contract in order to evaluate the desired quality attributes the dataset should possess. Then, by registering the contract in the system and running the appropriate workload, it will be possible to evaluate which quality rules have been broken, fix the procedure that generates the data, and then repropose the validation with the new, refurbished dataset. This way, it is possible to compare the outcome of a bad versus a good dataset.

Furthermore, by executing this process, several metrics can be measured about the process performance, like the average response time for each query (register data contract, find data contract, create workload, starting validation, polling for results, and others), and the percentage of broken rules before and after the dataset correction.

By taking a look at the columns definition, it is possible to create a set of quality rules that ensure that the necessary fields are present and in the correct format and domain (either numeric or value domain). For example, the humidity percentage values should be expected to be always between 0 and 100, as negative percentages or greater than 100% are mathematically impossible. Additionally, as a measurement of high quality and fidelity, at least 95% of all the weather measurements should be present, with the exception of the *Solar Radiation* column, as night-time records shouldn't report this value. Thus, the full set of quality rules that have been designed are listed below.

1. The `index, date, hour, region, state, station, station_code, lat, lon, elev` columns must not contain any null values, as these fields are the identifiers of the measurement.
   - **Dimension**: Completeness
   - **Type**: Complete
   - **Parameter**: 1.0
   - **Severity**: Failure, no tolerance.
2. All the weather measurements columns except `radiation_kj_m2` must be 95% not null, to ensure that most of the measurements exist in the dataset.
   - **Dimension**: Completeness
   - **Type**: Complete
   - **Parameter**: 0.95
   - **Severity**: Warning: 5%, Failure: 10%
3. The `radiation_kj_m2` column must contain values for 50% of the measurements, to account for the night-time hours and with a large tolerance to account for yearly daytime length variations.
   - **Dimension**: Completeness
   - **Type**: Complete
   - **Parameter**: 0.5
   - **Severity**: Failure: 15%
4. The `pcpn_mm, station_pressure_mb, pressure_max_mb, pressure_min_mb, radiation_kj_m2, humidity_max_perc, humidity_min_perc, humidity_perc, wind_direct_grad, gust_max_m_s, wind_speed_m_s` columns must contain non-negative values, which are physically impossible.
   - **Dimension**: Validity
   - **Type**: Min
   - **Parameter**: 0.0

- **Severity**: Failure, no tolerance.

5. The `humidity_max_perc, humidity_min_perc, humidity_perc` columns must not contain values larger than 100, since these fields represent percentages ∈ [0,100].
    - **Dimension**: Validity
    - **Type**: Max
    - **Parameter**: 100.0
    - **Severity**: Failure, no tolerance.
6. The `wind_direct_grad` column must not contain values larger than 360, since this field represents angle degrees ∈ [0°,360°].
    - **Dimension**: Validity
    - **Type**: Max
    - **Parameter**: 100.0
    - **Severity**: Failure, no tolerance.
7. The `state` column must contain only state codes for states belonging to the Southeast region.
    - **Dimension**: Validity
    - **Type**: Allowed Values
    - **Parameter**: SP, RJ, MG, ES
    - **Severity**: Failure, no tolerance.
8. The `date` column must be formatted in the ISO format "YYYY-MM-DD".
    - **Dimension**: Validity
    - **Type**: Custom
    - **Technology**: Great Expectations
    - **Call**: `expect_column_values_to_match_strftime_format`
    - **Parameter**: `%Y-%m-%d`
    - **Severity**: Failure, no tolerance.
9. The `hour` column must be formatted in the ISO format "hh:mm".
    - **Dimension**: Validity
    - **Type**: Custom
    - **Technology**: Great Expectations
    - **Call**: `expect_column_values_to_match_strftime_format`
    - **Parameter**: `%H:%M`
    - **Severity**: Failure, no tolerance.

Using these rules, a data contract was created following the data contract specification and registered on the Manager using the respective endpoints (see Appendix A for the contract content). For testing purposes, the contract enforcement was performed against a slightly modified dataset to introduce faulty measurements in two columns: `wind_direct_grad` and `humidty_perc`. The error injection was performed by adding values outside the domain of each column defined by the rules to 6% of the rows. This was done in order to evaluate further the system's ability to identify breaking rules. Taking this into account, it is expected that the initial validation would contain at least 2 rejected rules.

By executing the validation workload, it is possible to get a detailed result on the rules that were respected against those that were broken. The initial test of the dataset showed that out of the initial nine rules, 45 validations were generated (as each validation is applied to a single column), and from these, only 33 were initially successful (73.3% success rate). As this is a result much lower than expected, it is necessary to analyze the metrics provided in the results, to understand which rules were broken and why, as well as perform basic analysis of the read data, to possibly identify further errors with the original dataset. The biggest issue with the dataset is the use of -9999 as the value for null readings, instead of an empty field as it's normally done and was expected on the quality rules. These kinds of insights can then be forwarded to the data producer in order to be fixed for the producer to be compliant with the agreed contract.

After the correction of this value, the validation workload is re-executed and the new result yields 41 out of 45 successful validations (91.1%) plus two warning validations, so the remaining failed validations are related to the error injection. Figure 4.1 illustrates the result of the validations and shows that by sticking to the agreed rules, it's possible to increase the overall quality of the data, blocking major issues as soon as they are detected, and avoiding sending faulty data into the consumer pipeline.

Additionally, as this process was being executed, performance results on the systems were measured. Firstly, the execution time for different actions performed by the Data Contract Manager and Data Contract Validator was measured. These metrics were taken with the same dataset by executing the data contract registration process $n = 40$ times and the data contract enforcement process $m = 24$ times on a local Windows machine with 16GB of RAM and 1.80GHz i7 Intel Core processor. Some steps are executed several times during the same registration or enforcement process, so measurement amounts vary for each step but are always $n \geq 40, m \geq 24$. Since the measurements were taken in a local machine, outliers with a Z-score > 2.0 were removed to account for system usage peaks that may impact the time measurements.

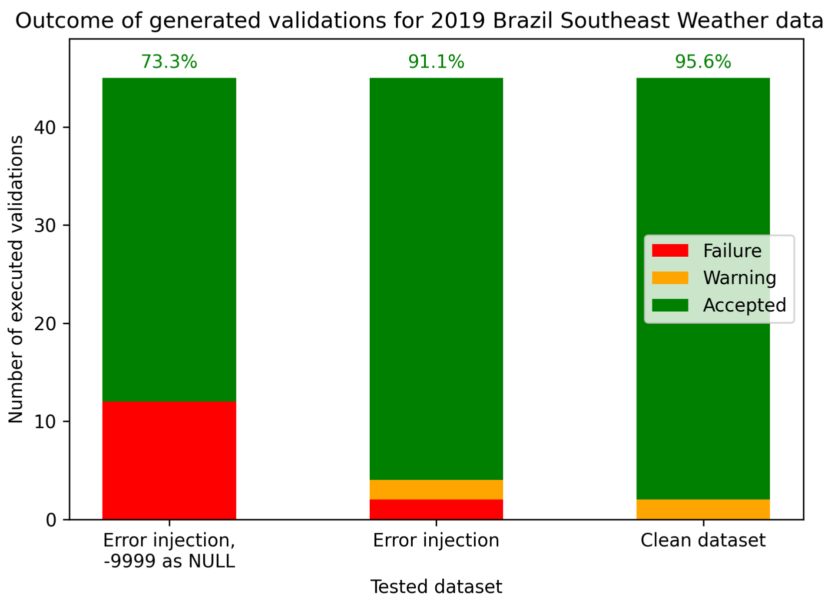


Figure 4.1: Comparison of validation outcomes on different quality levels of a dataset. By validating the data against the agreed rules, it is possible to quickly identify breaking quality rules, and to improve the overall quality of the dataset

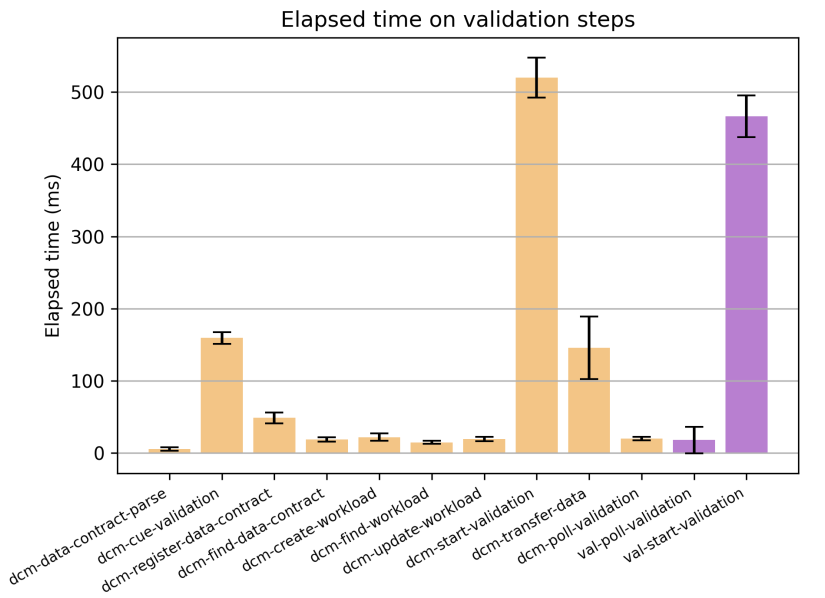


Figure 4.2: Execution time for actions performed by the Data Contract Manager (orange) and the Data Contract Validator (purple).

The results of these time measurements can be seen in Figure 4.2, excluding the actual quality check executions which will be shown later. By analyzing the results, it's evident that the processes related to some kind of validation take the most time. In particular, the graph shows peaks on the steps of the *CUE validation* step at registration time, the *Start Validation* step at validation time which account for the initial contact with the Validators, and the Validator *Start Validation*, which account for the rule translation into expectations and the creation of the checkpoint. It's important to notice that the *Start Validation* execution time shown in the figure, comprises also the time taken to receive the response from the Validators as this is a synchronous operation, so it includes the Validator's own *Start Validation* step elapsed time, so the actual Manager time is much lower.

By analyzing the results, is clear that the time spent on the *Transfer Data* step is lower than other steps, so the initial concern about slow executions isn't actually an issue. Moreover, all of the shown steps, including the transfer of data, are orders of magnitude lower than the actual validation process as will be seen later, and it is also evident that most of the system actions are in the order of tens of milliseconds, with most non-validation actions taking $50ms$ or lower, showing a good performance.

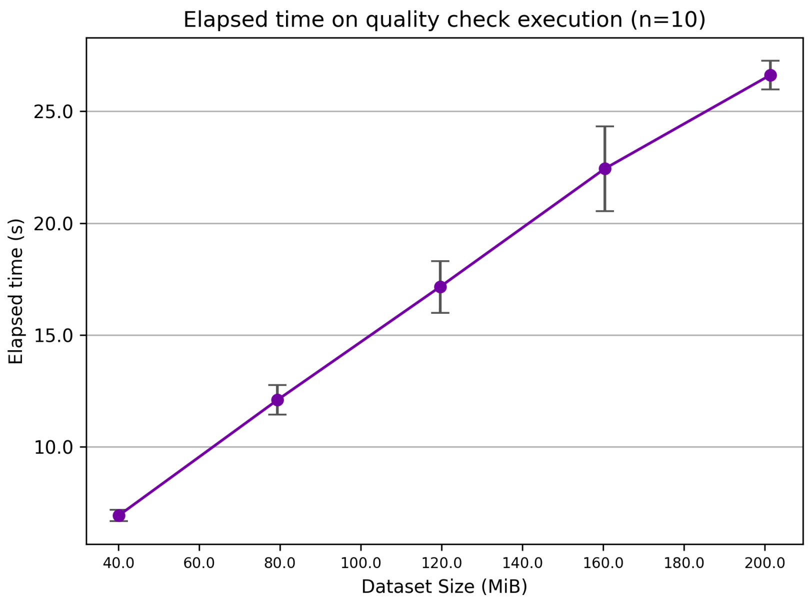


Figure 4.3: Execution time for the GreatExpectations quality check on various sizes of the analyzed dataset.

In that sense, and regarding the quality check execution, Figure 4.3 shows the elapsed time for the checkpoint execution on Great Expectations for different dataset sizes. Measurements were taken using the same dataset by varying the number of rows, and for each size $n = 10$ executions were performed. Since the measurements were taken in the same local machine as the others, outliers with a Z-score > 2.0 were removed to account for system usage peaks that may impact the time measurements. After collecting the values, it can be observed that the executed validation time is in the order of tens of seconds, and scales linearly with the dataset size, taking $26.62 \pm 0.64s$ on a dataset of size 201.4MiB.

With these results, and by taking the average elapsed time for each performed transition, it can be seen that the validation process accounts for around 96% of the whole execution process, so if further developments are performed on this application and these are concerned about time performance, the Validators' execution should be the main point for improvement.

Nonetheless, these numerical results validate the practical usability of the system and allow for further evaluations on performance, quality improvement, etc., thus validating not only the theoretical results by the implementation of the data contract format and the enforcement system but also the practical application of such a system. With only two executions of the validation process, it was possible to improve the overall quality of the dataset and show the potential of such a system with more complex rules.

Conclusions

In light of the extensive work undertaken to create and implement a data contract format and enforcement system, this thesis successfully demonstrates the usefulness of the development and implementation of a data contract format and a management service that can be integrated into data architecture platforms. Firstly, the data contract format was designed to be an extensible and flexible specification following closely the currently existing proposals. By carefully considering the varying needs of data contracts across different domains, the format provides a robust foundation that can accommodate evolving requirements and diverse data structures.

Moreover, the design of the Data Contract Manager has proven to create a successful service in terms of extensibility, configurability, and versatility. The manager's potential adaptability to different contexts and the ease with which it can be customized to meet specific organizational needs justify its practical utility. Additionally, the establishment of well-defined interfaces throughout the system, especially between the manager and the validators, positions the data contract system as a reliable tool for data governance, allowing it to identify faulty data as soon as it arrives in the system on a trust-but-verify scenario.

While the scope of this project was intentionally limited to serve as a functional proof of concept, the results obtained have validated the feasibility of the proposed approach, paving the way for future developments and integrations into more comprehensive and robust systems. In this sense, this thesis has not only contributed to the immediate goal of data contract enforcement but also left the door open for further advancements in handling a wide range of workloads and data contract scenarios.

4.1 Future developments

The outcome of this thesis offers a working baseline to start registering and validating data contracts in a controlled environment and works as a strong basis to build on, especially to overcome the limitations and reduced scope that were defined for the implementation. Specifically, the Data Contract Manager and Data Contract Validators are open to supporting new features, such as execution on a cloud environment; access to data store in cloud storage; and consequently achieving the main goal of the project of integrating this system as a service on the Witboost environment, to be part of the services that the platform has to offer.

In the same fashion, the project is open to extensions on the validation capabilities of the system. Additional validations can be implemented on either the Manager or Validator taking advantage of the data contract specification, which provides extensive metadata that is not being used in its totality. As an example, further validations based on the dataset schema or SLAs can be performed by including the appropriate validators. Furthermore, by taking advantage of the extensibility of the workload definition, more ingestion scenarios can be implemented, such as pull-based or stream scenarios on both source-aligned and consumer-aligned data products.

Lastly, a close look should be taken at the developments and improvements done by other communities and their proposals in the context of data contracts, as this is a subject that is rapidly evolving. As such, it is expected that different viewpoints and solutions to the same problem arise, but that the joint contribution of communities from different environments of the data engineering practice will provide a robust way to handle data contracts and data quality moving forward.

Bibliography

[1] International Organization for Standardization. *Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Data quality model.* ISO/IEC 25012:2008. International Organization for Standardization, 2008. URL: `https://www.iso.org/standard/35736.html`.

[2] Russell L Ackoff. "From data to wisdom". In: *Journal of applied systems analysis* 16.1 (1989), pp. 3–9.

[3] M. Golfarelli and S. Rizzi. *Data Warehouse Design: Modern Principles and Methodologies.* Mcgraw-Hill, 2009. ISBN: 9780071610391.

[4] Z. Dehghani and M. Fowler. *Data Mesh: Delivering Data-driven Value at Scale.* O'Reilly Media, 2022. ISBN: 9781492092391.

[5] J. Lechtenbörger. *Data Warehouse Schema Design.* Dissertationen zu Datenbanken und Informationssystemen. Aka, 2001. ISBN: 9781586032142.

[6] Amazon Web Services. *What is a data lake?* 2023. URL: `https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake`.

[7] M. Cagan. *INSPIRED: How to Create Tech Products Customers Love.* Silicon Valley Product Group. Wiley, 2017. ISBN: 9781119387503.

[8] M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis. *Fundamentals of Data Warehouses.* Springer Berlin Heidelberg, 2002. ISBN: 9783540420897.

[9] *The Six Primary Dimensions for Data Quality Assessment.* Data Management Association UK (CCSDS). Oct. 2013.

[10] Great Expectations. *Great Expectations. Welcome.* URL: `https://docs.greatexpectations.io/docs/`.

[11] CUElang. *About. How did CUE come about and what are its principles.* URL: `https://cuelang.org/docs/about/`.

[12] C. Sanderson. *The Rise of Data Contracts and Why Your Data Pipelines Don't Scale.* Aug. 2022. URL: `https://dataproducts.substack.com/p/the-rise-of-data-contracts`.

[13] *The State of Data Quality. Data Leader Strategies & Benchmarks.* Tech. rep. Monte Carlo, May 2023.

[14] Microsoft. *Introduction to Data Engineering.* Sept. 2021. URL: `https://learn.microsoft.com/en-us/dotnet/framework/wcf/architecture?redirectedfrom=MSDN`.

[15] H. L. Truong, G. R. Gangadharan, M. Comerio, S. Dustdar, and F. De Paoli. "On Analyzing and Developing Data Contracts in Cloud-Based Data Marketplaces". In: Dec. 2011, pp. 174–181. DOI: `10.1109/APSCC.2011.65`.

[16] OpenDataMesh. *Data Contracts.* Nov. 2022. URL: `https://dpds.opendatamesh.org/concepts/data-contract/`.

[17] PayPal. *Template for Data Contract.* URL: `https://github.com/paypal/data-contract-template`.

[18] AIDA User Group. *Open Data Contract Standard.* URL: `https://github.com/AIDAUserGroup/open-data-contract-standard`.

[19] dbt Labs. *Model contracts.* URL: `https://docs.getdbt.com/docs/collaborate/govern/model-contracts`.

[20] Agile Lab s.r.l. *witboost.* 2023. URL: `https://www.agilelab.it/witboost`.

[21] Elise Casey. *Data Pipeline Quality Checks.* Jan. 2023. URL: `https://medium.com/building-ibotta/pipeline-quality-checks-circuit-breakers-and-other-validation-mechanisms-761fc5b1ebe4`.

[22] Soda Data NV. *Soda Checks Language.* URL: `https://docs.soda.io/soda-cl/soda-cl-overview.html`.

Appendix A

Data Contract Format Specification

```
specVersion: 0.1.3
id: brazil-se-weather-contract-inmet-weather-domain-1.0.0
name: Brazil SE Weather Contract
kind: DataContract
version: 1.0.0
description: Contract to ingest hourly weather data from Brazil Southeast
↪  states

consumer:
  name: weather_domain
  group: weather_domain.com
producer:
  name: INMET
  group: inmet_mail.com

tags:
  - tagFQN: weather
    source: Tag
    labelType: Manual
    state: Confirmed
  - tagFQN: america
    source: Tag
    labelType: Manual
    state: Confirmed

access:
  eventType: push
  protocol: HTTP
  location: /files/weather/se
  accessConfiguration:
    pattern: '(.*)southeast_\\d{4}.csv'

dataset:
  name: weather_record
  schema:
    - name: index
      dataType: int
      description: ID of the record
      constraint: PRIMARY_KEY
    - name: date
      dataType: date
      description: Date of the record
      constraint: NOT_NULL
    - name: hour
      dataType: time
      description: Hour of the record
      constraint: NOT_NULL
    - name: pcpn_mm
      dataType: number
      description: Precipitation in millimeters
    - name: station_pressure_mb
      dataType: number
      description: Atmospheric pressure at station level in millibars
    - name: pressure_max_mb
      dataType: number
      description: Maximum atmospheric pressure in the last hour in
↪   millibars
    - name: pressure_min_mb
      dataType: number
      description: Minimum atmospheric pressure in the last hour in
↪   millibars
    - name: radiation_kj_m2
      dataType: number
      description: Solar radiation in kilojoules per squared meter
    - name: temp_c
      dataType: number
      description: Temperature at reading time in celsius
    - name: dewpoint_c
      dataType: number
      description: Dew point temperature at reading time in celsius
    - name: temp_max_c
      dataType: number
      description: Maximum temperature in the last hour in celsius
    - name: temp_min_c
      dataType: number
      description: Minimum temperature in the last hour in celsius
    - name: dewpoint_max_c
      dataType: number
      description: Maximum dew point temperature in the last hour in
  ↪  celsius
    - name: dewpoint_min_c
      dataType: number
      description: Minimum dew point temperature in the last hour in
  ↪  celsius
    - name: humidity_max_perc
      dataType: number
      description: 'Maximum humidity percent in the last hour, from 0 to
  ↪  100'
    - name: humidity_min_perc
      dataType: number
      description: 'Minimum humidity percent in the last hour, from 0 to
  ↪  100'
    - name: humidity_perc
      dataType: number
      description: 'Humidity percent at reading time, from 0 to 100'
    - name: wind_direct_grad
      dataType: number
      description: Direction of the wind in positive degrees
    - name: gust_max_m_s
      dataType: number
      description: 'Maximum gust speed in the last hour, in meters per
  ↪  second'
    - name: wind_speed_m_s
      dataType: number
      description: Wind speed at reading time in meters per second
    - name: region
      dataType: text
      description: Brazil region code of the weather station
    - name: state
      dataType: text
      description: Brazil state code of the weather station
    - name: station
      dataType: text
      description: Weather station name
    - name: station_code
      dataType: text
      description: Weather station code
    - name: lat
      dataType: number
      description: Weather station latitude
    - name: lon
      dataType: number
      description: Weather station longitude
    - name: elev
      dataType: number
      description: Weather station elevation
```

```
quality:
  - id: unique_index_rule
    name: Rule to ensure having a correct primary key column
    dimension: uniqueness
    type: unique
    columns:  [ index ]
  - id: not_null_indexing_fields_rule
    name: Rule to ensure having not null fields that identify the record
    dimension: completeness
    type: complete
    columns: [ index, date, hour, region, state, station, station_code, lat,
↪   lon, elev ]
    parameter: 1
  - id: not_null_measurements_rule
    name: Rule to ensure that most of the measurements exist in the dataset
    dimension: completeness
    type: complete
    columns:
      - pcpn_mm
      - station_pressure_mb
      - pressure_max_mb
      - pressure_min_mb
      - temp_c
      - dewpoint_c
      - temp_max_c
      - temp_min_c
      - dewpoint_max_c
      - dewpoint_min_c
      - humidity_max_perc
      - humidity_min_perc
      - humidity_perc
      - wind_direct_grad
      - gust_max_m_s
      - wind_speed_m_s
    parameter: 0.95
    severity:
      warn:
        tolerance: 0.05
      fail:
        tolerance: 0.1
  - id: not_null_radiation_rule
    name: Rule to ensure radiation rules exist for half of the data
↪   (sun-time)
    dimension: completeness
    type: complete
    columns: [ radiation_kj_m2 ]
    parameter: 0.5
    severity:
      fail:
        tolerance: 0.1
  - id: gt_zero_measurements_rule
    name: Rule to ensure that weather measurements which can't be negative
↪   aren't present
    dimension: validity
    type: min
    parameter: 0
    columns:
      - pcpn_mm
      - station_pressure_mb
      - pressure_max_mb
      - pressure_min_mb
      - radiation_kj_m2
      - humidity_max_perc
      - humidity_min_perc
      - humidity_perc
      - wind_direct_grad
      - gust_max_m_s
      - wind_speed_m_s
  - id: max_percent_measurements_rule
    name: Rule to ensure that percent measurements don't go over 100%
    dimension: validity
    type: max
    columns: [ humidity_max_perc, humidity_min_perc, humidity_perc ]
    parameter: 100
  - id: max_degrees_measurements_rule
    name: Rule to ensure that degree measurements don't go over 360°
    dimension: validity
    type: max
    parameter: 360
    columns: [ wind_direct_grad ]
  - id: allowed_states_rule
    name: Rule to ensure only southeast regions are present in the dataset
    dimension: validity
    type: allowedValues
    columns: [ state ]
    parameter: [ SP, RJ, MG, ES ]
  - id: custom_date_rule
    name: Rule to ensure dates are in correct format
    dimension: validity
    type: custom
    technology: GreatExpectations
    call: expect_column_values_to_match_strftime_format
    columns: [ date ]
    args:
      strftime_format: '%Y-%m-%d'
  - id: custom_time_rule
    name: Rule to ensure times are in correct format
    dimension: validity
    type: custom
    technology: GreatExpectations
    call: expect_column_values_to_match_strftime_format
    columns: [ hour ]
    args:
      strftime_format: '%H:%M'

pricing:
  priceAmount: 0
  priceCurrency: USD
  priceUnit: fullScan

serviceLevelAgreements:
  intervalOfChange: 1h
  timeliness: 1m
  upTime: 0.999

specific: {}
```

A.1 Schema properties

A.1.1 General information

```
specVersion: "major.minor.patch" # Semantic versioning
# General information
id: "<name>-<producer.name>-<consumer.name>-<version>"
name: string
kind: "DataContract"
version: "major.minor.patch" # Semantic versioning
description: string
ownerGroup: string
tags: # OpenMetadata tags
  - tagFQN: string
    source: "Tag"
    labelType: "Manual"
    state: "Confirmed"

consumer:
  name: string
  group: string
producer:
  name: string
  group: string
```

```
access:
  eventType: push | pull | event | stream
  protocol: string
  location: string
  security:
    token: string
  accessConfiguration: {}

dataSharingAgreements:
  purpose: string
  security: string
  intendedUsage: string
  limitations: string
  lifeCycle: string
  confidentiality: string
```

field	required	type	description	example
specVersion	*	`string`	Version of the standard format	`0.1.3`
id	*	`string`	Id of the contract, composed by the contract name, the producer and consumer name, and the contract version	`data-contract-template--producer-dataproduct1-1.0.0`
name	*	`string`	Name of the contract	`Data Contract for ingestion`
kind	*	`"DataContract"`	kind of the yaml, it specifies it's a data contract	`"DataContract"`
version	*	`string`	Version of the contract, in semantic versioning	`1.0.0`
description		`string`	Description of the contract	`Contract for reading some data`
ownerGroup		`string`	Who owns the contract in the company	`dataproduct1_corp.com`
tags		`Tag`	Open Metadata format for list of tags	
consumer		`Actor`	Consumer of the data contract, maybe not be present if it's a multilateral contract	
producer	*	`Actor`	Producer of the data	
Actor.name	*	`string`	Name of the actor	`producer1`
Actor.group	*	`string`	Group where the actor belongs, might be just the email	`producer_othercorp.com`
access	*		Object that holds the information to access the consumer endpoint	
access.eventType	*	`pull | push | event | stream`	Type of data transfer. The initial scope is interested only in push events	`push`
acesss.protocol	*	`string`	Protocol to use to push the information in the system	`FTP`
access.location	*	`string`	URI endpoint to define the location of the system that will receive the data	`ftp://ec2.mylocation.com`
access.security	*		Object that includes the information to initially contact the consumer endpoint.	`"${env.token}"`
access.accessConfiguration	*	`object`	Object that includes specific information and configuration values to access the data, might be used to identify files, streams, etc.	`{"pattern": "*.csv"}`
dataSharingAgreements	*		Covers usage, privacy, purpose and limitations as defined in Agile Lab Data Product specification	

Table A.1: Data contract format specification: General information fields

A.1.2 Dataset schema

```
dataset:
  name: string
  schema:
    - name: string
      dataType: string
      description: string
      constraint: PRIMARY_KEY | FOREIGN_KEY | UNIQUE | NOT_NULL
```

field	required	type	description	example
dataset	*	`Dataset`	Contains the schema information of the dataset	
dataset.name	*	`string`	Name of the dataset	`employee_table`
dataset.schema	*	`[Column]`	Contains an array of the schema of the dataset columns	
dataset.schema.name	*	`string`	Column name	`employee_id`
dataset.schema.dataType	*	`string`	Data type on yaml data types, so it's technology agnostic	`string`
dataset.schema.description	*	`string`	Meaningful description of the column purpose	`ID of the employee`
dataset.schema.constraint	*	`PRIMARY_KEY | FOREIGN_KEY | UNIQUE | NOT_NULL`	Basic constraints on the column, is equivalent to some quality check definitions but in simple format	`PRIMARY_KEY`

Table A.2: Data contract format specification: Schema fields

A.1.3 Quality

```
quality:
  - id: string
    name: string
    columns: [string]
    technology: string
    dimension: completeness | validity | uniqueness
    type: string
    parameter: number | string | [string]
    scheduleCronExpression: string
    severity:
      warn:
        tolerance: number
      fail:
        tolerance: number
```

field	required	type	description	example
quality			Array of quality checks to be performed on the dataset	
quality.id	*	string	Name of the quality check	`quality_check_not_null`
quality.name	*	string	Human readable name for the check	`"Quality check that a column is not null"`
quality.columns		[string]	Array of column names if the quality check is applied at column level	`[employee_id, employee_salary]`
quality.technology		string	Extensibility to invoke external tools to perform the quality check. If null, the native check will be performed	
quality.dimension		completeness | validity | uniqueness	Data Quality dimension that is related to the current check	`completeness`
quality.type	*	string	Quality check to be done, refer to quality check types table to more info	`complete`
quality.parameter		number | string | [string]	Value to be used in the quality check. It may mean different things depending on the quality check `type`	`0.99`
quality.scheduleCronExpression		CronExpression	Quartz Cron expression to be used for runtime checks	`0 20 * * *`
quality.severity			Object that holds the information to calculate the severity of the check. If this is missing, every check that is not exactly equal to the expected `quality.parameter` will trigger a fail alert	
quality.severity.warn			Object that holds the information to trigger a warning alert for the check. If a warning is triggered, the data won't be rejected but a warning shall be created	
quality.severity.warn.tolerance	*	number	Tolerance percentage to which the `quality.parameter` should hold before launching a trigger.	`0.05` a.k.a 5%
quality.severity.fail			Object that holds the information to trigger a fail alert. If a fail is thrown, the data shall be rejected. The structure is the same as `quality.severity.warn`	

Table A.3: Data contract format specification: Quality rules fields

A.1.4 Pricing

This is directly taken from the PayPal data contract specification, with the addition of "Full Scan" as a price unit.

```
pricing:
  priceAmount: number
  priceCurrency: string
  priceUnit: MB | KB | GB | TB | PB | fullScan
```

field	required	type	description	example
pricing			Object that defines the pricing of the ingestion of data	
pricing.priceAmount	*	`number`	Subscription price per unit of measure in `priceUnit`.	`9.95`
pricing.priceCurrency	*	`string`	Currency of the subscription price	`USD`
pricing.priceUnit	*	`KB | MB | GB | TB | PB | fullScan`	Unit of measure for calculating cost	`fullScan`

Table A.4: Data contract format specification: Pricing fields

A.1.5 SLAs

```
serviceLevelAgreements:
  intervalOfChange: string
  timeliness: string
  upTime: number
```

field	required	type	description	example
serviceLevelAgreements			Object to specify the SLAs of the contract	
serviceLevelAgreements intervalOfChange		string	Description of the interval of change in which data is updated. It uses ISO8601 units	`1h30m`
serviceLevelAgreements. timeliness		string	Description of the timeliness of the data. It uses ISO8601 units	`10s`
serviceLevelAgreements. upTime		number	Uptime percentage of the data	`0.99`

Table A.5: Data contract format specification: Service Level Agreements fields

Appendix B

Data Contract Format Validation

Cue script for data contract format validation.

```
package datacontract

import (
  "strings" // a builtin package
)

#ParsedString: {#s: string, output:
↪  strings.ToLower(strings.Replace(strings.Replace(#s, " ", "-", -1), "_",
↪  "-", -1)) }


#Version: string & =~"^[0-9]+\\.[0-9]+\\..+$"
#ParsedName: =~ "^[a-zA-Z-]+$" & (#ParsedString & {#s: name}).output

#OM_Tag: {
  tagFQN:       string
  description?: string | null
  source:       string & =~"(?i)^(Tag|Glossary)$"
  labelType:    string & =~"(?i)^(Manual|Propagated|Automated|Derived)$"
  state:        string & =~"(?i)^(Suggested|Confirmed)$"
  href?:        string | null
}
#DataSharingAgreement: {
  purpose?: string | null
  billing?:         string | null
  security?:        string | null
  intendedUsage?:   string | null
  limitations?:     string | null
  lifeCycle?:       string | null
  confidentiality?: string | null
  ...
}
#Actor: {
    name: string & =~ "^[A-Za-z-_0-9]+$"
    group: string & =~ "^[A-Za-z-_0-9.]+$"
}
#Access: {
    eventType: "push"
  protocol: "FTP" | "HTTP" | "HTTPS"
  location: string
  security?: token: string
  accessConfiguration: {...}
}

#Dataset: {
    name: string
    schema: [... #OM_Column]
}
#OM_DataType:   string &
↪   =~"(?i)^(NUMBER|TINYINT|SMALLINT|INT|BIGINT|BYTEINT|BYTES|FLOAT|⌋
↪   DOUBLE|DECIMAL|NUMERIC|TIMESTAMP|TIME|DATE|DATETIME|INTERVAL|STRING|⌋
↪   MEDIUMTEXT|TEXT|CHAR|VARCHAR|BOOLEAN|BINARY|VARBINARY|ENUM|JSON)$"
#OM_Constraint: string & =~"(?i)^(NOT_NULL|UNIQUE|PRIMARY_KEY)$"
#OM_Column: {
  name:     string
  dataType: #OM_DataType
  if dataType =~ "(?i)^(ARRAY)$" {
    arrayDataType: #OM_DataType
  }
  if dataType =~ "(?i)^(CHAR|VARCHAR|BINARY|VARBINARY)$" {
    dataLength: number
  }
  description?:        string | null
  fullyQualifiedName?: string | null
  tags?: [... #OM_Tag]
  constraint?:      #OM_Constraint | null
  if dataType =~ "(?i)^(JSON)$" {
    jsonSchema: string
  }
  if dataType =~ "(?i)^(MAP|STRUCT|UNION)$" {
    children: [... #OM_Column]
  }
}
```

```
#Quality: [... #QualityRule]
#CompletenessTypes: "size" | "complete"
#UniquenessTypes: "unique"
#NumberValidityTypes: "min" | "max" | "mean" | "stdev" | "custom"
#ValidityTypes:  #NumberValidityTypes | "allowedValues"
#CronExpression: string & =~ "^(?:[0-9,*\/-LW?]+ ){5}[0-9,*\/-LW?]+$" //
↪  Basic regex, but most of the expressions won't be valid cron
#ColumnName: or([for column in dataset.schema { column.name }])
#Dimensions: "completeness" | "validity" | "uniqueness"

#SingleParameterValue: string | number | int
#ListParameterValue: [... string] | [... number]
#ParameterValue: #SingleParameterValue | #ListParameterValue

#BaseRule : {
    id: string
    name: string
    columns?: [... #ColumnName] | null
        dimension: #Dimensions
        type: string
        scheduleCronExpression?: #CronExpression | null
    severity?: {
        warn?: tolerance: number
        fail?: tolerance: number
    }
    parameter?: _
    technology?: _
    call?: _
    args?: _
}

#CompletenessRule : #BaseRule & {
        dimension: "completeness"
        type: #CompletenessTypes
        [
                if type == "size" {
                    parameter: int
                },
                if type == "complete" {
                    parameter: number
                }
        ][0]
        technology?: null
        call?: null
}

#ValidityRule: #BaseRule & {
        type: #ValidityTypes
        dimension: "validity"
        [
                if type == "custom" {
                        call: *id | string
                        technology: string
                        args?: [string]: #ParameterValue | null
                },
                if type == "allowedValues" {
                        parameter: #ListParameterValue
                },
                {parameter: int | number}
        ][0]
}

#UniquenessRule: #BaseRule & {
        dimension: "uniqueness"
        type: #UniquenessTypes
        technology?: null
        call?: null
        parameter?: null
}

#QualityRule: #CompletenessRule | #ValidityRule | #UniquenessRule

#Units: "KB" | "MB" | "GB" | "TB" | "PB" | "fullScan"
#Pricing: {
    priceAmount: number & >=0
    priceCurrency: string
    priceUnit: #Units
}

#Interval: string & =~ "^([0-9]+Y)?([0-9]+M)?([0-9]+d)?([0-9]+h)?([0-⌋
↪  9]+m)?([0-9]+s)?([0-9]+ms)?$"
#SLAs: {
    intervalOfChange: #Interval
    timeliness: #Interval
    upTime: number & <= 1.0 & >= 0.0
}

#ProducerName: string & (#ParsedString & {#s: producer.name}).output
#ConsumerName: string & (#ParsedString & {#s: *consumer.name|""}).output

specVersion: "0.1.3"
// General information
id: string & =~ "^\(#ParsedName)-\(#ProducerName)-\(#ConsumerName)\([if
↪  consumer != _|_ {"-"},{""}][0])\(version)$"
name: string
kind: "DataContract"
version: #Version
description: string
```

```
tags: [... #OM_Tag]
consumer?: #Actor
producer: #Actor
access: #Access
dataSharingAgreements: #DataSharingAgreement
dataset: #Dataset
quality: #Quality
pricing?: #Pricing | null
serviceLevelAgreements?: #SLAs | null
specific: {...}
```