José Miguel Dias Simões

POWER Data Framework Architecture

1290 UNIVERSIDADE D COIMBRA

UNIVERSIDADE D
COIMBRA

José Miguel Dias Simões

POWER Data Framework Architecture

Dissertation in the context of the Master's in Informatics Engineering, specialization in Software Engineering, advised by Prof. Bruno Cabral and Prof. Vasco Pereira and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

José Miguel Dias Simões

POWER Data Framework Architecture

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Bruno Cabral and Prof. Vasco Pereira and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

José Miguel Dias Simões

POWER Data Framework Architecture

Dissertação no âmbito do Mestrado em Engenharia Informática, especialização em Engenharia de Software, orientada pelo Professor Doutor Bruno Cabral e pelo Professor Doutor Vasco Pereira e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

September 2023

Acknowledgements

I would like to thank Professor Bruno Cabral and Professor Vasco Pereira for their supervision, understanding and patience during the project's execution, for providing me with the tools, resources and validation needed to see this work through.

This work could not have been possible without the support of my family, especially my partner, Inês, who helped me through the lulls and stalemates, providing me with the motivation I needed to pull through and reach my objectives, accompanying me every day as I progressed through the writing and development of this dissertation. As the days dragged on, and difficulties mounted, her fighting spirit and tireless work ethic served as beacon of inspiration, and I know I could not have done it without her.

I thank my mother, who enabled me to reach this stage, through her love and sacrifice over many hardships along these University years. I can only hope she's as proud of me as I am of her.

I also extend my thanks to my dear friends who have accompanied me since the freshman years (and some even before that): my *brothers-in-arms* Rodrigo, Braga and Diogo, who sat with me many times discussing my work, helping me face my weaknesses as the project developed through its more tumultuous stages; Nuno, who provided a constant stream of motivation through his tirelessness and perseverance in his PhD project; Henrique, Inês and Mónica for their companionship in the late nights at the department; and a massive thanks to my good friend Bib, for providing some much-needed *tough-love* in my many slumps.

I would also like to extend my thanks to Altice Labs S.A. and our contact, Luis Cortesão, for their availability and for providing the information and feedback necessary for this project.

And lastly, I would extend my thanks to the members of the jury for their evaluation, feedback and judgement of my work, which was instrumental to the conclusion of this dissertation.

This work is funded by the project POWER (grant number POCI-01-0247-FEDER-070365), co-financed by the European Regional Development Fund (FEDER), Portugal 2020 (PT2020), and by the Competitiveness and Internationalization Operational Programme (COMPETE 2020).

Abstract

This document describes the work carried out by José Dias, in the scope of the "POWER Data Framework Architecture" internship promoted and hosted by the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra , in partnership with Altice Labs S.A., as part of the Informatics Engineering Master's course.

The objective of the internship is to specify a software architecture which meets the current business goals and needs of Altice Labs' data management processes, while also providing a stable foundation for future business growth into an IaaS or PaaS context and the adoption of novel data management and governance techniques and practices.

To provide the background for the architecture design process, novel approaches were analysed and reviewed. Following this brief contextualization, the preliminary requirement analysis process was initiated, resulting in a requirement specification to enable the analysis of architectural drivers. The conclusion of this preliminary step was the drafting of an architecture which met the functional and non-functional requirements albeit without experimental validation.

To evolve from the draft and mature the architecture into a fully validated and complete specification, an iterative methodology was followed - an adapted version of the Architecture-Centric Design Methodology (ACDM). In each iteration the architecture and requirements were refined and subsequently evaluated experimentally and theoretically.

The architecture was developed through two iterations of the design process, resulting in an experimentally validated data ingestion, storage and serving pipeline under the Lakehouse paradigm, and in a partially validated governance and administration platform. The project resulted in a requirement specification and architecture specification, which will be submitted as deliverables for Altice Labs S.A..

Keywords

Data Architectures, Big Data, Multi-Tenancy, ACDM

Resumo

Este documento descreve o trabalho realizado por José Dias, no âmbito do estágio "POWER Data Framework Architecture", promovido pelo Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologias da Universidade de Coimbra, em parceria com a empresa Altice Labs S.A., como projeto de dissertação do Mestrado em Engenharia Informática.

O objetivo do estágio é desenhar uma arquitetura de software que cumpra os objetivos e necessidades atuais dos processos de gestão de dados da Altice Labs, construindo uma base estável para o crescimento futuro do negócio num contexto de IaaS ou PaaS, bem como a implementação de novas técnicas e práticas de gestão e *governance* de dados.

Para contextualizar o processo de desenho da arquitetura, abordagens inovadoras foram analisadas. Após esta contextualização, o processo de análise e engenharia requisitos foi iniciado, resultando numa especificação de requisitos que permitiu a análise dos *drivers* arquiteturais. A conclusão desta etapa preliminar foi a elaboração de um rascunho da arquitetura que cumprisse os requisitos funcionais e não funcionais, ainda que sem validação experimental.

Para evoluir a partir do rascunho e progredir do rascunho para uma arquitetura validada e completa, foi utilizada uma metodologia iterativa - uma versão adaptada do "Architecture-Centric Design Methodology" (ACDM). Em cada iteração, a arquitetura e os requisitos foram refinados, validados e posteriormente avaliados experimental e teoricamente.

Foram realizadas duas iterações do processo de desenvolvimento da arquitetura, resultando numa *pipeline* de ingestão, armazenamento e disponibilização de dados validada experimentalmente no paradigma *Lakehouse*, e numa plataforma de governança e administração parcialmente validada. O projeto resultou numa especificação de requisitos e numa especificação de arquitetura que foram entregues à Altice Labs S.A..

Palavras-Chave

Arquitetura de Dados, Big Data, Multi-tenancy, ACDM

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, the introduction of the dissertation project is performed, starting by presenting a brief description of the context for the elaboration of this work, followed by a detailed look into the motivations and purposes behind the work to be detailed, as a prelude to the description of the objectives of this dissertation. Lastly, the document's structure and contents are outlined.

1.1 Context

This dissertation was written in the scope of the "Dissertation/Internship in Software Engineering" curricular unit of the Master's Course in Informatics Engineering, at the Department of Informatics Engineering, in the Faculty of Sciences and Technology of the University of Coimbra.

Altice Labs S.A. [7] is a research and development company, starting in the 1950s as a telecommunications (*telecom*) technology development initiative, providing numerous innovations through the application of new, advanced electronics technology to the Portuguese communications landscape. Presently, they are exploring the application of new technological patterns to the areas of home internet, IoT and 5G, all while optimizing and improving existing *telecom* services.

Altice Labs, as lead promoter in partnership with the **University of Coimbra**, the **Institute of Telecommunications** [76] and the **Pedro Nunes Institute** [77], is part of "**POWER - Empowering a Digital Future**" [43], a national project which seeks to create an innovative portfolio of products and services through the use of emerging technologies (such as 5G, Data-driven business models and Artificial Intelligence). The aim of this project is to create a new architecture to serve as the foundation for a new data governance framework for Altice Labs S.A. to allow for optimal use of the new research and development efforts within Project POWER.

The project started on the 13thof September of 2022, and concluded on the delivery date of the 5thof September of 2023, with an intermediate delivery on the 16thof January 2023.

1.2 Motivation and Purpose

Data is an increasingly important part of the digital landscape in the 21stcentury. In the age of Big Data, the unprecedented rate of data generation, collection and storage that we experience today, has led to challenges and bottlenecks in processing, especially given that the nature of these massive data flows often makes conventional systems and techniques no longer applicable due to their limitations in processing power or due to their lack of flexibility in regards to data variety [47].

Businesses worldwide have had the opportunity to collect large amounts of data to meet their business objectives and provide better service. The market for enterprise Big Data has been growing fast with an annual growth rate of 13.4%, being valued at 240.56 thousand million US dollars in 2021, and is expected to reach 655.53 thousand million USD by 2029 [64].

The use of data to enrich and improve services has an indefinite number of applications, due to the increasing ability to infer and observe patterns from low-information data sources, especially by leveraging Machine-Learning and other statistical computing methods [36, 85].

This growth inevitably brings about problems, not only due to the direct scalability issues but also in terms of the evolution of the supporting infrastructure and operational requirements to handle the issue of data governance, which involves the exercise of authority and control over the use, quality and storage of said data [1].

The main challenge can be summarized in being able to receive the large, ever growing amounts of data, finding a way to store it and process it to make it useful, all within growing security constraints [38] and an ever-evolving technological and regulatory landscape [97, 102].

Software engineers have designed a variety of reference architectures and patterns to guide the development of systems which can provide the necessary functionalities and qualities for modern enterprise applications. There are studies in the literature which introduce these architectures from a technical standpoint [67, 82] and there are also studies which seek to analyze and deconstruct the problem from an organizational standpoint [87], promoting practices and concepts to address the critical issues related to data governance.

Despite the large amount of new work and developments [102], the challenge of finding a correct methodology remains, and much research is dedicated to creating stable frameworks for data governance and high-volume Big-Data processing pipelines, leveraging the benefits of cloud computing [37] and highly-efficient, cost-effective stream processing that can adapt to increasingly strict requirements regarding data privacy, security and quality.

Recent technological developments (i.e. *Data Lakes* [65, 93], *Lakehouses* [32], etc.) have created an environment that is rich in potential solutions, creating an opportunity to optimize and tailor-make these architectures to fit individual needs,

requirements and business objectives and companies today may explore these innovative solutions and create data ecosystems for themselves and for their clients.

As one of the tasks of the **Project POWER**, and accompanying the research and development effort within the other sub-projects, namely the ones regarding the adoption of new organizational frameworks and data governance strategies, the development of an underlying infrastructure to support all the business goals, present and future, of Altice Labs is of utmost importance.

This goal can be achieved through the creation of a flexible, adaptive and scalable framework, the process of large-scale data analytics can be made more streamlined, secure and flexible, enabling the adoption of new organizational paradigms (such as domain-based access, self-serve data governance, etc.) as well as more business-oriented goals, such as the planned transition from an internal solution to an Infrastructure-as-a-Service (IaaS)/Platform-as-a-Service (PaaS) context using multi-tenancy.

The process of designing the architectures to support these services is increasingly important due to the growing complexity, scaling needs and performance requirements of such systems. Considering the scope of Altice Labs' business model and the end goal of producing an effective data management framework, a software architecture must be designed to account for all the relevant business needs, future infrastructure requirements, scalability concerns and regulatory constraints.

1.3 Objectives

Taking into account the issues previously mentioned as well as the potential for an new solution fit for Altice Labs' goals, the objectives for this internship can be defined as follows:

1. Research **technologies** and **solutions** for a cloud-based Big-data platform with IaaS capabilities.
2. Perform a **comprehensive requirement analysis** for this platform.
3. **Draft** and iteratively **refine** an **architecture proposal** to support the identified requirements, while **evaluating** the architecture's main attributes experimentally and theoretically.
4. Present a fully **documented**, justified and **evaluated architecture specification** for the Altice Labs Data Framework.

The first semester covered the first two objectives, resulting in an initial **Requirement Specification**, following the IEEE 830-1998 Standard of recommendations for Software Requirement Specifications [72], and in the production of a **draft architecture proposal**, described using a summarized component view.

During the second semester an iterative refinement and evaluation methodology, inspired by the **Architecture-Centric Design Method** (ACDM) [74, 83], was followed to fulfill the last three objectives, resulting in the fully detailed Architecture Specification following the **"C4" Architecture View Model** [105] for architecture descriptions, as well as the detailed supporting documentation (e.g. experimental record, validation).

1.4 Planning and Execution

The dissertation project was split into two halves - the first and second semester.

The first half was dedicated to defining the requirements for the system and creating a draft for the architecture. Although the tentative prototype was designed, the lack of a structured approach hindered its quality, forcing a re-structuring of the project's planning, documentation and review processes by the time of the intermediate defense.

The second half emphasised the iterative development methodology to leverage a systematic review and a more organised approach in order to build from the draft and create a more robust, validated and correct outcome.

Progress drastically increased once the refinement and review processes were adopted in earnest, and the validation of the architecture was improved dramatically by the introduction of experiments.

Unforeseen hardware limitations led to a delay in the project's experimental validation, causing the postponing of the final delivery from July to September to overcome these challenges and complete the experimental validation to ensure adequate coverage of the requirements.

While it was not possible to complete the experimental validation, the work was carried out as far as possible within the project's constraints, and presents good coverage of the functional and administrative requirements for the system, with a clear indication of future work and a solid documentation base for future efforts relating to this project.

1.4.1 First Semester

As described, the first semester encompassed the following objectives:

- Perform research on data management patterns, models and architectures
- Requirement Engineering and Specification
- Architectural Driver Specification
- Creation of a notional architecture (draft or quick prototype)

These objectives were broken down into six separate tasks:

- **Contextualization** - Understand the problem, perform preliminary research.
- **Planning** - Coordinate the task flow and work flow for the semester.
- **Requirements** - Requirement engineering, elicitation and specification.
- **Architecture Draft** - Produce a tentative "first-version" architecture.
- **Research** - Perform research on data management architectures/frameworks.
- **Report** - Document the process in the dissertation document.

A "*waterfall*" approach was selected to build the foundation for the future design work, owing to the successive nature of the Requirement, Architecture Draft and Report tasks. A large focus was placed on the requirement engineering processes to ensure that the bulk of the requirement work was completed prior to the architecture draft's creation.

To this end, planning was oriented toward contextualizing the problem, and structuring the project around the architecture design methodology which was selected for this dissertation project. The simple Gantt Chart in Figure 1.1 seeks to detail the work schedule on a task level, according to the outlined objectives.

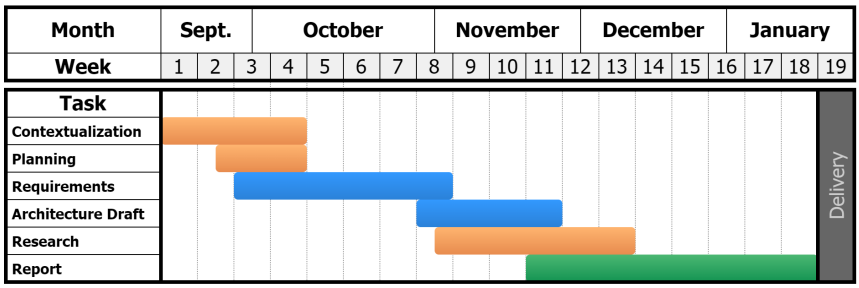

Figure 1.1: Gantt Chart detailing the work plan for the first semester. The "waterfall" approach is clearly visible.

The objectives were met, with a validated requirement specification and a draft of the architecture, featuring a notional approach, which sought the serve as a foundation for a more concrete and systematic process in the second semester. The research process was also successful in identifying solutions that could fit into the mold of the project, and building a contextual base for the architectural decisions to follow.

The workflow in the first semester was tumultuous but, ultimately, provided the necessary contextualisation and knowledge required to progress into a successful second semester. The executed work was analysed and summarised in Figure 1.2, indicating the aforementioned issues.

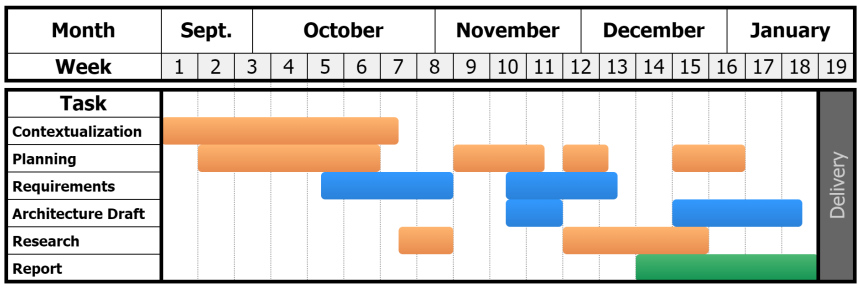

Figure 1.2: Gantt Chart detailing the executed work for the first semester. The numerous attempts to re-plan are visible, as well as the scattered, unstructured nature of the work.

1.4.2 Second Semester

In the second semester, an effort was made to correct the mistakes of the first semester, namely the unstructured nature of the work, especially considering the iterative design methodology called for a highly structured approach. With the feedback from the intermediate presentation, corrections to the report were necessary, and a change of pace was needed. The tasks for the second semester consisted in:

- **Planning** - Plan the work, task deadlines.
- **Document Corrections** - Correct the issues in the dissertation report, the requirement specification and clean up the documentation for the architecture draft.
- **Iteration #1** - Analyse the draft, refine it, verify if it meets requirements and run experiments to validate the architecture.
- **Iteration #2...N** - Repeat the process described for iteration #1 as many times as necessary to validate the architecture.
- **Architecture Spec.** - Develop an architecture specification to submit as a deliverable following the **C4 model** [39].
- **Requirement Spec.** - Develop a requirement specification document to submit as a deliverable following the **IEEE 830-1998 Standard** [72].
- **Report** - Conclude the dissertation report, describing the iterative methodology and the development of the final architecture.

The planned structure for the semester was as described in the Gantt Chart in Figure 1.3, along with the executed work scheduling.

In summary, the plan underestimated the amount of corrections necessary, and the first iteration was slightly longer than previously estimated. After the first iteration, some new corrections were necessary, which had been unplanned for, before the second iteration could proceed.

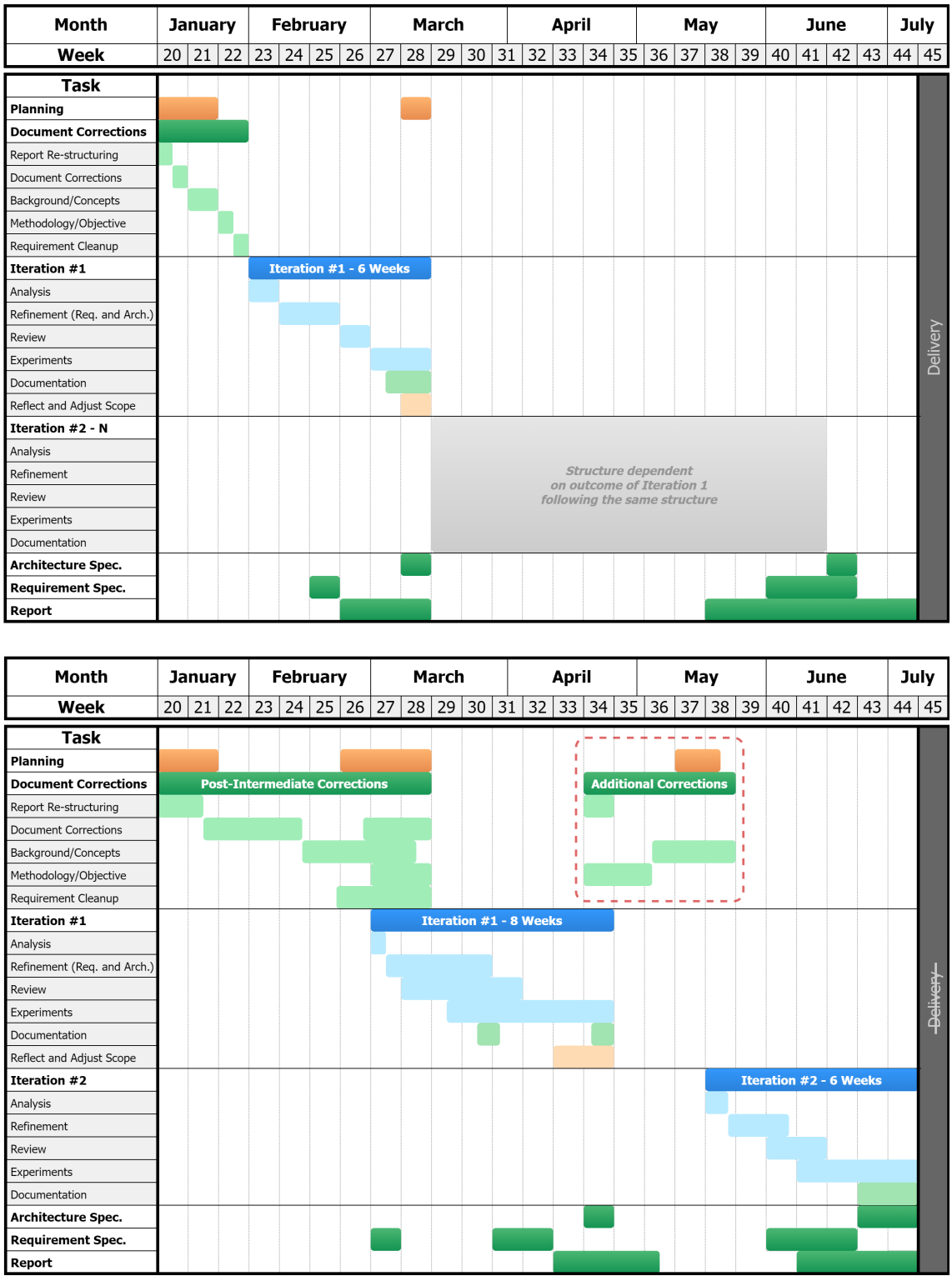

Figure 1.3: Gantt Chart detailing the planned work for the second semester (top) and the executed work (bottom). The grey block is a placeholder.

Despite these challenges, a more refined architecture was developed. Some work was unfinished, namely related to the Architecture Spec. deliverable, which was left unfinished due to gaps in the experimental validation stemming from technical difficulties. Additionally, the monitoring components, which had been de-

prioritized, were still in need of validation, calling for a possible third iteration.

These unfinished facets of the project motivated a delay from the delivery date of the 10thof July to the 5thof September. This additional period, its planning and execution records are discussed in the following sub-section.

1.4.3 Delay - Summer

As previously mentioned, three objectives remained upon delaying the delivery to September:

- **Experimental Validation of Iteration #2**- Hardware limitations caused the experimental validation to be incomplete. Additional hardware was acquired to assist.
- **Possible Iteration #3** - Monitoring-centric architecture developments.
- **Architecture Specification** - Without the confirmation that the components were indeed suited for the architecture, the specification was also incomplete pending these results.

To account for this, the work plan was drafted to finish up the experimental validation as soon as possible with the intent to initiate a *third iteration* which would complete the architecture (which up until this point had been developed in a *functional* and in an *administrative* perspective) with a monitoring/management focus. Figure 1.4 presents the planned and executed work scheduling.

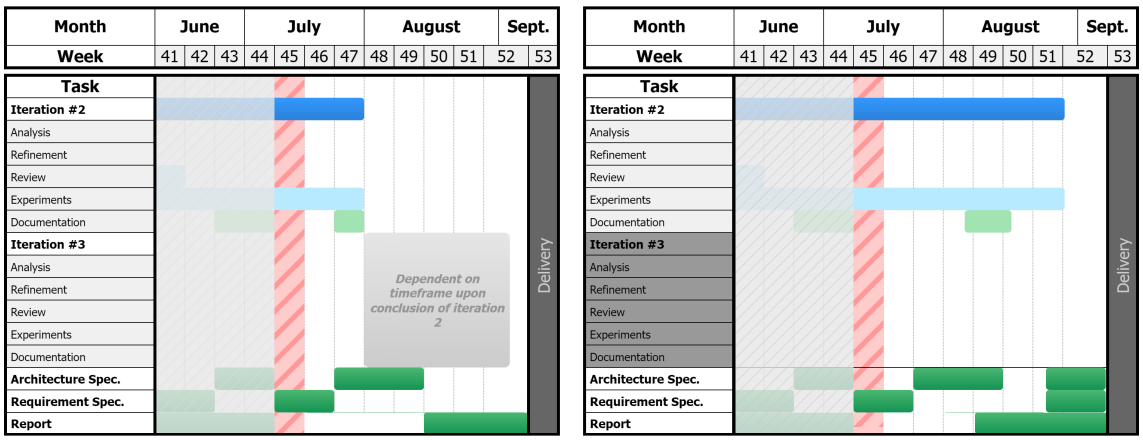

Figure 1.4: Gantt Chart detailing the planned (left) and executed (right) work schedules for the summer dissertation delivery delay period.

During this time the experimental validation was attempted using more powerful hardware but was not completed successfully in time for the submission. This resulted in a priority shift over to improving the final deliverables with revised diagrams and textual descriptions and cleaning up any trailing documentation to close out the iterative process and document the resulting architecture for future work, rather than attempting to rush a third iteration.

1.5 Document Structure

This chapter serves as a preface to the remaining content of the document, providing an outline of the context, motivation and purpose behind the production of this document and of the project as a whole, as well as detailing the objectives and planning for the project.

The second chapter covers essential context and groundwork for the architecture's development. It includes an overview of modern data management concepts, an examination of current models, systems and architectures, as well as a description of novel governance-centric approaches.

The third chapter will detail the methodologies followed in the different steps of the project. This description seeks to cover the two central steps of the project's process - requirement analysis and architecture design - and will explicit the strategies used for requirement elicitation, analysis and specification and the techniques employed in the architecture design process, the Architecture-Centric Design Methodology (ACDM). Additionally the project management methodologies which support the architecture development process are described.

The fourth chapter will present the state-of-the-art studied for this project - implementations of the prevalent modern data architectures, a survey on the supporting technologies and a case study of a cloud-native production-ready architecture.

The fifth chapter will provide a condensed requirement specification, providing a description of the system and its components, followed by a brief overview of the system's requirements and architectural drivers.

The sixth chapter will serve to document the process of architecture design following the ACDM, starting from the initial draft until the final version. Each section will encompass one full iteration, detailing the analysis, refinement, validation and experimentation processes which take place within it.

The seventh chapter will describe the final architecture. This description will inform on the framework architecture's final state.

The final chapter serves as the conclusion of the report.

Appendix A contains the Requirement Specification for the project and Appendix B contains the Architecture Specification, produced as deliverables for the project.

Chapter 2

Background and Concepts

Data is the central unit of information exchange in the digital age. It is a signal, a fact or a measure which is transmitted and serves as the carrier of information, which enables the building and dissemination of knowledge across the various disciplines, actors and businesses in the modern world. The market's perspective on the value of harnessing data has remained the same since its inception - collect, analyze, optimize - from the initial uses of business intelligence and data-analysis, such as task automation and cost-cutting [109], to the use of numerical and statistical methods to guide decision-making in businesses, to the modern notion of data-as-a-product and the focus on large scale data analysis as the key to improving value and quality.

While these changes could be analyzed on their own, from a conceptual and business management standpoint, to assist in the development of a solution it may be helpful to review the evolution of the supporting digital infrastructure, which has enabled the growth of the data market and provided the tools for the widespread use of data in the modern world.

To this end, a study of the data management landscape was performed, encompassing the analysis of the fundamental concepts and drivers of the field as well as the underlying elements which support the data-centric software architectures, in order to obtain contextual information and build a knowledge base to make informed decisions when designing the architecture to be used in Altice Labs' use-cases.

This chapter starts with a brief description of the core concepts and drivers of the data management landscape - the activities associated with data, measures of data and data governance. Following this description, the main data storage systems and models are presented from a chronological, market-driven perspective - the Database, the Data Warehouse, Lake and Lakehouse. Following this analysis is a presentation of the main architecture patterns - Lambda and Kappa - and how their approaches tackle the Big-Data processing paradigm. Lastly, novel approaches - Data Fabric and Data mesh - are presented, detailing their architecture, purposes, advantages and drawbacks.

2.1 Data Management Concepts and Drivers

To frame the presentation of data architectures and their components, it is crucial to first understand the key drivers that influence the growth of the data market and, by consequence, the development of the underlying digital infrastructure. These principles and concepts are what fuel the constant innovation, development and research attached to the topic of data management at several scales.

While data is a topic which can be described at great lengths and has been the focus of discussion for many years [104], it is more relevant for this project to aim research at the properties and concepts surrounding the management of data in the current enterprise landscape - the era of Big-Data.

For lack of formal definition, the Big Data era can be described as the current paradigm of data processing, involving very large bodies of data (the so-called big data) that cannot be handled or analysed by traditional data processing software due to its scale, processing or throughput limitations. Though it is not agreed upon when data can be considered *big* or how the paradigm itself can truly be defined, some proposals have appeared to provide a stable foundation upon which to build more research, providing **measures** of "big-ness" which can categorize data-centric activities [47, 57, 81, 108], and provide insight into the key drivers for software development in the field.

Analyzing these proposed traits, three distinct areas of concern emerge when discussing the **data as a business object**:

- **Quantitative Measures** - Related to the *physical* traits of data. Included measures have been defined as *Volume*, *Variability*, *Variety*, *Complexity*. These measures generally account for performance, scalability and availability design questions.
- **Qualitative Measures** - Based on the information within the data, defined in some sources as *Value*, *Veracity* or *Quality*. These measures can present questions to both the technology (integration, compatibility, correctness) and to the organizational structures surrounding it (auditability, traceability, etc.).
- ***Normative* Measures** - Related to the ethical, legal and social aspects of data management. These measures account for traceability, auditability and compliance-oriented design questions that, like the qualitative measures, can encompass both the infrastructure and the organizational structure itself.

The early days of the data market were focused mostly on the *quantitative* and *qualitative* measures of data, displaying massive evolution in the areas of *statistical analysis*, *data analytics* and *business intelligence* - the use of data to further business operations through direct analysis. This use kept growing as more technologies came along to support the market, allowing for large volumes of data analysis in the digital space.

With this tremendous growth in the late 1990s and early 2000s, regulation crept toward regulating this massive business activity which, due to technological advancements (namely mobile networks [110] and the proliferation of the internet [60]), was now able to collect sensitive data on an unprecedented scale, and possibly interfere with the privacy of individual customers. Social awareness grew with high-profile incidents, data leaks and privacy breaches, and the demand for increased attention to the normative measures of data became clear.

What followed was the creation of extensive regulation (and rapid evolution of existing regulations) regarding the use of private and sensitive data. The **General Data Protection Regulation (GDPR)** [61, 62], and other regulations (such as the *CCPA* in the U.S.A., *PIPEDA* in Canada and the *DPA* in the U.K. among others) present strict requirements for data-driven business operations and their software, which naturally results in a re-prioritization for businesses, software developers and architects alike to focus on measures of privacy, auditability, traceability and security, instead of merely being concerned with increasing the value of data. Additionally, upcoming legislature regarding the use of Artificial Intelligence and Machine Learning in business operations [63] (namely those operating on sensitive data) will bring further challenges to AI-driven Big-Data frameworks which have grown to unprecedented scales.

Under these constraints, the market and the scientific community in general have dedicated an increasing amount of attention to these *normative* measures [40] (Image 2.1), which relate to the to the management of data origins, lineages and tracking, traceability and life-cycle of data objects.

It is in this scope that the concept of **data governance** is born, merging the *qualitative* and *normative* measures to encompass the management of the usage and quality of data, and adapt solutions to a the current regulatory landscape. It can be defined as the creation of guidelines which pertain to how data is **stored, processed, used and disposed of**, all while maintaining utility to the business operations it pertains to.

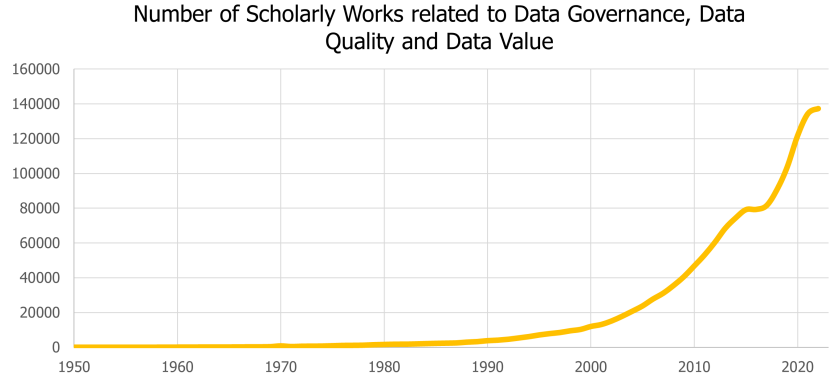

Figure 2.1: Evolution of research volume dedicated to the topic of Data Governance and its ancillary data measures. Results use the topic-search function at "The Lens" to analyse article volume for keywords "Data Governance" OR "Data Quality" OR "Data Value", displaying a marked increase in the 2010s.

In this section, some of the major concepts associated with data management

will be presented, in order to shed light on the foundation of data framework development:

- **Data Activities** - Describing, succinctly, what activities may take place in the modern Big-Data paradigm through the use of large amounts of data. Topics of *Business Intelligence (BI), Data Science (DS)* and *Machine Learning (ML)* are covered.
- **Data Structure and Flow** - Pertaining to the formats of data, the many types and categories of data items, flows and sources. Topics of *structured and unstructured data* and corresponding *formats* are detailed.
- **Data Processing** - Detailing the typical use-case for data transformations in the business landscape and what operations these transformations may contain. Topics of *Batch/Stream Processing* and *Extract-Transform-Load (ETL)* operations are discussed.
- **Data Quality** - Detailing concepts of data quality and value, integration and enrichment and why these qualities matter to a business. Some categorizations are discussed, along with *data cleansing* practices.
- **Data Governance** - Pertaining to data management in the modern business landscape, through the analysis of the key elements of the control and orchestration of massive data ops in an increasingly strict regulatory landscape.

2.1.1 Data Activities

Data, in the business sense, is often looked at as a carrier of intrinsic value to the business it pertains to. In this optic, starting from the very inception of the idea of a business, data has been harnessed as a way to **gauge the successes and failures of a commercial activity** [109] and **inform on business decisions**. Through recent developments, data can also be looked at as a source of indicators of future business needs, requirements and goals, through the use of widespread Machine Learning and Artificial Intelligence techniques to search for patterns in massive amounts of data.

Two main areas of activity are generally used to define the Big-Data use-cases:

- **Business Intelligence (BI)** - The use of techniques, tools and models to report on business activities, create business information and drive external and internal decision-making. These activities often rely on the structure and organization of the underlying data to perform simple and fast statistical analysis, resulting in a rapidly obtained output which can describe numerous traits of the underlying business activities.
- **Data Science (DS)** - The use of algorithms, statistics and scientific methods to attempt to extract/extrapolate knowledge and insights from data's underlying patterns. This use is typically attached to exploratory analysis

and methods, with the intent to use large-scale automated processing and analytics[1] pipelines to perform computations that would be unfeasible for standard direct analytics technology.

Business Intelligence, as its name suggests, is typically applied to the streams of data originating from business activities, providing a greater level of understanding of the state of affairs through the use of complex analytical methods. Generally BI initiatives result in dashboards, reports and easily communicable artifacts, which are useful not only internally but also externally, providing easy communication with stakeholders[2].

Data Science activities typically rely on advanced statistical algorithms, namely those developed by **Machine Learning (ML)** and **Artificial Intelligence (AI)** initiatives. ML's approach creates *models* which are essentially algorithms that *learn* (through pattern inference) from data to make predictions or decisions on future data without being explicitly programmed to do so.

Both these activities can be undertaken as services to external entities (serving reports to customers, performing data analytics, etc.) or as internal support for business decision-making. They rely on a constant supply of data in order to maximise their usefulness, and, as will be presented in the following sub-sections, rely on certain traits, processing pipelines and on the value of the data objects they manage.

2.1.2 Data Structure and Flow

Data, in the business sense, has evolved from paper spreadsheets to massive database tables and binary storage to meet the demands of the growing data market. In the current landscape, Big-Data systems have to harness many data sources, often times of wildly varying formats, structures and semantics. Typically, most modern descriptions involve three main types of data [47]:

- **Structured Data** - *i.e. Transactional data, database entries and records and tables, AVRO, Parquet, ORC* - This data possesses a rigid structure or *schema*[3] which facilitates querying and interpretation.
- **Semi-Structured Data** - *i.e. Logs, XML files, JSON files...* - While these files may not fit into a traditional database structure, they nonetheless possess tags and identifiers which allow for simplified consumption processes.
- **Unstructured Data** - *i.e. Natural language, images/video, metadata records...* - These files do not possess an easily identifiable structure and must be analyzed through more complex algorithms in order to provide their information.

[1] Analytics is the process of discovering, interpreting, and communicating significant patterns in data.

[2] A stakeholder is someone with an interest in a project or organization.

[3] A structured blueprint defining the organization, structure, and relationships of data within a database or data system, typically used to ensure correctness and facilitate interpretation.

Data formats conform to these categorizations. It is important to note that the discussion of data structure typically involves the underlying information of the data and how it can be access and processed. While most data types (structured or otherwise) could have their binary code accessed and analysed, the meaningful content within is typically only expressed through the interpretation of this binary data using some form of **rule or structure**.

Structured data will often find formats that present a strong definition of form (through enforced structures known as *schemas*) included within the file itself, encompassing the data object's structure and interpretation metadata. Examples include *Avro*, *Parquet* and *ORC* [44].

Semi-Structured data will hold the data and some information regarding its own structure [3]. This type of data can also be called *self-describing data*, if it holds the instructions to build an interpretation schema and to eventually parse the file's contents into an organised format. Example formats include *XML*, *JSON* and *Log* files, which hold *some* of the file's structure in their metadata. The remainder (e.g. schema information, headers) is typically either described in the data itself or included as an auxiliary file (e.g. index files, externally supplied schemas).

Unstructured data formats are typically either strict binary representations or some other type of file where there is fundamentally no internal organisation to the data contained within. While images, video and audio files do have the interpreting structure attached to them, the data within is not indicative of the contents of the file - of the *information* within.

And these data objects are typically moved from source to destination in one of two ways:

- **Batch Flows** - Data is collected up to a certain point, upon which it is bundled and sent to the destination. This is the standard for systems where high data volume processing is required, but real-time requirements do not exist.

- **Stream Flows** - Data is transferred continuously as a stream of individual records, allowing for real-time analytics on data, creating unique opportunities, but at a large cost due to the unpredictable nature and scalable requirements of real time streaming technology.

The systems which support the storage of large amounts of data needed to run a large-scale data operation are described in a detailed architectural perspective in Section 2.2.

2.1.3 Data Processing

There are many ways to process data, ranging from individual handmade analysis, to massive constant realtime processing pipelines, running automatically in

remote servers. In the current paradigm, two main processing method descriptions (heavily reliant on data flow structure) are typically employed to discuss the nature of a processing pipeline:

- **Batch Processing** - The application of massive data use, transformation and analytical operations on non-realtime sets of data records - *batches*. Typically relies on optimising operations to run on large data-sets and scale efficiently based on batch-size.
- **Stream Processing** - The application of realtime analytics and data operations performed on data streams, working record-by-record to ensure a steady flow of data in varying conditions.

These two methods are intimately tied to the business activity they are attached to. **Stream Processing** is typically used with data which has continuity requirements i.e. critical systems, monitoring, etc. **Internet of Things (IoT)** processing systems, for example, typically rely heavily on stream processing, as constant data flows are expected from the numerous individual systems which make up operational grids in this setting. Meanwhile, **Batch Processing** is typically used when the analytical processes do not have real-time requirements, i.e. weekly business reports, legacy system data, etc. In this case data must arrive *eventually* and be processed as a whole. Modern systems are typically built to manage both stream and batch processing modes, as described in Section 2.3.

In the topic of data processing, an often utilised concept is that of **Extract-Transform-Load (ETL)** or **Extract-Load-Transform (ELT)** operations, which describes data transformations which encompass the extraction of data from its source, the transformation and the loading of data to memory for posterior use (which can happen before or after the transformation, depending on when the processing happens).

2.1.4 Data Quality

While the discussion of structure, flow and processing essentially enables an understanding of the *physical side of data processing*, it is also important to be aware of the issue of **data quality**, for what good is a system that can process millions of data entries per day, where none of the data is valuable for any of the related business activities?

Data quality refers to the *usefulness* of the data and its *value* to the activities in pertains to. Another term typically used in this discussion is **data utility**.

As data management systems grow, and more data sources are accrued, increasing the volume and variety of the stored data to unprecedented levels, it is important to consider what data is actually providing significant value to the business, and which data is causing more "harm than good, increasing costs and providing no significant value. This topic has grown as a research problem [1, 102], and will likely continue to grow as the trend of regulatory strictness will impose limits on what kinds of data may be collected - further increasing the drive to maximise data value.

Data Quality can be analysed as the conjunction of a large number of factors [88, 101]. Some are presented here, although this is an open research topic, where the dependencies between them and their degrees of importance to the subjective measure of quality are still under analysis (Table 2.1). Nevertheless, the aggregation of measures helps to understand why data quality is such an important topic - the lack of quality in data is a multifaceted problem that must be tackled in order to maximise value

Trait	Description
Timeliness	Extent to which the age of data is appropriated to the relevant business task. *Volatility* is an related measure which can describe for how long a data object will be useful to the relevant task.
Currency	Measure of how "up-to-date" a data object is. Also known as *freshness*.
Consistency	How consistent is the data object when compared to previous data objects of the same activity, and how compatible is it with the processing pipeline.
Accuracy	Measure of correlation between data and its real-world counterpart (be that a fact, a product, a business reality, etc.).
Completeness	The extent to which data is of sufficient volume to match its relevant business activity.
Accessibility	Measure of how readily available and accessible data is for its intended purpose.
Reputation	Measure of how trustworthy the data is, stemming from its sources and ingestion processes. Also referred to as *trustworthiness*.
Reliability	Extent to which a data object can maintain its usefulness, value or performance under variable conditions.

Table 2.1: Measures of data quality as surveyed by Sidi et al. indicating a summarised description of each trait.

Some techniques attached to data quality maintenance and assurance - typically referred to as **Data Cleansing** or **Data Scrubbing** - include, but are not limited to:

- **De-duplication** - Removal of all ambiguous or duplicate data.
- **Standardization** - Enforcement of structure/format or naming standards through data transformation pipelines.
- **Verification** - Checking data for errors and marking or correcting them.
- **Enrichment** - Combination with other data for increased value, correctness, etc.

And, in the topic of enrichment, the idea of **continuous integration** of data appears - the idea of constantly combining data sources and creating unified views

that hold more value than the individual disparate data sources. The topic of integration is one that is highly discussed in modern data frameworks, due to the immense potential of combining data across many domains.

2.1.5 Data Governance

With the massive data frameworks of today managing all kinds, shapes and volumes of data on large, distributed clusters over many companies and business activities, one requirement becomes evidently crucial - *control*.

Control, in the scope of data management, involves two facets: the facet of *controlling data quality*, and the facet of *controlling data access*. By placing the focus on the data, and ensuring that through its entire life-cycle, quality assurance processes are used (such as data cleansing and integration) in a secure, compliant and effective way, the potential of the associated business activities is magnified, as the data processing attached to them becomes much more reliable, safe and powerful.

This control can generally be referred to as **data governance** [1], and it encompasses the processes, organisational structures and technology required to create a consistent data handling strategy across an organisation's business enterprise. Its major areas of effect can be summarised as:

- **Availability** - Ensuring that data is readily available and that it does not suddenly cease to be operable.
- **Consistency** - Making sure that data is consistent, correct, verified and compatible amongst the different domains and services within a business.
- **Security** - Safeguarding private data, access control and data integrity against malicious agents.
- **Auditability/Accountability** - Performing traceability processes on data to track its lineage, history and creating a reliable audit log for compliance purposes.

Through these processes, data's value becomes magnified, and the entire enterprise can reliably use it to meet their business goals. By mitigating the risks of poor data quality, and ensuring effective and efficient management, the income generation potential for data becomes maximized. On the front of regulatory compliance, the use of data governance practices has a number of benefits, namely the elimination or decrease in risk of regulatory fines and sanctions, and the maintenance of reliability in terms of business reputation.

Because of its multifaceted perspective, only some sides of data governance can effectively be tackled through architectural perspectives - some rely purely on organisational change and the restructuring of the company's internal business structure.

2.2 Data Systems and Models

Following the description of some of the core concepts of data management, and preceding the analysis of the high-level architectures used in data frameworks, it is pertinent to analyse the architecture of the **data storage systems and models** which house the data and its associated pipelines.

In this section, the four main systems in use today will be analysed: the **Relational Database**, the **Data Warehouse**, the **Data Lake** and the novel **Data Lakehouse**.

They will be described in isolation, isolating the higher-level abstract structures present in reference architectures patterns and frameworks models (as these are described in more detail in Section 2.3) and presenting the architecture of the systems in a fully-encapsulated manner.

2.2.1 Relational Database Management Systems

The **relational model** for data management was proposed for the first time in 1970 by E.F. Codd [35], as a system used to manage large data stores, and create a dependable model for both storage and access through a highly performant tabular format. The model was initially defined as being composed of *12 rules*, but has since been simplified to two core concepts:

1. Present data to the user in tabular form, as a collection of tables, each with a set of rows (records) and columns (attributes).
2. Provide relational operators, i.e. "Select", "Join", "Intersect" to manipulate table contents.

This model was adopted to create a fully featured system which used the relational theory proposed by Codd to create a robust, reliable storage system with highly performant and capable operators - the **Relational Database Management System (RDBMS)**.

Entities/records (rows) within the database hold a **Primary Key (PK)**, a unique identifier (ID) which may be referenced and used for query optimisations. This ID can be used by other entities to generate a referential relationship as a **Foreign Key (FK)**. This is a design pattern which allows for a flexible expression of relationships (**one-to-one** and **one-to-many**) between entities. Additional steps are required to use **many-to-many** relationship topologies, but these are also supported by the RDBMS.

This design creates very powerful and efficient management systems, with highly performant operations and software-level optimisations creating an environment fit for the increasing needs of the nascent data market. The transaction-based[4]

[4]Based on the idea that an action upon data can be seen as completed, much like a financial or trade transaction.

query language which was developed for these systems, **Structured Query language (SQL)**, presented a robust, simple and powerful statement-based language which provided the tools for querying any set or subset of data within a database in a very efficient way.

It was in this context that the concept of **ACID transactions** [70] appeared, containing a set of properties which must be ensured, within a database system, to guarantee data validity despite errors, power failures and other system difficulties. This set of properties and their descriptions was a very important development, as its database-centric design enabled the RDBMS to become a reference for reliable massive data storage and processing. The following properties, relating to a transaction, qualify it as ACID-Compliant:

1. **Atomic** - A transaction may be composed of multiple statements (such as successive filtering, indexing, etc.). Atomicity relates to the treatment of a transaction as a singular action/unit, which either *happens* or *fails completely*. If this property is assured, and each action is truly atomic, then the system's data is safeguarded from power failures, errors and crashes.

2. **Consistent** - This property states that a transaction can only bring a database from a consistent state to another, and it must be valid against all its own internal rules and schemas before and after the transaction.

3. **Isolated** - Considering that a distributed and concurrency-enabled RDBMS may have concurrent transaction execution on the same resource (table), isolation principles state that the global outcome of a set of concurrent should be as close as possible to a sequential application of the transactions, to avoid concurrency-related issues (dirty reads[5], non-repeatable reads[6], etc)

4. **Durable** - Related to the concept of "transaction commitment" - once a transaction is concluded, its changes are committed, and they will remain committed even in the event of system failure. This involves the storing of transactions which have completed into non-volatile memory.

Database systems are still very much in use today, as their predominance, for a long period of time, made them unrivaled in their potential. The next solution (the *Data Warehouse*, discussed in the following sub-section) was an abstraction which, for most of its existence, used RDBMS as its back-end.

Novel techniques such as **sharding**[7], **distributed databases** and even the adoption of alternative query languages such as **NoSQL** and the adoption of cloud-native solutions has allowed the fundamental concepts of the RDBMS to remain in use even today, more than fifty years after its inception.

[5]The retrieval of data which has been updated in a concurrent transaction, but is not yet finished.

[6]Where data is read before being permanently changed, and thus results in the returning of data which no longer can be found in the system.

[7]Type of horizontal database partitioning which separates databases into smaller, faster and more easily managed parts.

2.2.2 Data Warehouse

The concept of a **Data Warehouse (DW)** was introduced in the 1980s, as a conceptual model to facilitate data processing on large enterprise data flows, and replace the ad-hoc, non-*architected* solutions of the past. In essence, a DW consists of a subject-oriented data store which can be readily used to support management decisions [75]. It became the go-to solution for Business Intelligence (BI) reports and data analysis due to its efficiency and hierarchical structure which fit naturally within the enterprise environment

To discuss the technology and underlying architecture of a typical data warehouse system, some context is provided to understand the motivations behind the design of this technology. Subsequently, the system is detailed in full, along with its advantages and disadvantages.

Context

It was designed as a way to facilitate *data integration* while solving the inherent inefficiency of the systems which preceded it, especially when framed within the period of tremendous technological evolution which spanned the 1960s to the 1980s (namely the introduction of the previously mentioned relational database systems). These systems began showing their faults with the proliferation of **"extract"** programs (programs which look for data, extract it from its current store, and move it elsewhere), which, when done in a chain, resulted in an exponential increase in inefficiency, stemming from redundant querying, unstructured "extract-program" design and lack of structure in the data itself (i.e. fields with same names but different meanings, duplicate fields, etc.).

The change in approach which led to the Data Warehouse's success started with the separation of data into four scopes (Image 2.2 [75]:

- **Operational** - Data record currently in use in the business activity. Can be subject to immediate change, and is considered to be mutable and volatile. e.g. *The credit score of Subject X*.
- **Atomic** - The historical records associated with operational records of one or more activities. These are immutable and present a history of the associated subject within the integrated business activities. e.g. *The credit history of Subject X (numerous previous credit score records).*
- **Department** - Analytical, normalised data pertaining to the historical records of the atomic level. This data has no overlap with the atomic level, consisting in the output of statistical analysis pertaining to the relevant department e.g. *For the customer management department - Number of new subjects per month*.
- **Individual** - The small-term data used by executive information systems. Heuristics information that exists temporarily to serve a purpose. e.g. *452 new customers since 1982*

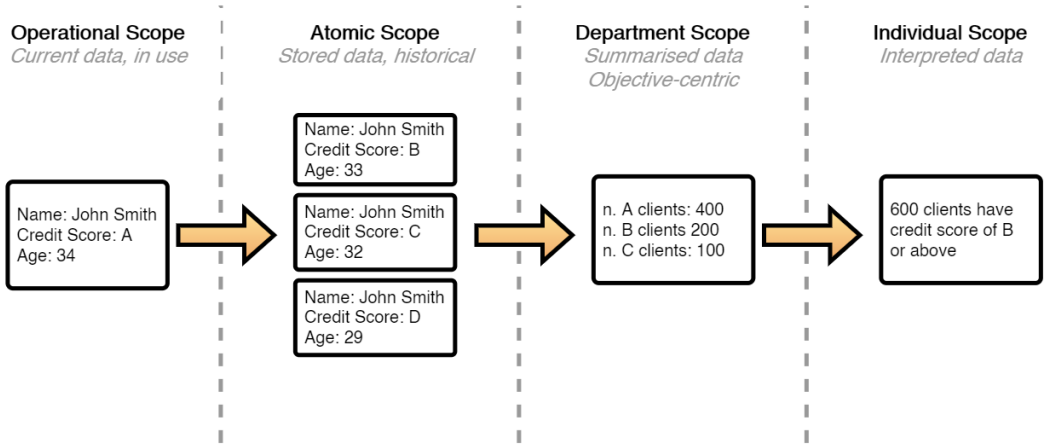

Figure 2.2: The four-scope model of data management within an organization.

This four-scope model encompasses the creation of the data (operational), the storage as data records (atomic), the aggregation and combination of data in a context (department) and finally the analysis and interpretation (individual). In and of itself, this approach creates more opportunities for efficient data usage, and lays the groundwork for the development of the **data warehouse** as a solution, centered around the atomic and department data scopes. One of the main advantages of this model is that, as data passes from the *operational* scope to the *atomic* scope, it can be **integrated** - combined with other operational data to create an enriched record - and gain value for the often multi-faceted business operations within a company.

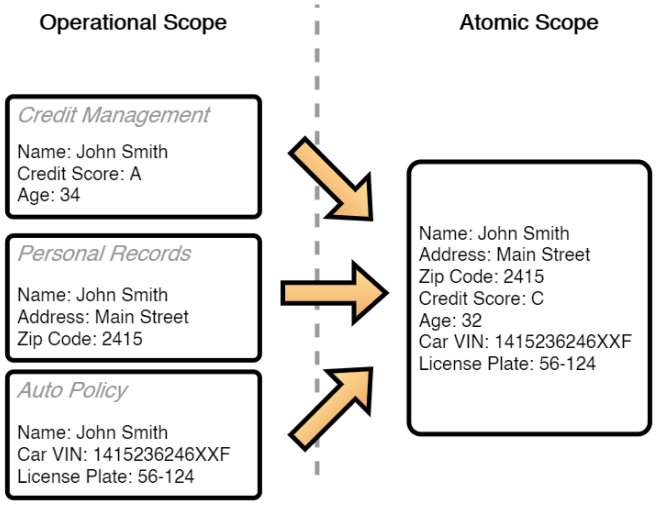

Figure 2.3: Data integration in the transition from the Operational Scope to the Atomic Scope. Multiple data records are combined to provide a more valuable dataset for statistical analysis.

So, a data warehouse essentially encompasses these four scopes, creating a system that is built around the passing of operational data into the "warehouse", which fuels the departmental views. Data flows from the current context, the immediate operations of the business, to the executive dashboards and business

intelligence views which are built off of summarised data pulled from the warehouse.

Definition and Description

The **Data Warehouse (DW)** can be defined as a data management and decision-making support system which encompasses the full life-cycle of data in the Business Intelligence (BI) context. Data is born in the operational context, over the course of several business activities, and it is recorded in the warehouse, with or without integration with other sources. Departments within the business can extract data from the warehouse and subject it to analytical transformations, producing summaries and aggregations which can then be used to build BI views for high-level business decision-making [92].

It uses a layered architecture, following the general layout defined by the previously identified scopes of data: operational, atomic, department and individual (described visually in Figure 2.4). The DW is a powerful system for BI use-cases that leverages the use of **Extract-Transform-Load (ETL)** operations to integrate multiple business data sources and create statistical value within data for analytics, dashboards, reports and other business uses.

An adjacent system is the **Data Mart (DM)** which exists as a less layered, activity-specific data store, existing as a leaf of the greater warehouse system. Data which is prepared and stored in the DW is then sent to these DMs, which serve as application-specific stores. The analogy which is used (warehouse-mart) allows for a simplified understanding of the concept of a data mart. The items in the warehouse are sent to the appropriate marts for sale; and in the case of the DW, the DM serves as a data store for appropriately selected data, which could be reports (in the case of BI-related departments such as sales, finance, etc.) or even records closer to the raw information (such as consumer-facing applications, etc).

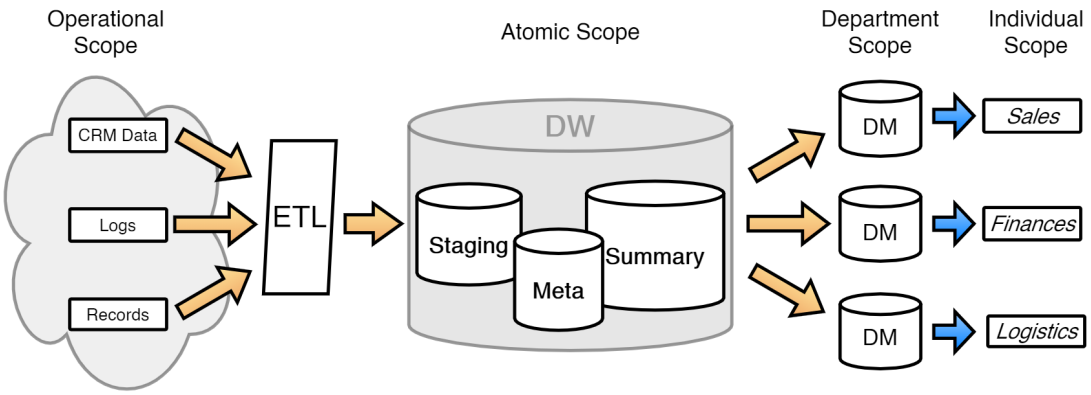

Figure 2.4: The generic architecture of a layered Data Warehouse. Data flows from business activities (such as CRM, Logs and Records) into the Data Warehouse after transformations (ETL). Afterwards, selected data is pushed to Data Marts to serve the needs of the business (Sales, Finances, Logistics, for example)

Advantages and Disadvantages

The Data Warehouse was able to bring a solution to many issues, namely the lack of structure in the immature data management frameworks of the 1980s. It's approach defined data analytics for a long period of time, and brought innovations and underlying structures which are still valid, even forty years after its conception.

The main advantages of the Data Warehouse include:

- **ACID compliance** - The use of internally ACID-compliant (Atomic, Consistent, Isolated and Durable) storage mechanisms, such as Databases render the DW capable of ensuring data operation validity.
- **Empowered Business Intelligence** - through the use of efficient integration and analytics on pre-selected domain specific data, BI can be sped up and decision-making processes can be enhanced greatly.
- **Potential for data quality** - the ETL chains and integration focus of the system can lead to massively increased data quality and value.
- **Efficient data access** - Simplifying the access chain by pre-selecting data and increasing granularity the closer data gets to its end-user allows for a better flow of information.

However, it was ill equipped for the tremendous change which would follow in the Big-Data era. Modern implementations of data warehouses seek to focus its use on the decision-making support processes that occur within a company, as the Big-Data workloads of massive exploratory data analysis do not mesh well with the structured, streamlined approach of the data warehouse.

In summary, the main disadvantages include:

- **Over-use of granularity and physical separation (*data silos*)** - While the use of department specific data marts may increase efficiency, it also makes the sharing of data more difficult. Departments effectively have their own data hoard (colloquially known as a *information silo*) and have no way to access other department's data, limiting interdisciplinary data sharing and potentially crippling exploratory business activities.
- **Complex and rigid ETL chains and limited flexibility** - The ETL chains which provide data value and quality also slow down data processing speed and potential, since not only are there numerous operations taking place at all time, these operations often rely on a predetermined structure of the underlying data, not being readily adaptable to new data formats or sources.
- **Limited usefulness in modern analytics context** - While the DW is a powerful tool for Business Intelligence, it is limited in its potential to fuel Machine Learning, Data Science and other exploratory endeavours, since these often rely on unstructured, highly variable data flows, which the DW is generally not built to support.

These advantages and disadvantages makes the Data Warehouse a powerful, albeit limited tool for efficient BI, which is most useful in cases where the data's underlying structure is immutable over time. For these cases, the DW is an incredibly developed tool, with many technological alternatives, and it remains a crucial part of the digital business landscape. Additionally, while it may present some limitations regarding exploratory data analysis, the DW can be integrated as a component of a bigger, more diverse system (as is described in sub-section 2.2.4) which effectively addresses these problems and maximises the utility of this system.

2.2.3 Data Lake

The concept of the **Data Lake (DL)** emerges in the 2010s, as a model for providing data stores that can adapt to the flexible nature of Big-Data workloads. While a Data Warehouse (DW) can be seen as a neatly arranged and organized collection of treated data, ready for consumption, the DL is the opposite, an unfiltered mass of raw, treated and mixed data, stored in a flat architecture and loosely organized, to provide the opportunity to extract yet unknown information from exploratory analysis processes [55].

It became the solution to the problems that began to arise with the structured approach of the DW, allowing companies to leverage the benefits of Machine Learning and Data Science into business decision-making, product design, marketing and many other facets of the business experience [92].

To provide a clear understanding of the Data Lake and its place in the modern business landscape, some brief context is presented, followed by a description of the architecture of a DL, and concluded by a critical look at its main advantages and disadvantages.

Context

With the changes in the technological landscape of the 2000s, namely those related to the massive increase in data volume and variety, the need for better solutions for storing and analyzing data emerged. The nature of the collected data started to shift, namely with the proliferation of the internet and mobile networks, resulting in a massive increase in semi-structured and unstructured data flows.

Accompanying this increase in data flows, developments in Machine Learning (ML) and the growth of the discipline of Data Science (DS) led to the notion of untapped potential hidden within data, information hidden within the imperceptible patterns of business data-flows which could be obtained with the help of exploratory analysis. During this change in the market's activity, the advent of cloud-based, scalable and inexpensive storage drove up the potential of data hoarding, by enabling large masses of data to be stored and efficiently used by these exploratory statistical methods.

To harness the volume and variability of these data-flows while maintaining steady

business operations, the idea to use a generic, centralised storage system to serve both structured needs (BI, etc.), and unstructured needs (ML, DS, etc) was born. It is in this context that the concept of the Data Lake originates, a way to store data from variable, mutable data sources, along with the traditional business information from static sources, reducing the overhead in separating one from the other, and reducing the amount of pre-processing done before data storage (one of the main inefficiencies of the Data Warehouse systems that preceded it).

Definition and Description

The **Data Lake (DL)** can be defined as a centralised data store which is not concerned with the form of the data, storing everything from structured data (database tables, tabular records, etc), to semi-structured data (XML, dynamically structured or self-describing data) and unstructured data (rich media data, audio, video).

The DL leverages scalable and inexpensive storage to build massive data collections of all aspects of a business, simplifying the process of data harvesting, and applying ETL only when strictly necessary, effectively reducing the global processing time. Figure 2.5 exposes the generic architecture of a DL system.

To navigate the DL's contents, usually some form of indexing is used, be it sequential indexing or metadata based indexing [92], where each item has its own unique identifier. After this identification, purpose-built and ad-hoc queries are constructed to pull data from storage.

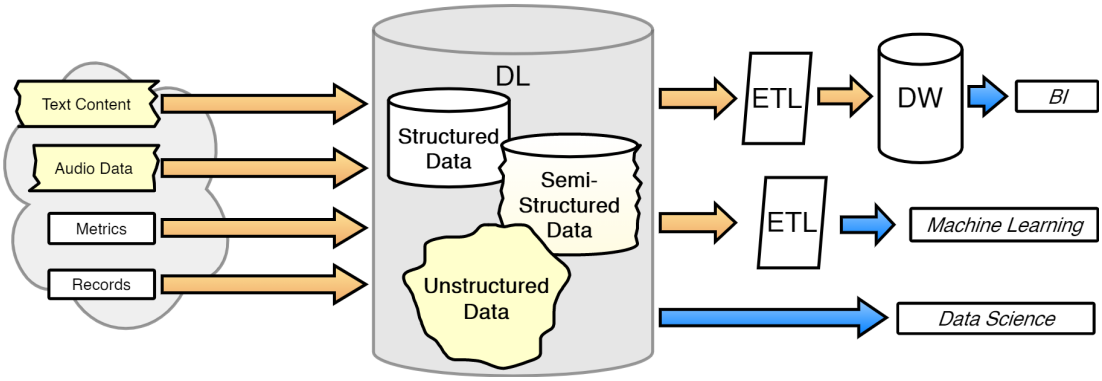

Figure 2.5: The generic architecture of a Data Lake. Data flows from business activities (such as CRM, Logs and Records) into the DL, where they are stored together. BI data flows are generally treated with ETL, and can even be sent to an intermediary DW before use; while ML and DS data flows can be consumer directly or after ETL.

Advantages and Disadvantages

In the new context of Big-Data, the Data Lake provided many opportunities. Empowering data exploration was the chief benefit over the more structured approach of the Data Warehouse, bringing a greater degree of flexibility to data

management systems across the board, by showing that structure was not strictly necessary.

The main advantages of the DL include:

- **Simplicity of implementation, startup** - The implementation of a DL, in its simplest form, relies on a single storage component and some utility access components, making it very simple to set-up. It is suited for any point of a business' data life-cycle, since it can be made efficient for all stages of development. Because of this, and due to the proliferation of infrastructure-as-a-service/platform-as-a-service initiatives, the DL has become a very popular managed service among cloud providers, bringing the possibilities of the DL with costs that grow along-side the business activity, empowering smaller businesses to participate in the data market.
- **Flexibility of storage and serving operations** - Storage which exists independently of the input format is highly desirable for today's multifaceted and interconnected business operations (namely in terms of metadata extraction, data enrichment and integration), eliminating limits (like the ones identified for the Data Marts, the *data silos*) which previously hindered data quality management.
- **Potential value** - The value of a formal data management system without the high startup costs or need for experience with complex ETL pipelines makes the DL a very attractive solution for data management.

However, the DL, which was considered by many to be a silver bullet due to its low startup costs and high prospects, attracted a significant amount of criticism due to its lack of perceivable value, in comparison to its "promise" of added value to business operations. The main disadvantages associated with this approach can be summarised as:

- **Data quality issues** - If there is a lack of pre-treatment and selectivity in the data ingestion pipeline, the data lake may potentially be filled with data that has no real use for the intended business objective. While exploratory data analysis thrives on large datasets and has many possibilities, the initiative to store any and all data can lead to overcrowded, costly DLs which at some point become intractable, and are filled with data that has no observable value - colloquially known as data *graveyards* or *swamps* [94].
- **Cost creep and storage bloat** - With current cloud-native systems being very popular with DL practices (due to their pay-as-you-go schemes benefiting the creation of scalable data stores early on in a business activity's digital life-cycle), it is possible that the rapid growth of storage requirements (storage bloat) may bring about uncontrolled cost expansions, leading to the failure of the initiative.
- **Lack of ACID compliance** - While not a strict requirement, the lack of ACID-compliance on operations within the data lake's centralised storage (owing to their variability

Owing to these factors, the Data Lake is clearly a system with a lot of potential, but the implementation must be meaningful and careful in order to avoid the pitfalls of excessive data hoarding, lack of data quality and bloat. Many of its issues can be mitigated by building around the DL with ancillary systems, as will be described in the following sub-section.

2.2.4 Data Lakehouse

As previously discussed, the Data Warehouse (DW) and Data Lake (DL) were created to solve problems related to the storage and treatment of large volume data streams, with the DW bringing a sense of order to the data pipeline, and the DL extending that order to encompass more flexibility. Both these systems are useful in a modern pipeline, and can be integrated with one another for a greater degree of adaptability.

This is where the concept of the **Data Lakehouse (DLH)** [32] appears: a conceptual framework to unite the advantages of the DW and the DL harmoniously, while attempting to create opportunities to tackle the data quality challenges that both these systems face. While the DLH is not yet completely well-defined, as a recent technology, it has attracted a significant amount of interest as a potential solution, bringing the potential of the DL as well as the manageability and control of a DW to a hybrid solution which enables not only high performance, but also governance and policy integration on an unprecedented scale.

Context

With the changes to the data market, enterprise challenges grew, and were gradually addressed with the introduction of the DW, and then the DL. However, many challenges still remained. All the previously identified solutions present disadvantages, and even when combined (hybrid or *two-tier*)[32], they result in some functional difficulties which make them unsuitable for a modern Big-Data workload:

- **Data Warehouse** - lacks flexibility, too complex and costly to set-up. ELT/ETL heavy, so scalability may be difficult.
- **Data Lake** - too inefficient, lack of opportunities for data quality. Scalable, up to a point, but may bloat too rapidly.
- **Hybrid solutions** - too complex to coordinate and integrate data with vastly different formats, bringing forward more ETL/ELT, more inefficiency, less scalability, etc.

It became interesting to design a data management system that combined the low complexity of the Data Lake with the efficiency of the Data Warehouse, without falling into the trap of the hybrid/two-tier solution which increased complexity for little gain. To this end, a concept came forth: to use an interface that simulated

a DW, "cataloguing" and simplifying the underlying data system, which stores mainly in an open format, and creating interfaces that refine the data from this open format into a readable, readily usable dataset at the end-users point of view.

Definition and Description

As previously described, the **Data Lakehouse (DLH)** is the combination of Data Warehousing (DW) principles with the Data Lake (DL). The DLH seeks to make data access uniform through the use of a **metadata layer/catalog** [32], reducing the complexity of data access (avoiding situations like intermediary DWs as described in Figure 2.5). Data is requested through the metadata layer, pulled and forwarded to the end-users (through one or more intermediate steps).

In essence it consists of an additional layer built upon a DL which provides DW-like access to a sort of "metadata catalog" of the data within. This catalog can be built to resemble a database and work in a transactional manner, providing ACID compliance and opportunities to map and manage the data more effectively. This is functionally achieved by implementing a series of APIs on top of the DL; first by creating a metadata layer, upon which domain-specific APIs can be built to fulfill the domain-specific data access conceptualised by the DW's data marts.

This metadata layer is often built indirectly (by using categorization processes that occur upon data ingestion, known as hooks) but can also be built by data analysis processes that take place with previously stored data (by using processes known as metadata crawlers [10])

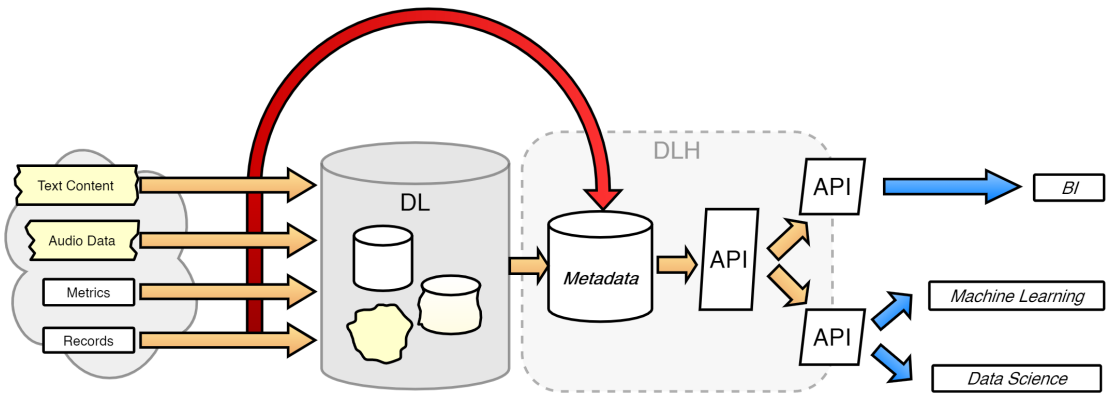

Figure 2.6: The generic architecture of a Data Lakehouse. Data flows leading to the DL are analysed and a metadata layer is built. This layer is then accessed through a series of APIs, each having its own specific set of access rules, ETL, etc. encompassed within to fulfill the department-specific portion of the DW inspired data access protocol.

The progressive nature of data flows in the DLH can be looked at as not only a form of data flow, but also as a form of data governance, as the APIs which pull the data from the DL can be configured to filter the data, or integrate/enrich it as they move toward their end-point. This makes the DLH a very useful and highly extensible tool not only for data storage management, but also for compliance, governance and overall control of the sharing and use of data.

As will be described in Section 2.4, the Lakehouse can be used as a steady platform for the development of governance based frameworks, since it creates a suitable abstraction

Advantages and Disadvantages

This additional layer built upon the Data Lake brings a number of advantages:

- **Efficient ETL/ELT support** - By adopting characteristics of transactional systems on the metadata level, it becomes much simpler to perform ETL or ELT, by simplifying the "extract" portion of these operations as a simple API-based query performed on the metadata layer.
- **ACID Compliant metadata layer** - This metadata layer is structured like a database, which allows for the implementation of transactional data access (ACID-compliant), making it a much more secure and reliable method of data management.
- **Flexibility** - Maintaining the flexibility of the DL with regards to accepted data formats (structured, unstructured and semi-structured) is a key benefit to this approach, since the gains in terms of usability do not result in a loss of potential for data collection.

But, while the DLH brings a number of advantages, some problems remain, namely those carried over from the DL, albeit in a lessened way:

- **Cost creep and storage bloat** - While the Lakehouse creates opportunities for data quality management, it does nothing to address the potential for accumulation of unnecessary data. Despite this, with the use of a metadata layer, the large volume of data within the system becomes more readily usable, creating opportunities to build value off of this "unnecessary data", or, at the very least, facilitates grouping of this problematic data, allowing for easy exclusion from processing should analytics point to a lack of value, effectively saving on storage space and costs.
- **Data Quality issues** - Similarly, the quality assurance methods are similar to those employed in the DL, thus presenting the same issues. However, by facilitating data analysis through the metadata and governance layer, it may be possible to facilitate a greater degree of integration, data sharing and enrichment, possibly tackling some of these problems.

The DLH then becomes a very interesting candidate for the development of new solutions. By offering tools to tackle the issues of both the Warehouse and the Lake, it creates a more stable foundation for the adoption of more efficient and compliant practices.

2.3 Data Architecture Patterns

With the previously mentioned management systems, there now remains the question of how they will be utilized in a production architecture, where large performance demands are the norm, and the organization and harmonious operation of these often massively distributed systems must be ensured. The formalization of software architectures, which map the components, their relationships, interactions and logical layout is crucial to this task.

While the architectures themselves are often incredibly varied and possess a wide array of technological solutions, options and components (mainly due to the differences in the projected business use-case of the architecture, as well as the stakeholder contracts [69]), some **reference architectures (RAs)**[8] and **patterns** can be used to start the development of a tailor made solution. These have been validated through formal review processes [20] or through extensive refinement based on experience within the industry.

Some patterns have emerged which, either by their simplicity or their powerful and flexible nature, have dominated the landscape for Big Data platform architecture design. These involve the creation of pipelines which can serve *both* stream and batch ingestion, a requirement for modern pipelines which can collect data from many variable sources.

For this work, two patterns were studied in depth, the **Lambda** and **Kappa** architecture patterns. These present solutions for managing the mixed data flows which are required for a Big-Data capable architecture, and are regarded as very strong patterns, due to their widespread use in the industry, and their battle-tested robust design.

2.3.1 High-Level Concepts

Before discussing the two main high-level architecture patterns for Big Data architectures, some concepts will be presented: the idea of Layers and Zones or Paths.

These concepts are often used interchangeably, since both of these concepts describe *logical components* - virtual divisions/partitions of a system, typically encompassing multiple physical components under a group. These can be considered *high-level descriptors* because they inform on function, form or quality while not physically representing any actually implemented components (low-level).

Layers

The concept of a layered architecture is self-describing - a layout where logical components (layers) encapsulate within them a set portion of the logic, communi-

[8] A reference architecture is a "blank-slate" typically used as a starting point or template for the design of a more featured and specific architecture solution.

cating with other layers and pushing data from the ingestion layer to the serving layer.

Typically, a **three-layer system** is used, encompassing the three main optics of data management, **Ingestion**, **Storage** and **Serving/Consumption**. Some architectures may be simplified, discarding storage altogether. Most architectures, such as the ones described in this section, can be observed through a layer-based view. This abstraction serves a number of purposes, namely the aggregation of components in a purpose-driven view, enabling more efficient requirement specification.

Additionally, it makes the communication of architectures and their overarching behaviour much simpler and evident.

Zones/Paths

The concept of **processing/storage zones** can essentially be summarised as the definition of high-level abstract "paths" for data to take in its course through the architecture which confer some information regarding the nature of the contained data flows or stores.

As an example, a system which relies on readily accessible data may build a logical structure known as a "hot" zone/path, where data flows in real-time and is readily query-able and volatile. That same system may have a separate stream of data which is not under such strict performance requirements, necessitating more cleanup and processing before eventually being stored. This stream of slower, less readily available data may be called the "cold" zone.

While modern architecture patterns (such as the ones discussed further in this section) use these concepts, they are often defined on a per-architecture basis - some cloud providers have adopted naming schemes like those described in the **Medallion Architecture** [48] - the **Bronze, Silver and Gold** zones, representing data quality in a progressive maturity model; while some stick to the "hot/cold" or "fast/slow" dichotomies.

2.3.2 Lambda Architecture

One of the main architecture patterns which emerged to tackle the Big-Data problem, focusing specifically on the challenge of handling both stream and batch data within the same architecture in a cost effective manner, is the **Lambda** architecture [89], first proposed by Nathan Marz.

This architecture proposes the use of a layered composition, featuring a **Batch Processing Layer** and a **Stream Processing Layer**, two separate sets of processes which serve the mixed requirements by using the native traits of both stream and batch data to serve up **data that is processed in motion** (the real-time streamed information) and **data that is processed at rest** (the batch data). The data is then consumed by its respective *job*, either a real-time view or a batch view (typically

either applications, dashboards or services). Some interpretations of the Lambda architecture have an additional layer, the **Serving Layer** which is used to prepare the batch data for viewing purposes. A diagram representing the typical description of a Lambda architecture is present in Figure 2.7

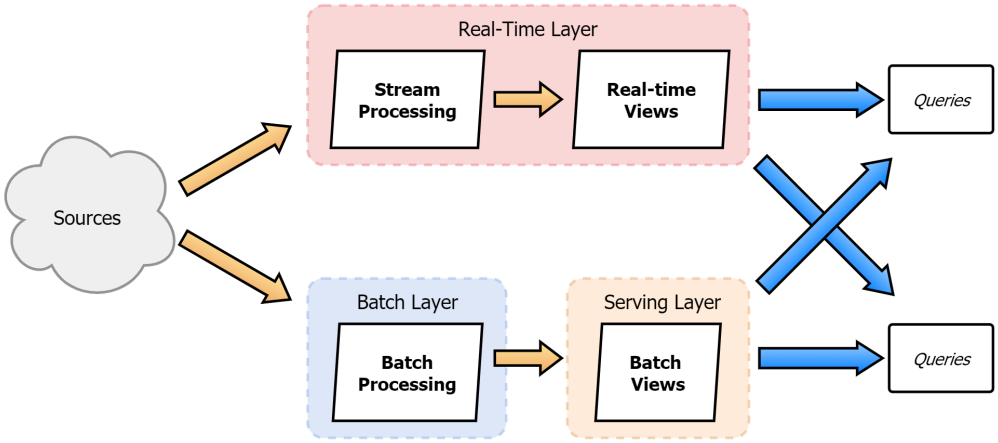

Figure 2.7: The generic architecture layout used in Lambda architectures. In this example, data is processed through two separate layers, the real-time stream layer, and the batch layer. Additionally, a serving layer is employed to organise the batch data into consumable views which can be queried by external systems.

Within these layers, components were selected specifically for their ability to match the corresponding data-type with the utmost efficiency, never inter-mixing their flows or sharing data between one another - two separate pipelines feeding different views.

This approach is very effective due to its simple premise - *deploy the right tool for the job* - and it provides numerous benefits to the cost management of such a system: the batch processing is very high volume, so it is done less frequently in order to minimize spending; and the real-time streams, which have higher performance requirements, can be managed independently, enabling cost/rate control measures only on the "expensive" part of the architecture.

However, this design presents significant challenges [86] regarding the additional problems that come with managing two separate fully encapsulated applications (real-time and batch) at the same time.Other issues appear when attempting to combine data between sources after its processing tooling, since the processes which transform and process the data are typically not intended to produce data that is readily combine-able, but rather directly consumed. It is worth noting that for a straightforward use-case that is low on data discovery and integration activities, this architecture presents numerous benefits, and can be set-up *ad-hoc* in a very efficient manner.

2.3.3 Kappa Architecture

The two main problems of the Lambda architecture - *excessive overhead/complexity* from dual pipeline management and the *lack of integration* potential - precipitated

the development of a new solution, one that simplified the processing stack, unifying the two vastly different data flows, avoiding the two-system solution entirely, and simply enabling the stream processing flow to accommodate the batch flows too.

This was the main concept behind the development of the **Kappa Architecture** model [80], proposed by Jay Kreps, which extends the functionality of the real-time streaming layer to accommodate both of the previously defined "batch" and "stream" flows. This use, which was previously limited by the fact that real-time processing is more expensive, computationally, than scheduled batch operations, was now enabled through the use of highly scalable and elastic solutions that are enabled to hold some historical values of the data they process (like *Apache Kafka* [24]).

Essentially, by "*holding onto*" (logging, saving for re-transmission, etc.) the instances of incoming data in a stream, through the use of re-processing, a batch-like method of operations is achieved, wherein a processing job can be set to re-execute on "historical" records, allowing both direct, real-time and immediate processing and batch historical data analysis on the same flow. Figure 2.8 is the generic a simplified version of the Kappa pipeline, using a data store to re-transmit data whenever necessary for stream processing of a collection of historical data.

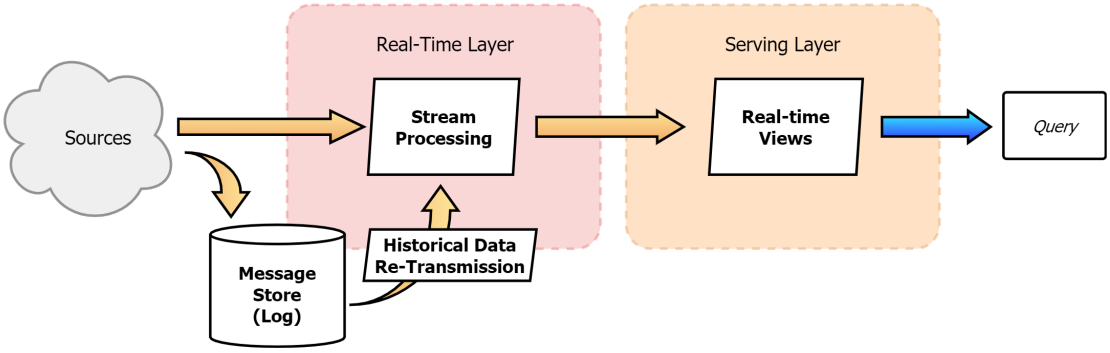

Figure 2.8: The generic architecture layout used in Kappa architectures. In this example, data is processed only in the real-time layer, with historical data being retransmitted from a message store.

In essence, whenever a historical analysis/view of a specific time period is needed, the only necessary step to harness it is to perform a re-transmission of data that passed through the ingestion streams during that time-frame. This is very easy with modern ingestion systems, very computationally efficient and greatly simplifies the technology stack required to make use of both batch and stream data.

While Kappa does simplify the technology and underlying mechanics of such a system, it also brings some challenges, namely that of the centralised point-of-failure - the message stores/logs. Because it no longer has a dedicated layer for each data flow type, it is less fault tolerant by default, but even more so when the entirety of the now abstracted *batch layer's* functionality relies on the maintenance of a consistent log throughout large up times, through possible system failures.

2.4 Novel Governance-Oriented Approaches

The technological advances presented previously, as well as the emerging patterns for high-performance, scalable data processing, have enabled the data market to grow and host a number of innovations which seek to expand the data framework into something beyond the previously identified limitations.

With the "physical" aspects of data generally resolved and accounted for with high-throughput and massively performant cloud processing, there comes a need to target the qualitative and business-oriented aspects of data: data quality, security and privacy, in an environment that tends further and further toward ubiquitous data sharing among business entities.

Two novel approaches will be analysed closely: the **Data Fabric** and the **Data Mesh**. Their innovative proposals will be described, and their core ideas will be used to guide our understanding of the modern data governance landscape, and shed some light on how architecture design can be used to achieve the goals set by these state-of-the-art frameworks.

2.4.1 Data Fabric

In short, the **Data Fabric (DF)** consists in the use of virtualisation techniques to build a standardized access layer that can perform data transit between a large variety of endpoints [71], mixing on-prem environments with hybrid, *multi-cloud* architectures, decentralizing data processing and providing a holistic view of the business environment - the users see the whole data ecosystem as a singular entity. Figure 2.9 displays the generic high-level structure of the Data Fabric.

This is effectively achieved through a loose coupling of many services and data systems via APIs, creating an abstraction which removes the technological complexities associated with the fetching and use of data in a Big Data architecture. To do this, this layered architecture proposes the use of a **massive, highly complex AI-driven pipeline** (the **data fabric pipeline**) to collect, process and manage data from a variety of systems, joining them together with the assistance of AI/ML-powered data groupings, transformations, cataloguing and sorting. The output of this pipeline is a set of metadata maps (known as knowledge-graphs [59]) which define relationships between data-sets, how they may be integrated, what data-sets they may relate to, and many more meta-information regarding the underlying data. This information is extremely useful in the efforts of data-sharing, self-service data exploration and business data integration, creating opportunities and potential in otherwise disconnected data sources.

The benefits of a DF architecture are mainly related to the simplification of data access and to the abstraction of the complex inner workings, ETL and curating pipelines and data transit which takes place in complex data systems, facilitating data transformation and integration by allowing data to be made available across all systems. As per IBM's definition [71] of the DF architecture, the main benefits include:

- **Integration** - Integration is the core concept behind the DF. By using advanced machine-learning techniques, along with tools known as knowledge-graphs, it is possible to create expansive metadata-maps, providing shared insights for data discovery. This, combined with the unified data access platform creates a rich environment for the continuous creation of value within data.

- **Democratization** - The previously mentioned unified data access also facilitates the development of self-service data applications, making data available to more entities, such as data engineering teams, developers, analytics dashboards, etc.

- **Protection** - Unifying also enables centralised access control, providing a simpler and more powerful access control stack which can operate with policies, restrictions, encryption, views and domains, creating a stable environment for the evolution of governance and compliance-centric change.

These benefits are mostly centered around the simplification of the *serving-side* of the architecture, and while this presents tremendous opportunities, it also involves a massive degree of complexity in the underlying systems which support this virtualised data sharing, access and transit throughout the massive multi-cloud environments that host the DF.

Concerns emerge regarding the processes used to share data between vastly different systems and standardise access [78] which often require the interpretation of a schema for the data. This is an open research topic [41, 42, 68], as it may be required to create a model for schema-inference which can satisfy the flexibility requirements of a massive collection of varying storage and processing systems, as well as the simplicity and agility required for a self-service data framework.

High-level Architecture

The Data Fabric's high-level architecture can be described as an **interface** to regulate, facilitate and orchestrate highly **scalable, flexible and expansible data access**. By creating a large layer between the users and the storage systems, and housing within it large **pipelines for data quality, categorization, governance and integration**, it is possible to create a centralised access point for highly disparate source systems that not only **facilitates access** but **empowers users** (namely in the data discovery scope).

Inside these pipelines, by leveraging AI/ML's massive ability to process incoming and historical data, the DF can **build up knowledge-graphs**, which can provide great opportunities for data exploration and insight services, resulting in metadata catalogs that can be used to find new data, find new relationships in data, and map out possibly valuable data ops with other departments, teams and domains.

A simplified look at the DF's architecture is presented in Figure 2.9.

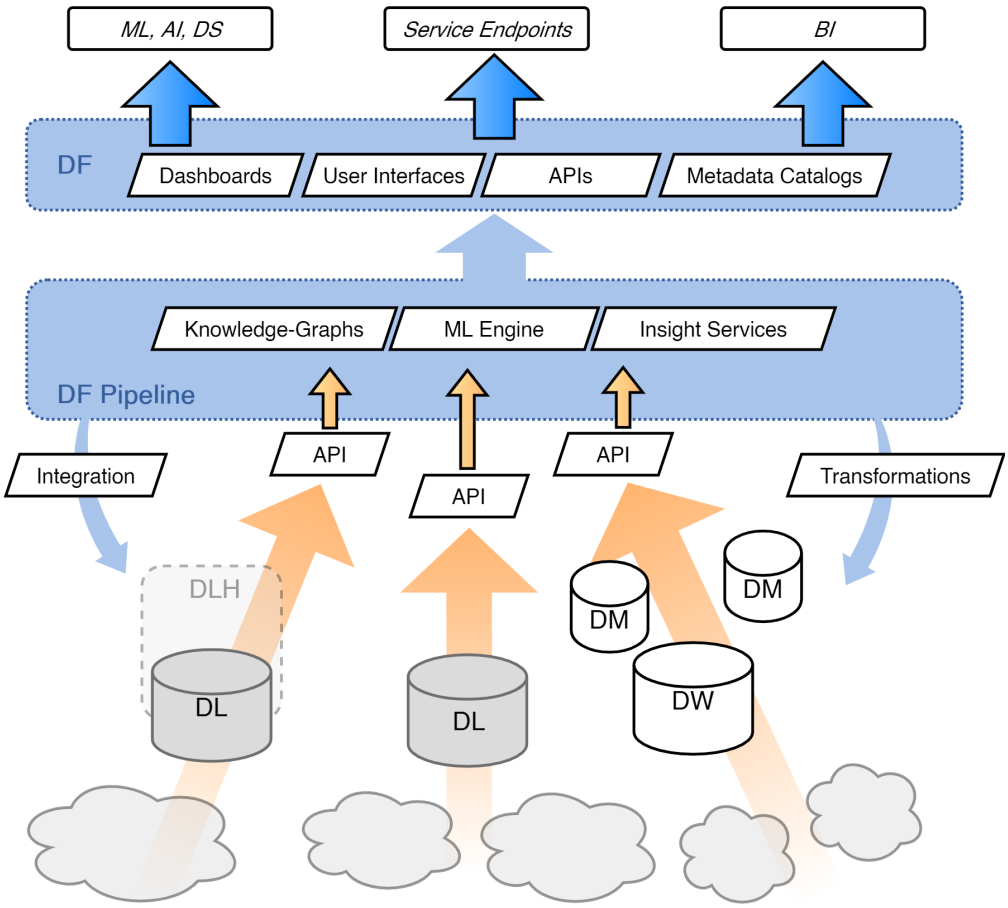

Figure 2.9: High-level structure of the Data Fabric (DF). By orchestrating several services (in this case, a Data Lakehouse (DLH), Data Lake (DL) and Data Warehouse (DW)), combining their inputs through the ML/AI-driven DF pipeline, and then unifying them under a common access point, the DF abstracts way access to individual services, providing a common layer with potential for numerous data-quality related activities (such as integration, ETL, cataloguing).

2.4.2 Data Mesh

The concept of a **Data Mesh (DM)** was first proposed in 2019 [53] as a potential model architecture to support the transition from a data-driven to a data-centric organisation. The Data Mesh architecture seeks to empower business domains within an organisation by providing an organisational framework that greatly simplifies the design of the architecture (eliminating most of the complicated integration tooling), while empowering governance and compliance within the business. It leverages the technological benefits of the *Data Lake*, *Lakehouse* and *Fabric*, while providing the guidelines for the construction of a self-serve, domain-driven data sharing environment.

This "architecture" does not operate on the same levels as the Fabric, Lake or Warehouse, because while it uses these technologies as the underlying storage services for the data management processes, it is not concerned specifically with what technology lay beneath, but rather how its value can be maximised within a

business context, where many teams of several distinct domains may wish to interact with data, discover data value and empower their business activity, relying on each-other and working in harmony, like a *mesh*.

It was designed initially as a way to shift the focus of the Big-Data paradigm away from continually increasing the performance, load and complexity of systems into a more intuitive, innovation-oriented ecosystem, which prioritises increasing data quality and opportunities rather than the act of aggregating and using growing amounts of data.

Due to its novelty, it is still in the early stages of development, and is a topic of much discussion [66, 87], however, this description will generally focus on the definition provided by the author, Zhamak Dehghani [53].

The Data Mesh consists in a **series of guidelines** which, in short, seek to guide multi-domain businesses on how to share data between domains and enrich it, creating more value and making data the center of business decision-making. These guidelines involve maximising the value of data, by creating **rules** and methods for sharing and passing it between domains, essentially **turning data into a product** with an intrinsic value that must be maximised. This is achieved through the use of a **data-infrastructure platform**, which provides all the methods for data sharing, and lets each **domain share, handle and own its own data**, creating a self-serve ecosystem for data value maximisation.

To describe the Data Mesh thoroughly, it is best to look at it from the bottom-up, starting with its core operating concepts, and finishing on how current technologies (like the Fabric, Lakehouse, etc) can be used to empower this new solution. The main pillars of the DM are the following concepts:

- **Distributed Domain-Driven Architecture**
- **Product Thinking applied to Data**
- **Self-Serve Platform Design**

The following subsections will detail each of these pillars, as they present the argument for why the Data Mesh is a viable candidate for the next generation of data management frameworks.

Distributed Domain-Driven Architecture (DDDA)

Based on the concept of Domain Driven Design, the idea is that a singular business domain *should host and serve its own data*, and take responsibility and ownership of their own data, rather than simply offloading it into a generic shared storage service. This design philosophy can still rely on centralised infrastructure, like a Data Lake, but requires that *individual domains maintain control and singular ownership* over their own data. When data is shared to other domains, a duplicate is created, only with the data that pertains to the destination domain.

Product Thinking applied to Data

The concept of focusing on the product as the object of quality maximization, for its usefulness to the consumer, for its return on investment, etc. has long been used in software engineering (i.e. teams designing APIs, interfaces or micro-services to achieve he goals of the organisation).

The novelty consists in applying this concept to the data-sets used between the company's many domains, and attempt to *maximize the quality within data at the domain-level*, creating the so-called "Data-Product". This data-product is the object of value maximization and the goal of each business domain should be to increase the value and quality of their data-products as much as possible, in order to build a higher-value output through integration and data consumption.

In this vein, the Data Mesh specification provides some guidelines for treating data as a product. To maximise the value of a data-product, the following metrics are worth noting and striving towards as the main traits of a good data product:

- **Discoverable** - Data-products must be readily available through a catalog, registry or through meta information. This is what ensures that other domains can reliably look up data, perform data discovery activities, and build richer data-sets through integration.
- **Addressable** - In the same vein as the previous entry, the idea of a robust categorization and address system for data-products is especially interesting considering integration within a decentralized storage solution, where many disparate storage systems may be connected in tandem (much like a Data Fabric), and data-products must stay consistent.
- **Trustworthy** - This metric refers to how well a domain can ensure that the data it holds has value, no errors, and that it can be trusted to reflect facts and the reality of the business it pertains to. Processes of integrity testing and data cleansing/enrichment can provide some degree of trustworthyness, while data lineage and provenance tracking can further increase confidence in a data-product.
- **Self-Describing** - The data-product should hold enough information within itself to allow for independent exploration by outside entities. Data schemas are the foundation for this metric, but any additional metadata will improve the value of the data-product.
- **Inter-operable** - Describing how data-products within a Mesh must adhere to global standards in order to be truly inter-operable between all domains. These standardisations should take place at a global level, defining entities, schemas that enable all owners of a data-product to conform and create an environment where integration is simplified.
- **Secure** - Just like trustworthiness builds value, the guarantee that only the responsible domain can see their own data-product in its integral form is a necessary assurance to build value within a product. This trait can be readily enforced using policies, role based access control and encryption.

Self-serve Platform Design

Using the two previous points, if a **domain** is responsible for **its own data-products**, then there exists a possibility for tremendous simplification of the underlying architecture: *create a domain agnostic architecture where the only infrastructure that is needed is the one used to support a data-sharing environment.*

While traditional integration of a new data-stream and ETL pipeline would often require at least some re-tooling and complexity, plus the integration of the new stream into storage, serving and enrichment processes; the self-serve platform design eliminates *nearly all* of this complexity by providing the infrastructure for sharing (the previously discussed global standards, APIs for declaring schemas, pushing and pulling, requesting data, etc.). By isolating the data-infrastructure (in this case, the logic of requesting data, sending data, schema communicating, cataloguing, etc) and ignoring all domain-specific solutions, the overhead to integrating new data-products is tremendously reduced, since the infrastructure to support their entry into the system already exists, it merely needs to be registered.

This self-serve platform will then bring a host of benefits:

- **Less reliant on data integration engineering** - By pre-defining the rules-of-engagement and giving domains the tools to publish their own data, on their own terms, data integration pipelines are much easier to implement. A domain may request another's data through a contract, with clearly defined policies, enabling compliant integration between different domains and activities.
- **Presents more opportunities for data sharing** - Because of this facilitated integration/sharing pipeline, it is much easier for domains to use external data, opening the doors to exploratory data analysis, ML-enabled analytics, combining and enriching the data and creating more value within business activities.
- **Empowers data discovery** - Using massive data catalogs and tracking the movements of data throughout the framework, data discovery teams have a rich supply of information to assist in making business decisions, creating new domains, activities and interpreting the massive data flows that occur within the system.
- **Simplifies ETL pipeline complexity** - By making each domain responsible for their own products, and by extent, their own ETL/ELT and data transformation pipelines, they can be more purpose-built, simplified and less generic, creating better opportunities for data engineers to maximise quality within the domain.

And with this, the DM is realised. While it is still a relatively new concept, and implementations are still tentative, it provides a toolset for building data value which is objective and creates not only business value, but also compliance opportunities, which is an increasingly important factor in modern data framework design.

High-level Architecture

The high-level architecture of the Data Mesh uses a **domain as the main building-block** of the architecture, with the hope of creating a network of interoperability, while using a globally accessible **data-management infrastructure layer as a platform** (DIaaP) to facilitate data-sharing, enrichment and integration, shifting concerns away from the underlying storage (which may be virtualised). A simplified diagram of this architecture is presented in Figure 2.10.

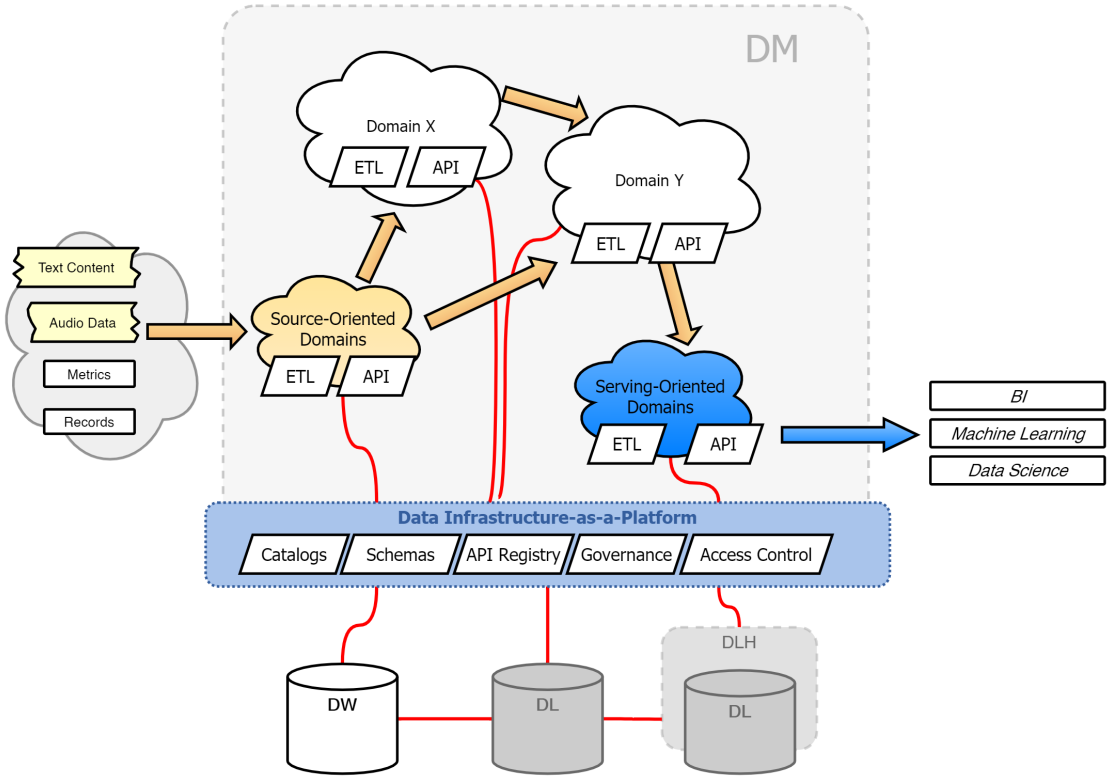

Figure 2.10: High-level structure of the Data Mesh (DM). The mesh is composed of several domains operating in tandem, using a common data-infrastructure layer. Source-oriented domains feed domains X and Y, which feed the service-oriented domains. In this case, data storage systems are virtualised (access is performed indirectly through the DIaaP layer.).

Chapter 3

Methodology

In order to develop an architecture, it is important to follow rigorous methodologies which enable its growth from the idea-stage into a fully realised outcome which meets all requirements. These methodologies span the entire design process, including the scope of project management, and ensure a traceable and verifiable approach which leads to a valid outcome.

In order to provide an understanding of the methodologies employed during the project, this chapter aims to provide a comprehensive documentation of the management, engineering and design processes which were followed, ensuring that all steps are clarified thoroughly and that all key decisions and rationales are well understood.

- **Project Management** - The process of handling the project's numerous aspects, from risks, to issues and to the structuring of the documentation and design processes to follow.
- **Requirement Engineering** - The process of eliciting requirements, validating them and developing a comprehensive requirement specification, to present a characterization which adequately addresses all functional and non-functional aspects.
- **Architecture Design** - The process of creating the final architecture, selecting components, evaluating attributes and testing their functionality; all while documenting the design decisions in a clear, meaningful way.

The first section of this chapter will broadly discuss the architecture development methodology - the **Architecture Centric Design Methodology** - and the reasoning behind its selection. Auxiliary information, namely related to requirement engineering, project management and risk tracking methodologies are also briefly presented.

The second section focuses on internal project management and change handling methodologies, namely the steps taken to ensure that traceability and risk tracking are not only systematically performed but generate opportunities for

improvement; and that change within Altice Labs' needs or within the project's requirements can be handled efficiently without causing excessive delays.

The third section will focus on the requirement engineering process, detailing the requirement elicitation, the refinement strategies for the requirement specification and the notation/format used to support it.

In the fourth section, the architecture-centric design methodology is detailed further, detailing all its stages. Additionally, the strategies employed for systematic architecture review are described.

Lastly, the documentation standard of the Architecture Specification is described - the C4 model.

3.1 Methodology Overview

For this project, the selected methodology for the architecture's development was a slightly reduced adaptation of the **Architecture-Centric Design Methodology (ACDM)** [83, 84], an approach which seeks to leverage a repeatable development cycle of refinement and systematic review to maximise the exploration of architectural drivers, requirements and technological choices.

This methodology is described in detail in Section 3.4, elaborating on its several stages and presenting the main advantages of this methodology when contrasted with less flexible and more sequential approaches. Due to the extensive amount of concepts relevant to the **requirement engineering process** which is a pre-requisite to the ACDM's iterative process, Section 3.3 is dedicated to presenting and detailing the fundamentals necessary for the requirement specification process which took place during the project.

Because this methodology relies on the iterative review of the architecture after a cycle of refinement, a formal tracking of issues and traceability within the evolution of the architecture is necessary. To this end, the **RAID Tracker** (Risks, Assumptions, Issues and Dependencies) was selected to provide a centralised record for the ACDM architecture review, as well as ancillary tracking regarding project dependencies and risk management (described in detail in Section 3.2)

3.2 Project Management

In the design of software intensive systems, especially while using frameworks which rely on systematic and traceable analysis, it is crucial to maintain a record of the numerous processes, stages, inputs and outputs that take place during development.

To this end, there are numerous project management frameworks that aim to reduce the overhead of project management, especially at scale and with teams of multiple contributors. These frameworks range from relatively simple spreadsheet based implementations to fully-featured software platforms centered around traceability and coordination. The operational conditions of this project do not call for extensive management frameworks, due to fact that only one actor is tasked with the development of the architecture, and due to the reduced scope of the modified ACDM which is used in the architecture development process.

To assist in the traceability process inherently necessary to this project (to assist in both supervision and workflow), a relatively simple methodology was chosen - the **RAID (Risks, Assumptions, Issues and Dependencies) Tracker** - which uses a simple spread-sheet based format to include information relevant to four crucial aspects of the software development process as a whole. The four titular aspects which are tracked can be summarized as follows:

- **Risks** - This category includes potential events or situations that could have a negative impact on the project's timeline, resources, or overall success. Risks are identified with some attached fields: *Priority*, *Impact* and *Likelihood* which result in a richer characterization.
- **Assumptions** - Assumptions are the factors and conditions that are considered to be true, but they have not yet been fully validated or proven.
- **Issues** - Issues are problems that have already been verified during the project execution. In the context of this project, issues may be related to the project itself or they may be specific to the architecture development process which is detailed in the following sections.
- **Dependencies** - Dependencies refer to the relationships and inter-dependencies between various project tasks, activities, or deliverables.

The use of this tracker is crucial to correctly use not only the ACDM, owing to its reliance on systematic architecture reviews to be performed every iteration, but also the requirements engineering and change management processes which also take place.

3.3 Requirement Engineering

As previously mentioned, the ACDM features an in-depth Requirement Stage (focusing on driver elicitation and requirement specification) before the iterative process begins in earnest. These concepts indicate how requirements will be specified, and as such, should be briefly presented in order to facilitate the understanding of the Requirement Engineering process which took place for this project.

In order to lead up to the full description of the ACDM, this section will provide the definitions of the different types of requirements:

- **Constraints** - Fixed restrictions of technical and regulatory nature that must be taken into account during architecture design.
- **Functional Requirements** - Requirements which describe what the system can do in terms of functionalities, what it can be used for and which interactions several actors can have with it.
- **Non-Functional Requirements** - Requirements which describe how the previously indicated interactions occur, providing measures for validation.

3.3.1 Constraints

Constraints, in the requirement engineering context, indicate hard limits for architectural decisions, and a set of restrictions that will influence the choices made around them. Constraints are generally inflexible

These restrictions can appear in a **business/organisational context** - relating to operational resources, business operations, cost management - or on the **technical side** - existing hardware limitations, pre-selected components and currently used technology stacks.

Constraints, in this case, will be represented through the use of simple text-based descriptions, with an indication of which category they relate to (Table 3.1).

ID	Source	Category	Description
BC-001	PRE	General	The system must evolve into a multi-tenant cloud-based architecture, with one deployment for several client provider.
TC-001	PRE	BI Data Flows	Business Intelligence (BI) data flows/storage processes must be compatible with the Prometheus monitoring solution.

Table 3.1: Sample definition for constraints. BC-001 indicates a business constraint, and TC-001 indicates a technical constraint.

Business constraints demand a careful analysis and generally can be used to guide the creation of several related requirements, by understanding which qual-

ities the system must possess in order to fulfil these restrictions, whereas technical constraints generally prescribe a simpler analysis of compatibility and interoperability between system components and the identified technical constraint.

3.3.2 Functional Requirements

The functional requirements of the system may be defined as those which express the system's intended functionalities. They pertain to the uses of the system and the goals which it intends to achieve.

To specify a functional requirement, the analysis is focused firstly on the several **actors**, external entities who interact with the system. By analysing their goals, needs and tasks, it is possible to build a comprehensive set of "**use-cases**", scenarios which involve actions and interactions between actors and the system, which are of value to these actors [34].

Functional requirements can be presented in a variety of ways, namely use-case diagrams or simple text-based descriptions. Diagrams are particularly useful when the focus of the analysis is on the definition of system functionalities, by facilitating stakeholder validation of changes.

Considering that, in this case, the requirement elicitation resulted in a very strict and well-defined functional analysis, the use of diagrams is deemed unnecessary, as the functional requirements are unlikely to change. The preferred method of presentation will be straightforward, actor-centric text-based descriptions, in a tabular format, as exemplified in Table 3.2. This format presents four columns associated with a requirement:

- **ID** - Internal tracking reference for entries.
- **Actor** - Actor which the constraint/functional requirement relates to.
- **Source** - Internal tracking reference for the iteration of the requirement specification.
- **Description** - The requirement itself.

ID	Actor	Source	Description
FR-IL-001	System	REQ-1	The system logs new component additions/connections through an API
FR-IL-006	Process Manager	PRE	The process manager can configure scheduled continuous data acquisition (data streaming windows).

Table 3.2: Sample definition for functional requirements. Two examples for one of the system's high-level components, with a unique ID and a Source identifier related to the iteration in which the requirement was last changed.

3.3.3 Non-Functional Requirements

The non-functional requirements (or quality attributes) of the system are defined as those which specify a quality associated with either a component, a connection or the system as a whole. Typically they will be associated with a functionality, giving more information as to **how** the system will perform it, existing as a support for the functional requirements.

There are many ways to qualitatively describe a piece of the system, although typically some standard categories [33], such as the **ISO/IEC FCD 25010** standard [73] are employed to facilitate the description of system qualities:

- **Availability** - Related to qualities such as fault tolerance, downtime, etc.
- **Performance** - Related to qualities such as latency, processing time, throughput, etc.
- **Scalability** - Related to how the system will scale to match growing demands.
- **Modifiability** - Related to how adaptable the system is to changes.
- **Testability** - Related to qualities such as fault discovery, monitoring, etc.
- **Usability** - Related to qualities such as learning rate, efficient use and error minimization.
- **Interoperability/Compatibility** - Related to how well the system can interact with other systems.

While these categories are not strictly limited most qualities can be expressed under these terms (e.g. the quality of Elasticity can be expressed as Scalability), and this makes it easier to comprehensively analyse non-functional requirements. Other qualities may also be expressed, namely those pertaining to the **business activities** related to the system (time-to-market, product lifetime, cost/benefit). Expressing these qualities is crucial to the process of architecture design, as they are the measure of correctness and completeness of the architecture specification. Relating to the previously described ACDM, the review of the architecture must include a thorough validation of the components against the non-functional requirements, often-times requiring experimental analysis to assert that the system can meet all the needs which it is designed for.

To facilitate the experimental validation of these quality attributes, a specific format is available, which breaks down the quality into a **six-part scenario** description [33], which expresses how the system responds to a certain event (stimulus). In this response, the system or component (artifact) will demonstrate the desired quality (response), in a clear and measurable way (response measure). The general definitions for the six components of a quality attribute scenario can be visualised in Image 3.1:

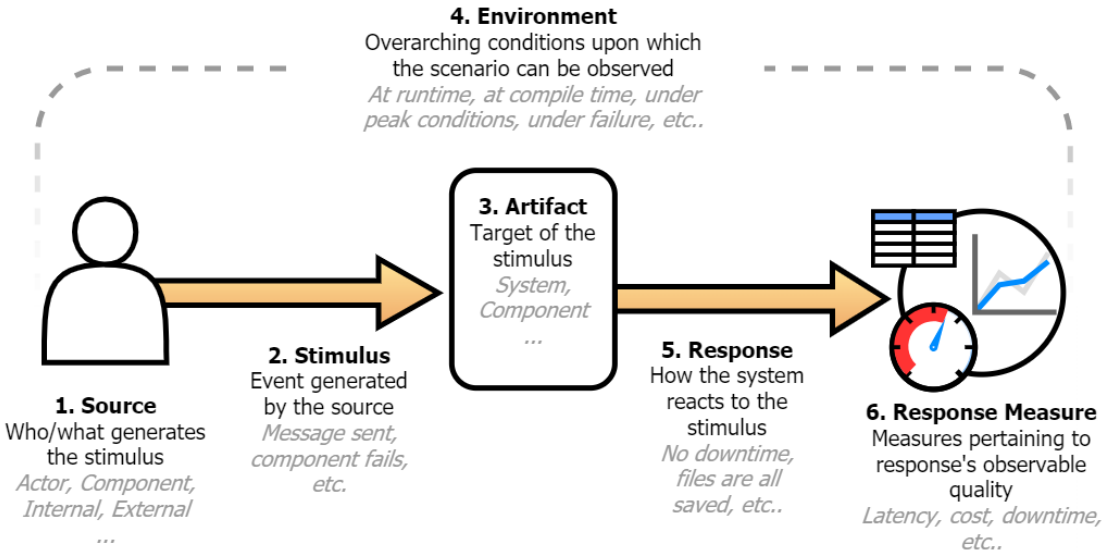

Figure 3.1: Schematic representation of a six-part quality-attribute scenario, displaying the six components included in the formal definition.

This description will effectively guide the testing for the attribute, as it specifies precisely where, when and how this quality may be verified. Because of the requirement that the response measure be measurable, the testing cases are already well defined even before experimentation planning begins. Aside from these direct benefits to the design process, the use of these six-part scenarios has the potential to improve the communication of system qualities to stakeholders, by creating concrete and concise descriptions which encompass non-functional requirements, expediting the validation process both internally and externally.

Table 3.3 presents the format used to record the non-functional requirements, showing a sample of one of the quality-attribute scenarios identified for the data ingestion processes in the system.

QAS-IL-003 - *REQ-1* - Availability
Source - Ingestion Queue **Stimulus** - A cluster of message brokers enters failure state **Environment** - At runtime, under normal operations **Artifact** - Ingestion Queue
Response - Process Manager is notified, new components are instantiated to serve existing streams. **Response Measure**: The system does not incur in significant downtime (>30s) during cluster failure.

Table 3.3: Sample definition of a quality-attribute scenario. This example features a unique ID, a source identifier related to the versioning of the requirement specification, and a summarised description of the relevant system quality that is expressed by the scenario. Following this header, the six-part scenario is detailed.

3.4 Architecture Design

The design processes followed to create this architecture use the guidelines set by the aforementioned **Architecture-Centric Design Methodology (ACDM)**, a methodology which puts the architecture at its core, seeking to efficiently and quickly build up a solid foundation for a software intensive system. It is worth noting that the strict definition of the ACDM will not be followed, as an adapted version is used to better fit with the personnel limitations of the project. In this methodology (which is detailed in the following sub-section), an iterative cycle is performed, necessitating a systematic review of the architecture at each repetition. This review must follow a strict and traceable process in order to verifiably demonstrate that the architecture meets the previously identified requirements.

3.4.1 Architecture-Centric Design Methodology (ACDM)

The **ACDM** [83] is a scalable methodology developed to facilitate the development of architectures for software intensive systems. It was developed during the evolution of the software development process, by analysing empirical experiences found during the use of the **Architecture Trade-off Analysis Method (ATAM)** [79], to be used as a dedicated architecture development process which could be integrated into other software development frameworks (such as Waterfall, Agile methodologies).

This methodology assists in creating an architecture which can complement the organizational processes and activities which will rely on the projected software. In practice, this means that the architectural decisions themselves are what will aim to support the business activities, rather than off-loading this support to the implementation side of the software development process.

There are numerous advantages to the approach defined by the ACDM [84], mostly due to the fact that major issues are addressed before any implementation work commences:

- Requirement definition can be more comprehensive;
- Product expectations are set early;
- *Unknowns* are tackled early;

It comprises a set of stages (8 in total) (Fig. 3.2) which start with the elicitation of requirements, to build an understanding of the architectural drivers, and follows that up with the creation of a first version of the architecture (known formally as the *notional* architecture).

This first version is reviewed and a choice is made: if it is ready *(Go)* then it moves to production; if not *(No-go)* then experiments and refinement will follow, until a new version is developed and submitted for review. This process will go on until the architecture is ready for production.

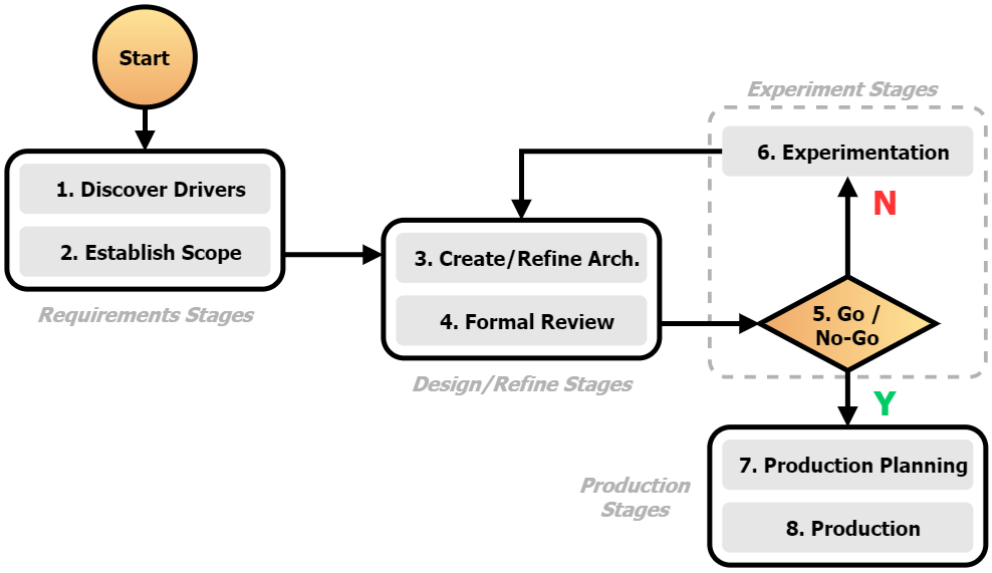

Figure 3.2: Diagram of the ACDM's stages, showing the split point in Stage 5 where the decision is made to either push to production or go back to a refinement stage should the architecture need further iterations.

The stages can be described briefly as follows:

- **Requirements Stages (1-2)** - Elicit requirements, discover, document architectural drivers and establish the scope of the project.
  - **1. Elicit Requirements, Discover Drivers** - Meet with stakeholders, extract drivers and requirements.
  - **2. Establish Project Scope** - Define project scale, limitations, etc. to prepare for the first draft.
- **Design/Refine Stages (3-4)** - Create the notional architecture, subsequently refine it, and perform a systematic review.
  - **3. Create Notional Architecture/Refine Existing Architecture** - Build a notional architecture (basic). On subsequent iterations, refine the architecture.
  - **4. Formal Architecture Review** - Using tracking, systematically review the architecture using the identified requirements.
- **Experiment Stages (5-6)** - Depending on the results of the review, move to production (Go) or experiment and return to stage 3 (No-Go).
  - **5. Go/No-Go** - Based on the review's outcome, decide if architecture is ready for production.
  - **6. Experimentation** - If it is not ready, perform experiments to target the architecture's refinement.
- **Production Stages (7-8)** - Plan the implementation, build and deploy the final software product. This stage will not be used in this project.

For this project, considering the personnel limitations, hardware limitations and the fact that the project was performed outside of Altice Labs S.A., the **Production Stages (7-8)** were discarded, and the architecture would be the end-result, no implementation should follow.

To shed light on the inner workings of the methodology, the following sub-sections will focus on describing the ACDM's several stages as well as the adaptations to the method made for this project and present the strategy employed for traceable architecture review, which will factor in during the description of the design process in Chapter 6.

3.4.2 Requirements Stages

The requirements stages involve the rigorous extraction of drivers to create a working characterization of the system which may be used to create the notional architecture. In this stage, typically, there is an **Architectural Driver Elicitation Workshop (ADEW)**, which uses a structured approach (presented in Table 3.4) to gain an understanding of the functional and non-functional aspects of the system, along with any relevant restrictions/constraints. In short, the meeting is structured to progress from a broad-stroke problem description to the description of operations, use-cases and functionalities, leading into the description of the quality-attribute requirements and constraints associated with these functionalities.

This flow enables a gradual introduction to the relevant architecture-design concepts, ensuring that the stakeholders can have a meaningful contribution to the requirement specification, while accounting for differences in familiarity with the software architecture design process.

Activity	**Objective**	**Duration**
Introduction	Describe workshop agenda and goals	10 mins.
Use-Case Discussion	Describe most relevant use-cases for the target software system	30 mins.
Operational Descriptions	Identify the main actors, functional requirements, context, workflow and other relevant information regarding the previously identified use-cases	30 mins.
Identify Quality Attribute Requirements	Presentation of the concept of Quality Attribute, extraction of quality information and non-functional requirements for each use-case.	30 mins.
Identify Business Goals/Constraints	Identify constraints such as number of users, budget, time constraints, legal aspects, time-to-market, etc.	15 mins.
Identify Technical Constraints	Identify pre-existing software architecture decisions	15 mins.
Summary	Present overview of meeting contents	15 mins.

Table 3.4: Typical structure for the ADEW, including the main activities, a description and their respective duration.

The ADEW may take place over several meetings, where the overarching goal is to consolidate data which can lead to a successful requirement specification.

This stage will result in a first specification which should provide enough context for an initial driver extraction and scope definition for the project. However, it is worth noting that this will likely not be the last time that requirements are formulated, as during the refinement processes requirements may change by virtue of the discovery of new information, submission of changes from the product-owner or even external factors (such as passing of new regulations).

3.4.3 Design/Refine Stage

These stages are repeated in the iterative portion of the methodology. As such, it can be split into two parts: the **initial design/creation phase** (occurs at the start of iterative process only once), and the **refinement phase** (occurs on every subsequent iteration). Either of these stages is finalized with a **formal architecture review**.

Initial Design Phase (Notional Architecture)

After the initial requirement and scope definitions, the architecture can be defined, albeit in a low-fidelity state. This is called the **notional architecture**, where the main goal is to rapidly produce a first version of the architecture, where issues may be identified and tackled as soon as possible. It represents a partitioning of the system to be the target of successive iterations in subsequent stages. The notional architecture will typically include:

- **Component views** - What are the building blocks of the system.
- **Context information** - How the system interacts with external actors.
- **Run-time perspectives** - How the system will operate during execution.
- **Physical perspectives** - How the system will be mapped to the underlying hardware.

With this notional architecture, or draft, it is possible to begin the ACDM in earnest and apply the iterative methodology which will mature this prototype into a correct and complete architecture.

Refinement Phase

In iterations other than the first (where this stage is omitted, and instead the previously described notional architecture is created), this stage consists in the refinement of the architecture, requirements or specification, adding new technologies, stacks or components in an effort to meet the requirements and fix the issues

identified in previous iterations. Generally, this stage can be described as a 4-step process:

1. **Analysis** - Critical analysis of the architecture with assistance from the professors and Altice Labs, usually through a presentation of the current state of the architecture with information and clarification requests.
2. **Update Requirements** - Implement the necessary changes identified in the analysis (new requirements, layers, components, etc.)
3. **Update Architecture** - Rework the architecture's technologies and solutions based on the requirements.
4. **Consolidate Documentation** - Summarise the current state of the architecture in preparation for the experimentation. This includes diagrams, descriptions or any other supporting documentation.

Wherever the architecture is found to be lacking in regards to the requirements, new solutions have to be considered and researched. The exploration of new technologies and alternatives should be open and extensive, to ensure that should a component not meet the requirements, an alternative can be presented for further validations and the architecture design process does not come to a halt.

Formal Architecture Review

After the architecture has been created or refined, a comprehensive review must take place. This review will involve the validation of the architecture against the requirements, to ensure that the newly added or selected components can perform their functionalities according to the specification. Typically, the result of this review is a table where **Requirements** are **mapped** with **Architectural Elements**, with a corresponding description containing the reasoning and justification behind their match. This mapping may present one of three values pertaining to the mapped relationship:

- **Validated** - Experimentally or analytically validated, provides reasoned assurances that the requirement is met by the element.
- **Partially Validated** - Partially validated through documentation analysis and research, requires deeper validation through either technical analysis/research or experimentation. Should generate an issue during review phase, and serves as a temporary marker that a requirement-element mapping is a promising candidate.
- **Issue** - Invalid element for current requirement. Includes a pointer to the Issue Tracking table.

A simplified and shortened example of a **Requirement-Element Mapping Table** is presented in Table 3.5.

Requirement	Architectural Element	Status	Description
Ingestion latency <100ms	Apache Kafka	Validated	Experiments indicate average latency <100ms under several load profiles
Access breaches detected 99.999% of times	Apache Ranger	Partially Validated	Documentation points to 1 in 10 million false negatives.
Downtime <10s for Storage	Apache Hive	Issue	**See Issue #004**.

Table 3.5: Simplified mapping table of an architecture review, presenting on mapping of each type (validated, partially validated and issue).

Through the systematic recording of requirements and how the architectural elements serve to achieve them, it is possible to verify if any issues exist within the current architecture, and move to the next stage - the decision-making process for either **refinement** or **production**. A formal issue tracking system is essential to this step, providing a framework for systematic analysis, planning and mitigation of the identified issues.

For this project, using the previously described **RAID Tracker** tool, and pointing from the previously mentioned **Requirement-Element Mapping Table** to the **Issue Tracker**, it is possible to assign **types** to the identified architecture issues, providing a way to plan for different kinds of refinement to take place in the next iteration. The issue types which are used in this tracking are presented in Table 3.6.

Issue Type	Issue Text	Description
1	*No action required*	Provide the clear rationale for why the issue is not necessary to act upon.
2	*Repair, update or clarify documentation*	Update the documentation, and document these changes, being specific on what changes were required.
3	*More technical information required*	Analyse the problem, perform research, ask technical questions or collect other resources online to inform on the issue.
4	*More information on Architectural Drivers*	Request information from stakeholders or clarify requirement base internally.
5	*Experimentation required*	Plan and conduct focused experiments to explore the driver under review.

Table 3.6: Issue types used in the Issue Tracking for the formal architecture review stage of the adapted-ACDM methodology.

The review process will result in a table of issues which can clearly indicate where the architecture presents flaws, either due to incompatible components, unfinished or lacking documentation, or even just lack of certainty in decision-making, requiring further validation and the creation of an experimental plan (defined in the following sub-section). The categorization of issues will allow for the appli-

cation of different handling strategies, namely when tackling issues of **Type 5**, which will require a dedicated and more extensive stage later in the iteration.

In summary, the review is structured as follows:

1. **Map Requirements to Elements** - For each architecturally significant requirement, an element must be indicated as a match in the mapping table.
2. **Label Requirement-Element Mappings** - For each of the mappings, a status should be provided (*validated, partially validated* or *issue*).
3. **Track Issues** - For each issue identified in the mapping table, an issue is instantiated in the RAID Issue Tracker, with the appropriate type
4. **Handle Issue** - According to the issue type, plan a handling strategy and either execute it in the "experiment stages", or pass the issue to the next iteration's refinement stage.

3.4.4 Experiment Stages

After the Design/Refine stage, the output of the architecture's review is analysed carefully. Should there be no issues remaining (or all issues involved being of **Type 1 Issues**, where no action is required), the architecture is presumably ready to move into production for planning *(Go)*.

However, if the output of the review indicates issues of any other type, then it is necessary to refine the architecture in a new iteration *(No-go)*. Pending the existence of **Type 5 Issues**, experiments may be required to finalize the validation of previously defined mappings.

Go/No-Go

The analysis of the architecture review performed previously, the issue tracker must be analysed to guide the remaining steps. A large number of outstanding issues or active problems will lead to an insufficient architecture in need of refinement, with possible problems revealing themselves into implementation, whereas conversely, an architecture with little or no outstanding issues may be ready for production.

The aptly named **"Go or No-Go"** stage consists in this decision, which will start a new iteration through refinement or push the architecture for production. Should the architecture need refinement, then it will pass onto the Experimentation stage, to attempt to resolve all current **Type 5 Issues (Experimentation Needed)**.

Experimentation

Issues which call for experimentation will typically come from the absence of assurances and data/metrics related to element's relationship to the relevant quality attributes. This will mean that it is impossible to truly consider the requirement met, and the element cannot be provided as a valid component for the final architecture.

The *experiment*, as defined in the ACDM [83], consists of a targeted prototype, which aims to gather data to assist in the validation of the architectural drivers (in this case, the element's relation to the relevant quality-attributes). This experiment can take many forms, including but not limited to:

- **Individual Component** experiments - Analyze the behaviour and extract metrics based on a singular component's execution/configuration. Experiments of this type will be logged as "*INDIV*".
- **Component Interaction** experiments - Analyze the behaviour of N systems in linked operation. This is intended as a middle point between the individual analysis and the system-wide analysis, giving partial validation of harmonious operation. Experiments of this type will be logged as "*INTER*".
- **System Interaction** experiments - Analyze the system as a whole, the many interactions which take place and the general compatibility of the various selected technologies. Experiments of this type will be labeled "*SYSTEM*".

These three are the main types of experiment which will be used in this project, utilising lightweight set-ups and "quick" prototyping to ensure that most of the architecture can be covered by the experiment plans. Accompanying an experimental stage is a detailed log featuring all relevant metrics, data and methodologies employed during the experiment(s) which took place.

3.5 Architecture Specification

The specification or technical description of the architecture will be presented using the **C4 Model** [39, 105], which uses notation-agnostic diagrams, leveraging different levels of abstraction to communicate the architecture in four distinct perspectives: Context, Containers, Components and Code.

The C4 Model uses a simple box-diagram language, where different entities are connected by arrows detailing their interactions and dependencies. This abstraction is based on the idea of "zooming" into the details of the system, starting a broad perspective (the Context Diagram) to understand the system's interaction with external factors, and then progressing to a closer view of the system's internal execution (the Container diagram), followed by a deeper look into the functionally programmed units (the Component diagrams) within that container.

The fourth level (the Code diagrams) are typically seen as optional in the earlier stages of the development of an architecture, especially since they relate to how the components should be programmed and how their functionality will be coded.

For this specification the Code diagram was excluded, as the development of stable, production ready specifications and code diagrams (which ultimately would be the final step before the actual implementation) is a part of the Production stages of the chosen architecture design methodology (ACDM), which will not be performed within the scope of this project.

- **Context Diagram** - Detail the system's relationship to external systems and actors to contextualise its operation.
- **Container Diagram** - Detail the system's *containers*, i.e. individually executable/deployable units.
- **Component Diagrams** - For each of the previous containers, detail the internal components which encompass the container's responsibilities.
- **Code** - For each component, detail the internal software functions, methods and calls to implement them in the production scenario.

This model facilitates the understanding of the architecture by providing increasingly granular perspectives on an otherwise complex distributed system, making it ideal for communicating the structures and components which make up the pipelines and processing paths for the final solution.

Chapter 4

State-of-the-Art

Following the contextualization and background analysis for the field of data management architecture development, it was crucial to analyse and research production implementations of these systems, and understand how the supporting technologies can contribute to the end-goals of these systems.

Following the presentation of two current implementations - Uber's Kappa architecture implementation, and SmartNews' Lambda architecture implementation - a survey of technologies is presented. This survey was performed in accordance with the project's requirements for cost controllable, preferably open-source software components. Finally, a more detailed look into an AWS-based Lakehouse architecture is presented as a case-study for a tried and tested production-grade architecture which would meet the requirements of the project.

4.1 Implementations

While these architecture patterns provide a strong and well reasoned high-level view of how the data flows should be processed, the actual implementation can vary greatly depending on which general approach is used. To discuss this, two implementations will be briefly presented: the *Kappa Architecture* as implemented by *Uber* [107], and the **Lambda Architecture** as implemented by *SmartNews* [103]. While these cannot be considered reference architectures, they provide a point of view into the actual implementation of these architectures, and how they are currently being used in production.

4.1.1 SmartNews Lambda Architecture

This implementation of the Lambda pattern leverages the diverse **Amazon Web-services** [8] technology suite to build a massive, highly available and performant platform. Figure 4.1 displays a technology-component based view.

This architecture was designed in 2016 using the available technologies, and while today it may present several inefficiencies, it was quite powerful for its time, pro-

viding near real-time user queries and interactivity, and enabling a large amount of flexibility in the output layer, with many native integrations being possible with popular *Business Intelligence* dashboards and team-management tools (such as Slack).

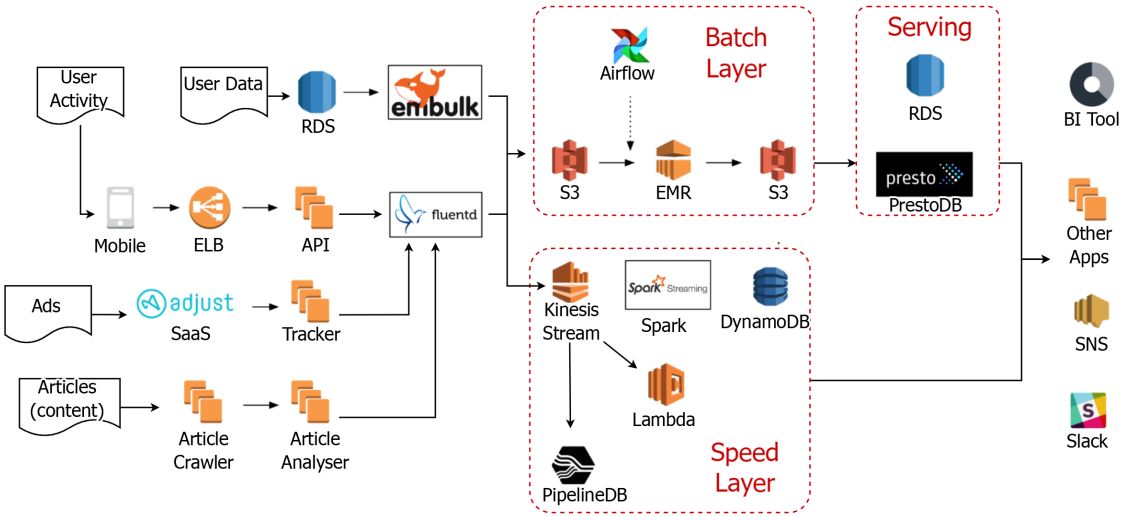

Figure 4.1: The technology-component diagram of the SmartNews architecture, featuring the Input, Batch/Serving, Speed and Output layers and their main data flow paths (adapted from SmartNews, 2016. [103])

Using a variety of input technologies, both batch and stream data processing takes place. To briefly describe the data processing pipelines, a data-centric perspective is presented for each of the data flows:

- **Batch Processing** - In the Batch layer, data is stored for historical purposes (in an *Amazon S3* storage service). It may then be subjected to data-set scale transformations using an appropriate processing technology (in this case, an *Amazon EMR*-based solution) before being recorded in its new state. It is then served to external services through the use of powerful query and reporting engines (*PrestoDB* for queries and *Amazon RDS* for database-storage).
- **Stream Processing** - As data is pushed into the speed layer, a streaming service (*Amazon Kinesis Stream*, in this case) is employed to move it to its destination, be that a *direct serving* path (through *AWS Lambda* serveless actions) or through a *transformation-enabled path* (in this case, using *Apache Spark* as its main ETL/ELT engine, and *DynamoDB* as the supporting database system).

This implementation is particularly interesting because, by using the Amazon Webservices (AWS) managed technology stacks, most of the overhead associated with the dual platform approach is avoided, since AWS features native integrability and inter-operability between all its services, as well as scalable *backends* with a common unified dashboard.

4.1.2 Uber Kappa Architecture

One look at a successful Kappa architecture implementation is Uber's solution. Using **Apache Kafka** [24] as a centerpiece, it uses this massively distributed messaging system to create both a realtime and a batch-oriented pipeline. Figure 4.2 presents a very summarised high-level view of the architecture.

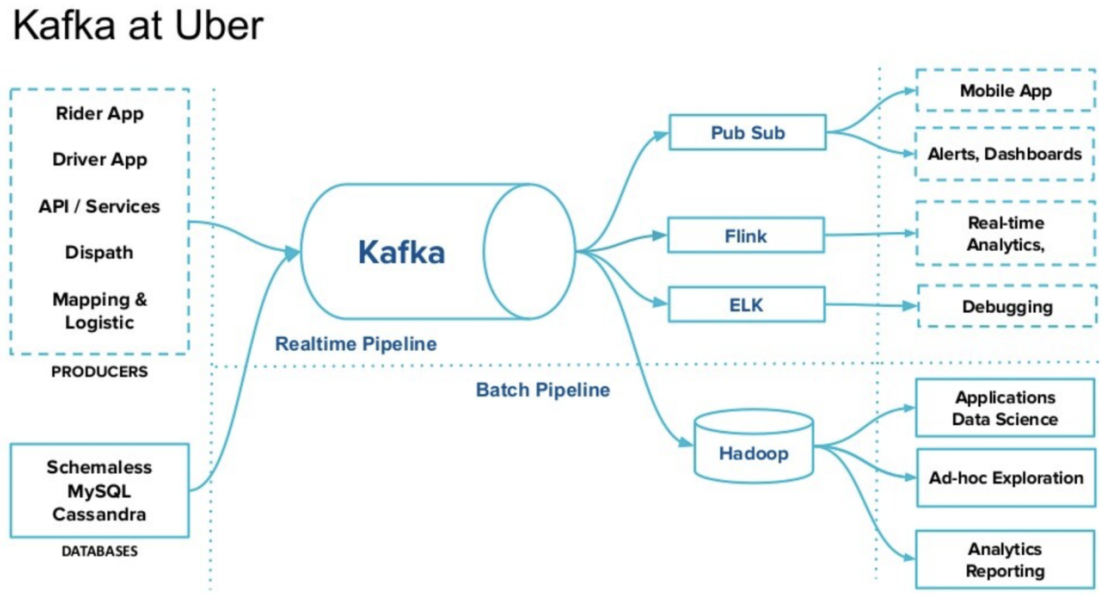

Figure 4.2: The technology-component diagram of the Uber Kappa architecture, showing the data sources, the ingestion and processing pipelines, as well as the destinations for the processed data (Kai Waehner, 2021 [107])

.

By hosting the entire ingestion pipeline as a single real-time process, some benefits are apparent, namely the simplicity of the overarching design. Both types of data processing are still supported:

- **Batch Processing** - Data moves in through the real-time pipeline (*Kafka*, in this case) and is stored for later use (*Hadoop*). Additionally, re-transmission is supported, enabling full data-sets to be processed at will through the realtime processing pipelines as an alternative to dedicated batch processing systems.
- **Stream Processing** - Data is passed through (*Kafka*) and is provided to the numerous systems which require real-time data feeds. In this case, a pub-sub consumer is used for alerts, dashboards and the mobile app (enabling distributed communications through a lightweight messaging protocol), a highly performant stream processing engine is used for real-time analytics (in this case, *Flink*), and a log processing stack can be used (in this case, *ElasticSearch, Logstash and Kibana*, also known as the ELK stack).

The flexibility of this system has enabled Uber to grow tremendously, by simplifying the data processing stacks and attaching them to a high performance, highly scalable and fault tolerant native system, like Kafka.

4.2 Supporting Technology

To better understand and *chart* the technological landscape surrounding the development of Big Data software systems, a survey was performed on currently widespread solutions. This study intended to list out the major alternatives for different parts of the architecture, namely the *entry* of data into the system (ingestion), the *storage* of the data, and the *use* of the data (serving and consumption).

According to the requirements of the project, this description will focus mainly on the open-source alternatives which are readily available and highly documented, namely those belonging to the Apache Software Foundation [28], with brief mentions of managed/SaaS implementations. Numerous vendors offer either partial solutions or full suites.

While this chapter will present a surface-level view of these technologies, further more focused explorations (namely of their differences, drawbacks and advantages) are presented in Chapter 6 during the iterative design process. It is important to note that not *all* available or even popular technologies were researched due to time constraints, and the research effort focused mainly on technology which had previously been identified as part of the preliminary information gathering performed prior to this dissertation project.

Additionally, as a way to understand the design process of a modern cloud-native architecture, a case study was performed on an Amazon Webservices (AWS) native Lakehouse platform, showcasing the opportunities attached to a potential AWS migration and showing how the industry leader's solutions address the top requirements of a Big Data architecture.

4.2.1 Data Ingestion

There are many technologies fit for use an ingestion point for a more complex data analytics system. In a modern ingestion pipeline, some qualities and features are desirable:

- **Scalability**, to match variable input loads, growing business operations and legacy systems.
- **Integrity**, to ensure that data is moved correctly and without error.
- **Performance**, to guarantee realtime, near-realtime or correctly scheduled data ops.

In the study for this application, some technologies proved to have a high potential; they are mapped in Figure 4.3, showing both on-prem solutions and managed options.

Amongst all others, **Apache Kafka** [24] is the prime candidate, featuring a robust, simple architecture of coordinated brokers, passing messages from producers in

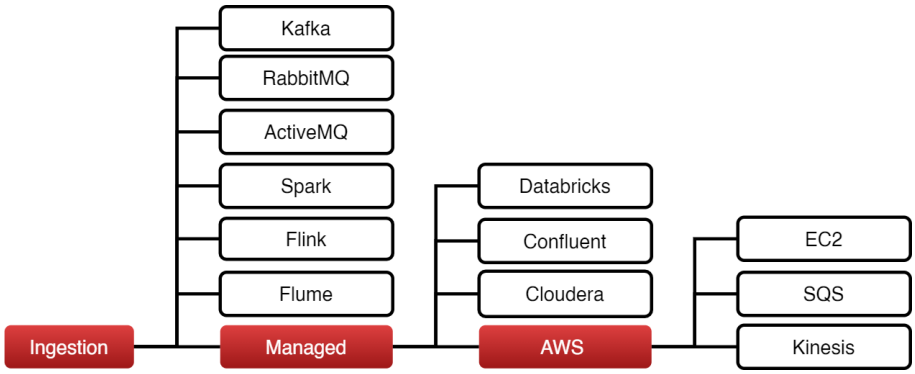

Figure 4.3: Map of Data Ingestion technologies which were analysed as part of the preliminary research. Coloured nodes indicate categories.

a publish-subscribe configuration, and building an ordered, immutable log of all passed messages, which is synchronised with the consumers. It is scalable by design, highly performant and available, and has become the standard in data streaming operations.

Additionally, attached to Kafka there are a number of additional "modules", in the currently expanding *Kafka Ecosystem*: **Kafka Connect** (data integration framework to connect to external data sources/destinations), **KSQL** (real-time SQL based data processing for Kafka flows) and **Kafka Streams** (a processing application tool, which enables the development of complex data pipelines using Kafka as a backend).

Other messaging systems, like **RabbitMQ** [100] and **ActiveMQ** [4] also provide scalable messaging, albeit in slightly different ways than Kafka, mostly related to how messages are handled. Whereas Kafka creates an immutable log and propagates it to the destinations, RabbitMQ and ActiveMQ treat messages as atomic entities, not allowing for the same kind of record-keeping natively.

Apache Spark [25] is a distributed computing engine which has many potential uses, data ingestion being one of them. Spark can read data from many different file formats, and pull them for analytics, ETL and any further processing. **Apache Flink** [30] is a similar processing framework that, while sharing many qualities, focuses mainly on *stream processing*, whereas Spark has a much bigger emphasis on *batch processing*. Additionally, Apache Flume [27] is an ingestion tool used primarily for log/event data collection which is commonly used for data transit wherever processing is not required.

On the managed side, **Confluent** [46] offers a managed Kafka service, using highly fault tolerant mechanisms to ensure high availability and throughput, making it one of the top choices for massive Big-Data ingestion pipelines. If a larger, more diverse suite is required, both **Databricks** [49] and **Cloudera** [45] offer fully featured Lakehouse platforms with robust open-source based ingestion pipelines.

Within the managed ingestion processes, a special distinction is made for the Amazon Webservices (AWS) native solutions: **Kinesis** [15] and **SQS** [19]. Kinesis is a highly scalable messaging system that shares a lot of traits with Kafka. The

main difference between them is the way they store the log data - Kafka uses file-based logs, whereas Kinesis uses sharding and high-speed databases.

4.2.2 Data Storage

The search for technologies fit for the underlying storage systems of the target architecture was framed around the functional distinction between the Lakehouse (which is effectively a metadata layer) and the physical systems below (namely the Data Lake). Research centered around two monolithic centerpieces - **Hadoop/HDFS** [22] and **Min.IO/S3** [90] - which provide the *raw* storage solution upon which abstractions (such as the aforementioned data management systems) can be built.

Several solutions were researched, as presented in Figure 4.4, presenting both self-deployed (on-prem or cloud) and managed alternatives.

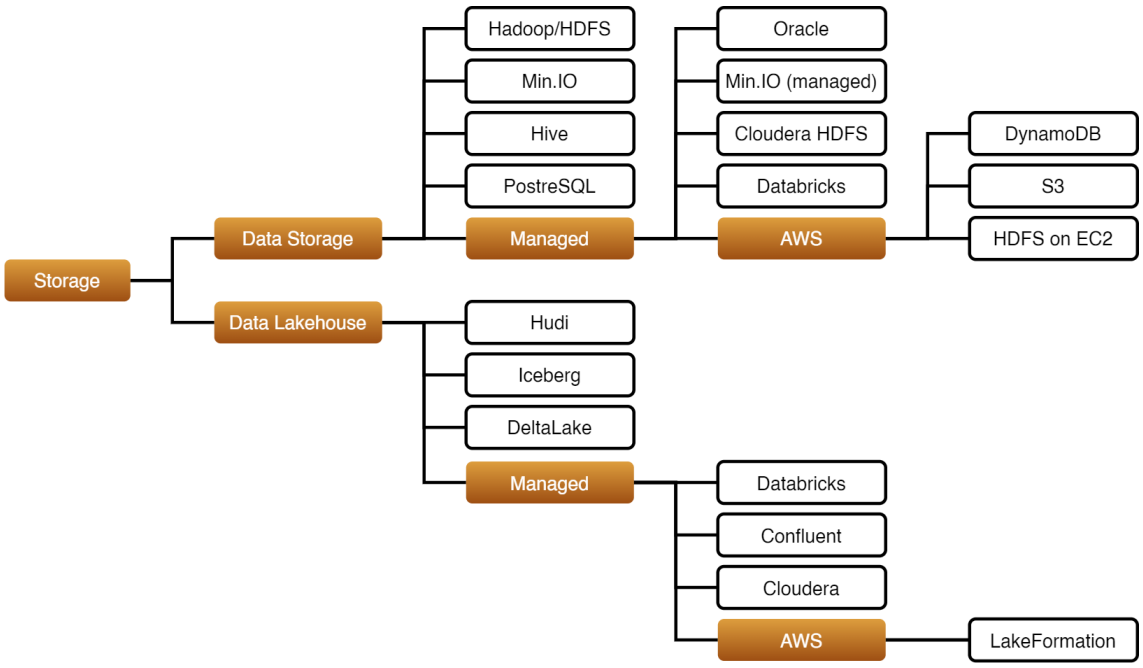

Figure 4.4: Map of Data Storage technologies which were analysed as part of the preliminary research. Included are both raw storage and lakehouse-ing technologies. Coloured nodes indicate categories.

As previously mentioned, the main components under analysis were **Hadoop and the Hadoop Distributed File System (HDFS)** and **Min.IO/S3**. They functionally achieve the same goal - massive format-agnostic data stores with rich APIs to manipulate the data within, while offering the tooling and access paths necessary to use it effectively.

However, they are fundamentally different in how they achieve this:

- **Hadoop** creates a *distributed file system* which is deployed in controlled clusters, creating what essentially amounts to a monolithic storage service that is well suited to the **MapReduce**[52] processing method, a model which uses a master-worker dynamic across clusters to perform tasks in a granular manner, allowing for parallelization of data processing tasks on a large level. Many data processing suites (namely those in the Apache foundation) have been designed to use MapReduce natively in their processing jobs, ensuring a high degree of compatibility with this technology, creating a very rich *ecosystem* of tools surrounding Hadoop.

- **Min.IO/S3** is a more lightweight solution, which simplifies the data store by removing the hierarchical imposition of a file-system, relying instead on the **object-storage concept**. In object storage, data is encapsulated within an object, along with all its associated *metadata*, and all objects are stored and accessed in a flat *namespace* (i.e. through unique keys). This removes a lot of the complexity of previous large data-stores, and creates a simple to store data with native scalability. This makes it well suited to Cloud-Native environments, where scalability and elasticity are core requirements.

In summary, Min.IO presents a more lightweight alternative to managing large data-sets, at the cost of not having tailor-made highly performant processing stacks like those found in Hadoop's ecosystem, having been at the center of data processing architectures since its inception [47]. Despite this, while Hadoop's ecosystem presents a significant advantage in the process of architecture design (providing ample amounts of components with native compatibility and integration), recent developments, such as the wide-spread adoption of S3 API support, have created opportunities to transition from the stiff and complex HDFS-based systems to a more flexible, cloud-friendly approach with Min.IO.

As part of this study, other storage systems were also analysed, as either standalone solutions or building-blocks to assist in the creation of a diverse, hybrid architecture. **PostgreSQL's** [98] open-source database is still considered one of the *de-facto* database solutions for distributed relational database management systems. Additionally, to provide data warehouse-ing capabilities, **Apache Hive** [23] takes a metadata-driven approach upon the Hadoop file-system, enabling high-speed queries and analytics tools in a SQL-like language.

Most of these services are also available in managed form, with the previously mentioned **Cloudera, Databricks** companies along with **Oracle** [96] or even the **Min.IO SaaS** deployment, which provide these components as a web-service; and AWS offering **S3** [18] as one of their most popular and featured services, along with possible HDFS deployments on their container-based deployment system (**AWS EC2** [14]) or even SaaS database deployments like AWS' **DynamoDB** [13].

Data Lakehouse

As previously described, the Lakehouse presents a unique opportunity - harness the flexibility of the Data Lake, while maintaining the performance and quality of a Data Warehouse. Functionally, this is done through a metadata system (which may be an additional storage layer, a table or an abstraction-based system) which serves as the interface for all data access.

Several technologies present options for this use-case, all centered around the creation of compatibility between large unstructured Big-Data storage and highly structured, schema-reliant data processing engines. While all technologies which are presented seek to add the same features (ACID compliance, Metadata management, Scalable data management), they all do so in different ways, with different drawbacks.

Delta Lake [54] is one such platform, creating a three-stage data management system (*ingestion*, *refinement* and *aggregation*) with strong integration with existing ingestion stacks (like *Spark, Kafka*), storage systems (*S3, Hadoop, Hive*) and serving platforms (*Spark, Presto, etc.*). It is primarily suited for *Spark* workloads since that is its most developed connector.

Apache Hudi [26] is a storage-abstraction based solution which creates a metadata table as data is streamed into the system. Hudi has very good integration with many query engines and ingestion platforms. While Hudi is still very new, it is incredibly promising due to its flexibility, relative simplicity and suitability for both batch and streaming workloads. Hudi possesses a great degree of integration with *Spark* and *Flink* on the ingestion side, and with *PrestoDB* and *SparkSQL* on the serving side.

Apache Iceberg [31] takes a fundamentally analytic-centered approach, being built around the standard lakehouse feature-set but with a clear focus on enabling large scale column operations and schema-evolutions (changes to data structure) on massive data-sets.

Meanwhile, on the managed side, all the previously referred entities (**Databricks, Cloudera, Confluent**) provide their own metadata governance frameworks (with Confluent focusing more on the issue of data stream governance options). **AWS** offers a complete platform in the **LakeFormation** [17] service, an **S3** native service which uses schema crawling[1] and metadata catalogs to create a governance framework around the data lake.

It is worth noting that because the aforementioned lakehouse platforms run on top of storage systems, it is possible to host them within a managed environment (like S3 or Databricks/Cloudera cloud stores) freely, albeit taking into account the increased processing power required to maintain the metastores associated with the greater degree of metadata transit.

[1]The process of inferring and/or imposing schemas on unorganized and unstructured data stores.

4.2.3 Data Serving and Consumption

For the serving portion of the data processing framework, several technologies were researched as potential solutions, most of them centered around the queries used to pull data from the Storage layer, either directly or through the Lakehouse meta-stores.

This inevitably honed our research toward technologies which presented native compatibility with the previously mentioned Lakehouse technologies, as that is what the queries would effectively be performed on in the production use-case.

Figure 4.5 shows the tech-map of the analysed components:

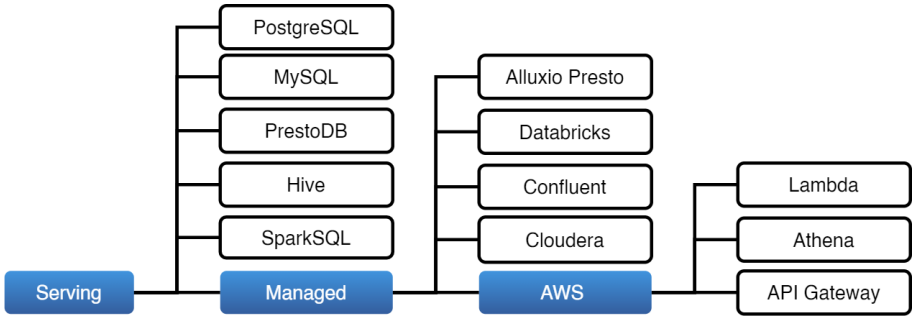

Figure 4.5: Map of Data Serving technologies which were analysed as part of the preliminary research. Coloured nodes indicate categories.

Paired with the use of databases as storage components, the use of **PostgreSQL** [98] and **MySQL** [91] is logical, as both present scalable and highly-performant solutions for *SQL-compliant*, *ACID-compliant* querying. These technologies, through the use of Lakehouse components, are usable even in the Data Lake context (although typically more modern, cloud-native serving components are used).

Cloud-native and oriented solutions, like **PrestoDB** [99] present innovative solutions to the task of massive, highly performant querying through the use of meta-data stores (which can provide higher-level mappings and ACID/SQL-compliant data serving) and a coordinator-worker distributed architecture. As a plus, all of the researched Lakehouse services feature native, low-code integration with PrestoDB, enabling their internal *metastores* to synchronise with Presto's query engine seamlessly and automatically. Presto also supports open data formats, like *Parquet* and *Avro*, making it ideal for use with Lake and Lakehouse-ing technologies.

The previously mentioned **Apache Hive** warehouse system also presents a SQL-based query engine implementation using its own internal metadata store, although lacking in support for unstructured or semi-structured data formats.

Apache Spark also features a separate module, built to run natively and in a scalable manner in Spark execution clusters - **SparkSQL** - allowing for easy to use SQL queries. SparkSQL presents an interesting opportunity due to its strong integration with the analysed Lakehouse technologies.

On the managed side, the previously mentioned providers (**Cloudera, Databricks,**

Confluent also support their own in-house query engines, featuring high-speed, scalable and highly available engines. Along with these, **Alluxio** [6] is a cloud-service provider which presents a unique version of PrestoDB with massive performance optimizations for Big-Data workloads, using a caching system built upon their proprietary "*Alluxio Distributed Filesystem*" [5] which is a storage-level abstraction for increased performance. **AWS** solutions rely on a high-degree of integration with the AWS suite, and using scalable and elastic access methods, like APIs (such as the **AWS API Gateway** [11]) or other *server-less* systems, like **AWS Lambda** [16], which enable the creation of RESTful[2] query services and remote-code execution pipelines using integrated internal AWS technologies like *DynamoDB* queries and **AWS Athena** [12].

4.2.4 Administration and Data Governance

Modern data architectures require a large focus on administration, governance and auditability use-cases in order to meet regulatory compliance. While these concerns could theoretically be handled through organizational structures and paper-trails, it is impossible to scale these processes to the size and volume of a Big-Data workload without the use of supporting technology.

To ensure compliance, two optics are considered: Data *Access* (access control technologies) and Data *Governance* (the use of metadata management systems and catalogs). Access control stacks form the basis of a compliant system, making sure the only people who see the data are correctly authorized individuals; and metadata management systems enable data discovery use cases in a scalable manner while providing numerous compliance options (lineage tracking, integration, policy-based retention).

The researched technologies are identified in Figure 4.6.

In terms of the metadata management systems (or simply put, the catalogs), three main options appear as self-deployed solutions: **OpenMetadata** [95], **Datahub** [2] and **Atlas** [29]. Out of the three, Atlas is the only one that does not currently offer a SaaS option.

These solutions all present the same core functionalities and goals - to build governance opportunities through purpose-built systems, using extensive integration to gather metadata at all steps of the data framework (ingestion, storage and serving), and create an easily usable and accessible tool to assist in auditing, traceability and data discovery use-cases.

OpenMetadata uses an entity database (**MySQL**) and an implementation of **ElasticSearch**[3] to create a UI which holds all data entities within a system, their respective mappings and relationships, and even their lineage, previous versions, etc. It has its own internal schemas and operates over a vast amount of connec-

[2]Built to be used through HTTP, following a standardised set of rules and conventions to enable remote utilisation.

[3]Elasticsearch is a powerful, distributed search and analytics engine designed for fast and scalable retrieval of data from large volumes of diverse sources

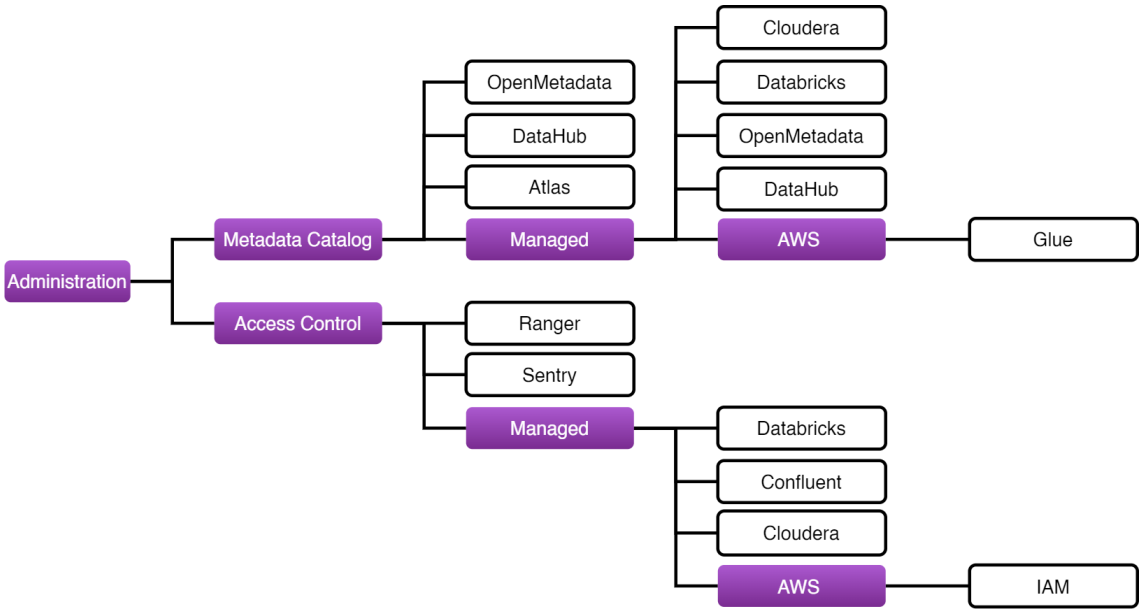

Figure 4.6: Map of Administration and Data Governance technologies which were analysed as part of the preliminary research. Both metadata catalogs and access control modules are encompassed. Coloured nodes indicate categories.

tors (namely supporting all the identified components present in the described research).

DataHub is very similar, although it focuses more on developing new integrations (with an easy-to-design connector system), and providing, generally, the same features as OpenMetadata, albeit in a simpler package with a more user-friendly interface. The other alternative, **Atlas**, is similarly featured, but restricted by its focus on the HDFS/Hadoop platform. Integration with cloud-native object storage is one of Atlas' main weak points and its reliance on the HDFS ecosystem makes it a less favourable choice from a flexibility point of view.

On the managed side, most cloud-vendors offer some form of metadata catalog (generally proposing Apache Atlas implementations), with **AWS's Glue Catalog** being the indicated component when using a mostly AWS-based stack, due to its native integration with all other services through "one-click" configuration. It is worth noting that while **AWS Glue** is a metadata catalog, it possesses some features more akin to a Lakehouse component (such as direct data access through the Catalog), which are not typically supported by metadata catalog components.

These components enable the creation of a tentative framework for exploring data governance and integration on a much larger scale, especially through the interoperability between the data discovery tools and the access control stack.

4.3 Case Study: Amazon Webservices Lakehouse

As part of the research process of technologies and current solutions, a special degree of attention was given to the solution proposed by **Amazon Webservices**

[9] for the Lakehouse paradigm. This fully featured and tailor-made architecture aims to serve all the main requirements of the Lakehouse: *Scalability*, *Governance*, *Performance* and *Security*; and it provides the components (all within the AWS suite) to achieve them.

This analysis focused on identifying technologies and how a reference Lakehouse framework would be performed in a cloud-native setting. While it may be valuable to analyse each individual technological solution and understand their core principles and integration, the scope of this work and the associated time constraints resulted in a focus on a more surface-level approach. Figure 4.7 presents a container/service view of the architecture.

It uses a layered approach along with a hybrid storage solution encapsulated by a Lakehouse layer:

- **Ingestion Layer** - Ingest data into the Lakehouse storage layer, connecting to internal and external data sources. This layer can deliver batch/stream into warehouse/lake systems.
- **Storage Layer** - Responsible for providing durable, scalable and cost-effective elastic components using a hybrid approach - a Data Lake and a Data Warehouse, to support a variety of activities using a zone-based system (*raw, trusted, enriched, modeled*)
- **Catalog Layer/Lakehouse Layer** - Store of business and technical metadata related to the datasets below. This is the main engine for constructing the queries, by supplying schema information, granular partition information and data discovery opportunities through an elastic search engine.
- **Processing Layer** - The ETL transformation layer which provides validation, cleanup, normalization and enrichment operations to the many data flows within the framework.
- **Consumption Layer** - Query systems, distributed ML Jobs and services that combine data from multiple end-points to serve BI and ML-analytics services. Supports multiple user access levels.

While the analysis of the architecture centered mainly on technology identification and registration into the knowledge base for the architecture design process, some interesting structural considerations were observed:

- **The use of a Warehouse and a Lake in the Lakehouse architecture**, which is not a fundamental requirement of the Lakehouse's formal definition (as the Lakehouse paradigm merely seeks to add Warehouse traits to the Data Lake) seems to indicate that this architecture is capable of a great degree of flexibility in the type of storage systems that it supports. By providing a toolset for integrating Lake data into the Warehouse and vice-versa, a hybrid solution is achieved, providing cost-effective storage and powerful enrichment pipelines.

- **The use of schema crawlers in the Data Lake** provides an interesting way to perform **schema evolution** on varying data flows and stores. This feature has a lot of potential to drive significant value in terms of data quality management.
- **The use of Identity and Access Management (IAM) and Access Control** services at every step of the framework is not something typically seen, but necessary in the AWS solution due to the single dashboard that is accessible by all the users. This pattern may be useful in architectures where a centralised tailor-made dashboard is required.

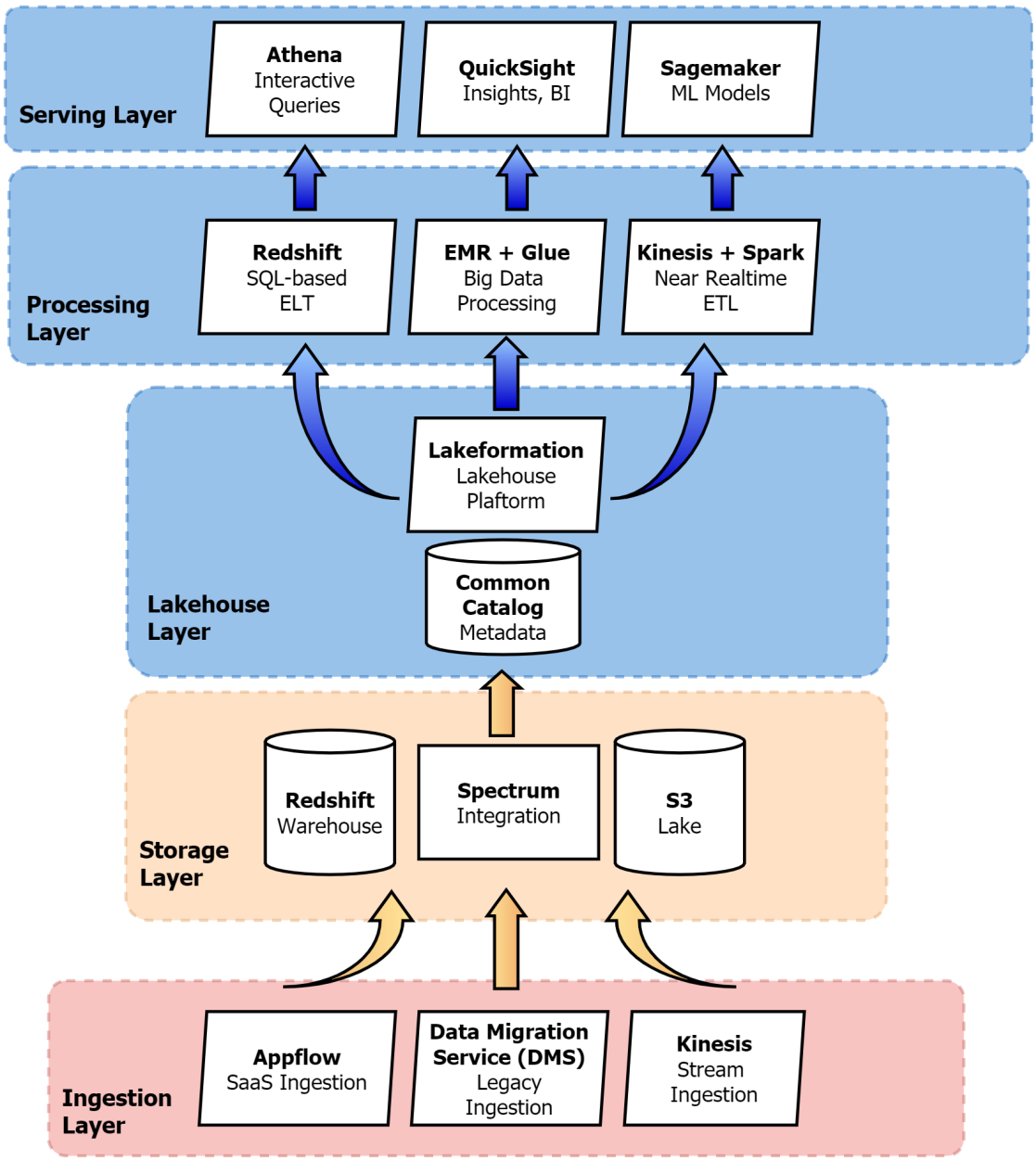

Figure 4.7: Technology/Service-centric view of the Amazon Webservices Lakehouse Reference architecture. Flow of data is from bottom to top.

Chapter 5

Requirement Analysis & Specification

To lay the foundations for the development of the architecture, it is crucial to understand the drivers which will motivate the decisions made in the design process. In short, the stakeholders' desires, needs and goals must be reflected in the outcome, their requirements must be analysed, categorized and refined into a format which can guide development during the architecture phase.

As previously described in Chapter 3, these requirements are grouped into **functional** and **non-functional** requirements. Alongside these descriptions, the identified **business** and **technical** constraints are also analysed to outline the regulatory rules and restrictions, and any limitations inherent to this project. This information is presented in tabular form. The sources of the requirements are identified as mapped in Table 5.1.

To present the requirement specification process, firstly a broad description of the system is performed following a description of the architecture's partitioning. Secondly, the constraints attached to this system are presented briefly, followed by general descriptions of the requirements associated of each layer. The requirement specification is integrally presented in Appendix A, with this chapter serving as an abridged version of the requirements of the system.

Source ID	Description
PRE	Extracted from preliminary documentation by ALB.
REQ-0	Outcome of initial refinement (pre-draft).
REQ-1	Requirements in the first iteration.
REQ-2	Requirements in the second iteration.
*	Revision or small refinement/adjustment.

Table 5.1: Source ID mapping and description for table entries.

5.1 System Description

With the previously provided context for the system's purpose and general goals, an in-depth description of the system is required, to set the stage for the architecture design process.

The system's main objective is to enable the processing of *large quantities of data* in a **multi-tenant** capable, highly-**scalable** and **available** context. Data is received from source systems, with varying types, sizes and formats, and is passed on to storage systems, for later use by the system's various business endpoints (e.g. data analytics, visualization, ML, etc.).

Throughout this "functionality" chain, there must be provisions for data quality monitoring, traceability and auditability, as well as considerations for infrastructure monitoring, automation and system health checking. These features will allow the system to adapt to the multi-tenancy context, involving external companies and clients in a compliant manner (especially with regards to data privacy regulations) through strict access control, domain *lock-out*/management and auditability.

5.1.1 Functional System Partitioning

To design the architecture and organise the requirements, it will prove useful to create logical units that encompass sets of functionalities and qualities. Following the initial documentation provided by Altice Labs S.A. there are three views to consider:

- **Functional View** - Those pertaining to the main intended functionality - data processing. This includes a further subdivision:
  - **Ingestion** - The entry of new data into the system.
  - **Storage** - The storage and management of large data volumes.
  - **Serving** - The presentation and consumption of data by external services.
- **Administrative View** - Pertaining to the regulatory constraints and encompassing the components which enable compliance.
- **Orchestration View** - Encompassing the components responsible for monitoring, automation and alerting of the system's execution.

So, to describe the system, a **five-layer** partitioning is performed: *Ingestion Layer*, *Storage Layer*, *Serving Layer*, *Administration Layer* and *Orchestration Layer*, following the aforementioned perspectives. The requirements will be developed and organised according to this partitioning, which, in itself, evolved over the course of the semester, initially with a three layer partitioning, before maturing into its current form after receiving feedback from Altice Labs S.A.

5.1.2 Functional View

As previously discussed, this perspective encompasses three layers - the **Ingestion Layer (IL)**, **Storage Layer (SL)** and **Serving Layer (SV)**. These layers are where the majority of the system's data flows will occur, bringing data in from legacy systems and new sources, transforming it, storing and then delivering it to its intended endpoint. Figure 5.1 presents a simplified view of the Functional View, with its main tasks highlighted, layer by layer.

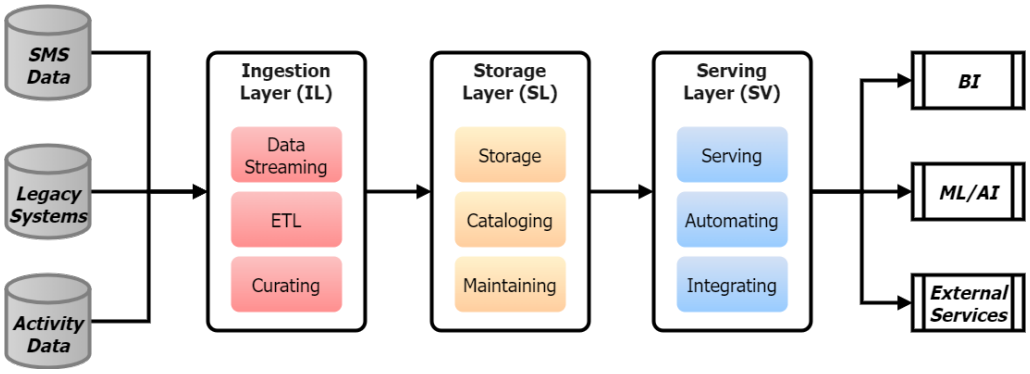

Figure 5.1: Schematic representation of the functional view of the architecture. Data flows are represented by arrows, and within each layer the main functionalities are identified.

Data which is streamed in from external sources is passed through the IL, receiving optional transformations and possibly even being curated based on its quality or other metrics.

Afterwards, the data is passed to the SL, where storage will take place. This storage should include some form of cataloguing as per the Lakehouse pattern, to allow for mixed data storage with warehouse-level performance.

Finally, in the SV, data is pulled from storage. This layer supports automation of pulling and querying, and provides the necessary tools for data integration, connecting the data to external services and data sharing. The SV will also encompass access control requirements, by virtue of being the external interface of the system.

The main drivers of these layers can be identified as:

- **Ingestion Layer** - *Performance* and *Scalability* in the data streaming operations to handle the variable loads and peaks of daily operations, especially in the multi-tenant environment.
- **Storage Layer** - *Availability* and *Integrity* of the storage systems, and the ability to ensure that data is never compromised even under hardware failure.
- **Serving Layer** - *Security* and *Traceability* of user actions, to create a stable and compliant data-sharing environment.

5.1.3 Administration Layer

On top of these functional components, the **Administration Layer (AL)** serves as an all-encompassing set of components that will enable the management of data throughout the entire lifecycle. Figure 5.2 provides a simplified look at the main tasks supported by the AL.

By leveraging modern technologies such as AI, Knowledge Graphs and more, it is possible to use metadata derived from all the layers of the system to build a characterization of the many data flows for auditability and traceability-related tasks.

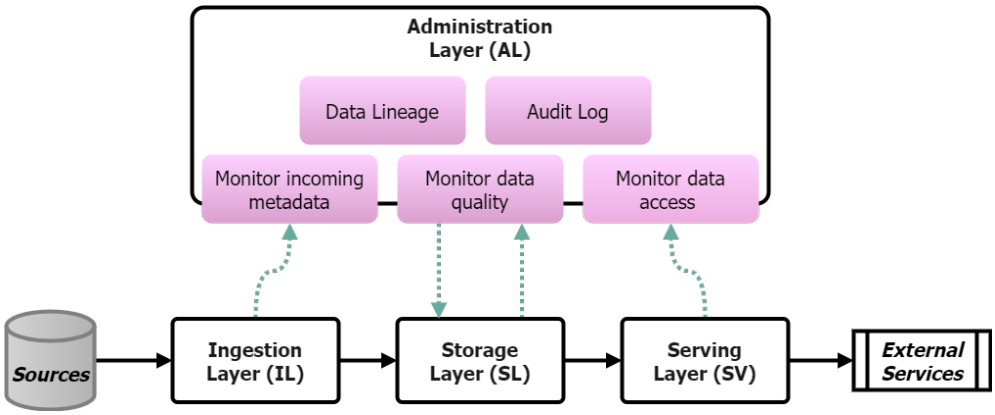

Figure 5.2: Schematic representation of the administrative view of the architecture. Metadata flows are represented with dotted green arrows. The data sources and endpoint external services were joined into a single component for this view.

Through these functionalities, it is possible to maintain an extensive audit log and use it to ensure compliance, as well as assist in tasks related to data quality and understanding how data changes throughout its lifecycle by monitoring all changes made to data objects and integrating them into a comprehensive lineage view.

The main drivers for this layer are **Isolation**, **Privacy** and **Usability**:

- **Isolation and Privacy** - In ensuring that data is not widely available to anyone who seeks it, and that each domain/team/user has their own set of data and access is controlled.
- **Usability** - In creating an environment that can facilitate integration, data sharing and inter-operability between domains in a safe, compliant way.

5.1.4 Orchestration Layer

In parallel with the management of the data, there must also be a feature-set for managing the system's execution, and to ensure that scalability, performance and productivity does not suffer during the various states of operation the system may be subjected to.

In the **Orchestration Layer (OL)** components work together to ensure the system's health, through a combination of logging, log analytics and automated maintenance routines. Figure 5.3 provides a simplified view of the main activities that take place in the OL.

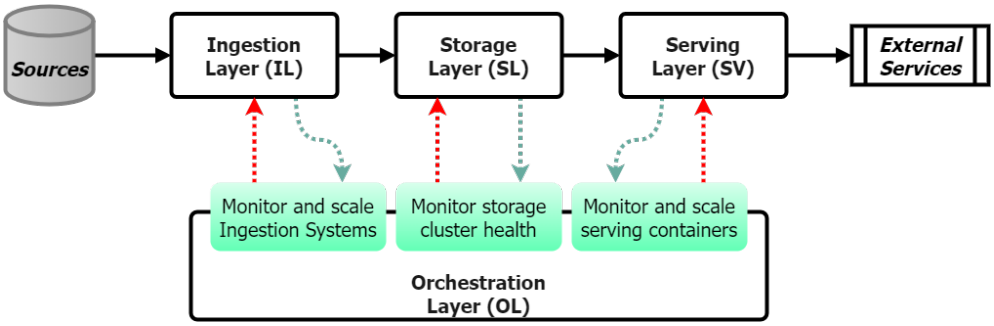

Figure 5.3: Schematic representation of the functional view of the architecture. Data flows are represented by arrows, and within each layer the main functionalities are identified.

To support these functionalities, distinct message queues/pathways must be defined, to ensure long-term scalability and flexibility of the monitoring and management solution. The potential for this layer is significant, since it may use its role as overseer to significantly automate the management and health checking of the system through the configuration of alarms, automated actions and reactions to system events. Taking this into account, the key drivers for this layer are **Consistency**, **Modifiability** and **Scalability/Performance**:

- **Consistency** - In logging, monitoring and maintaining the harmony of the system through constant, real-time analysis of system health data.
- **Modifiability** - In adding new monitoring components, new analytics operations, action automation and even registering new components into the monitoring suite. These needs must be met, allowing for a resource-conservative approach to management and maintenance.
- **Scalability/Performance** - In ensuring harmonious operation at all load levels, the monitoring, rate-limiting and automation must be able to perform, and scale up/down whenever necessary, maintaining an elastic profile.

5.2 Constraints

The project's constraints were identified mostly from the preliminary documentation supplied by Altice Labs S.A. and the Architectural Driver Elicitation Workshop (ADEW) which took place prior to the start of the dissertation.

Throughout the semester, several constraints were altered, namely the Business Constraints, which were refined to better represent the privacy-related issues brought up over the course of the project.

Because this solution encompasses the functionalities performed currently by a large number of systems, the technical constraints associated with this project are generally regarding how the new system will get its data (from source systems) and how it should present it (to endpoint services).

5.2.1 Technical Constraints

The technical constraints attached to this project are centered around the existing solutions at Altice Labs S.A., and are designed to facilitate the integration of the new solution into their currently used technology stacks and legacy systems. As was described in Chapter 3, Section 3.3.1, these restrictions are *hard line* predefined decisions which must be accounted for in terms of compatibility.

By considering these restrictions when choosing components, and ensuring compatibility with the new solutions, it is expected that the changes which occur with the implementation of the new system will be significantly easier on the organization, even if, in a future revision, better options are found.

ID	Source	Description
TC-001	PRE	SMS/Phone data is accessed by Cube Navigator/MicroStrategy
TC-002	PRE	SMS/Phone data is stored in Greenplum
TC-003	PRE	SMS/Phone data is accessible via REST API and MongoDB
TC-004	PRE	SMS/Phone data is monitored via Prometheus Alertmanager
TC-005	PRE	BI data is received from CSV and Oracle databases
TC-006	PRE	BI data must be available via SQL interface
TC-007	PRE	BI data flows/storage processes must be compatible with the Prometheus toolset
TC-008	PRE	Internal clients (helpdesk, etc) receive data from a Kafka MQ

Table 5.2: Technical constraints identified for the project.

5.2.2 Business Constraints

The business constraints identified for this project (Table 5.3) are centered around two core goals:

- Ensure that the system can support the growth into a multi-tenant scenario.
- Ensure that regulatory compliance can be achieved and maintained.

Due to privacy regulations, and in order to protect customer privacy, some considerations must be taken when choosing components, namely the fact that they must support privacy measures, and meet the security requirements of the GDPR, with provisions for auditability of user actions. This will become particularly relevant when external clients initiate operating on Altice Labs S.A.'s solution. When this shared operation begins, new concerns emerge regarding cost management, both internally (i.e. keep internal costs down, cap energy spending or resource usage) and externally (i.e. create packages and service-level agreements for external parties).

ID	Source	Category	Description
BC-001	PRE	General	The system must evolve into a multi-tenant cloud-based architecture, with one deployment for several client provider.
BC-002	PRE	Privacy	The mandatory security requirements of the GDPR are fully met.
BC-003	PRE	Costs	Software components included in the architecture should prioritize accessibility and focus on low-cost alternatives (free-to-use being especially desirable)
BC-004	PRE*	Costs	Costs should be controllable, either by limiting resource usage or by creating cost ceilings.
BC-005	PRE	Training	Barrier of entry for software component usage/modification should be as low as possible, with good documentation and support.
BC-006	PRE	Costs	Software components should, ideally, use open source software.
BC-007	REQ-1	Privacy	The software manages personal and non-personal data in separate ingestion, storage and serving streams.
BC-008	REQ-1	Privacy	Personal data must be anonymized through Generalisation, Randomisation or Masking.
BC-009	REQ-1	Privacy	Personal data must not be kept for a period longer than useful or necessary for business operations.

Table 5.3: Business Constraints as identified for the project.

5.3 Requirements

Throughout the project, the requirements shifted and evolved, starting from their immature form, adapted from the ADEW's output and from initial contextualization meetings, into their final form, after many revisions and presentations, stakeholder feedback and research work. In this section, some of the identified requirements will be presented.

For brevity, the majority requirements will be omitted, and the functional and non-functional characterization of each layer will be presented instead. The full Requirement Specification is available in Appendix A.

The description of these requirements relies on three actors (apart from the system itself):

- **End-User** - The *client*, *data scientist*, *analyst*, a user that can operate within the system, but cannot manage it.
- **Data Manager** - A *data manager*, *Data Protection Officer* (DPO) or similar, a user that can manage the data, privacy policies and data sharing rules within the system.
- **Process Manager** - An *infrastructure manager*, *system admin* - a user that can interact with the system's back-end directly, coordinate and orchestrate the various components.

5.3.1 Ingestion Layer

As previously described the Ingestion Layer (IL) has been designed and envisioned as a set of scalable, highly performant components that move massive volumes of data from the source systems to their designated endpoint within the Storage Layer.

The functional requirements of the IL focus on the ability to configure the system's data streams through scheduling, duplicating, automating and readily modifying existing streams. By meeting these requirements, the cost of connecting a new data-source or forwarding data to a different place is vastly reduced, resulting in a much more flexible system.

ID	Actor	Source	Description
FR-IL-002	System	PRE	The system shall accept configurations for both streaming and batch data sources.
FR-IL-005	Process Manager	REQ-1	The process manager can configure automated pulling from external data sources
FR-IL-006	Process Manager	REQ-1	The process manager can configure scheduled continuous data acquisition (data streaming windows)

Table 5.4: Sample Functional Requirements for Ingestion Layer (IL).

This flexibility is complemented by the identified quality-attributes which the components of this layer must hold: **Scalability**, **Availability**, **Performance**, **Integrity** and **Compatibility**. These are the main traits of this layer's components, and result in a fast, reliable system for moving data into the framework without faults, problems or delays.

QAS-IL-003 - *Availability* - REQ-1
Source - Ingestion Queue **Stimulus** - A cluster of message brokers enter a failure state. **Environment** - At runtime, under normal operations. **Artifact** - Ingestion Layer
Response - Process Manager is notified, new components are instantiated to serve existing streams. **Response Measure**: The system does not incur in significant downtime (>30s) during cluster failure.

Table 5.5: Sample Quality-Attribute Scenario for the Ingestion Layer (IL). In this case, the quality-attribute scenario relates to Availability.

5.3.2 Storage Layer

After coming through the Ingestion Layer's streams, data arrives at the Storage Layer (SL), where it must be stored, organized, catalogued and treated, if necessary.

The functional requirements of the SL focus on features necessary to create a scalable and reliable data store. The ability to visualise data, snapshot, organize metadata and even provide transformations and ETL on incoming and outgoing data-streams give ample tools for data management, and create the ideal ground for a Lakehouse architecture.

ID	Actor	Source	Description
FR-SL-003	System	REQ-1	The system shall provide a functionality to create snapshots of data on-demand.
FR-SL-006	System	REQ-1	The system shall provide the ability to store data for a specified period of time.
FR-SL-007	System	REQ-1	The system shall allow the storage of structured, semi-structured and unstructured data.
FR-SL-008	System	REQ-1	The system shall enable scheduling of storage maintenance operations (de-duplication, backup, etc.)
FR-SL-010	System	REQ-1	The system shall continuously extract metadata from incoming data streams.

Table 5.6: Sample Functional Requirements for Storage Layer (SL).

Non-functional requirements for the SL focus on **Availability**, **Integrity** and **Reliability**, as all the storage system's data must be backed-up, secured and highly available to ensure harmony within the system. Additionally, some considerations are made for **Security**, to ensure that unauthorized access is promptly denied.

QAS-SL-005 - *Reliability* - REQ-1
Source - Storage Components **Stimulus** - One of the storage components ceases to function and communicate. **Environment** - At runtime, under normal operations. **Artifact** - Storage Layer
Response - Failure is detected, operations are handed over to replicated backups. Faulty service is restored. **Response Measure**: Data must never be lost, either through image backups or distributed replicas, and the original storage component must be rebuilt and repopulated fully within 1 day.

Table 5.7: Sample Quality-Attribute Scenario for the Storage Layer (SL). In this case, the quality-attribute scenario relates to Reliability.

5.3.3 Serving Layer

When data, within the Storage Layer, must be used, it is the Serving Layer's (SV) components which will bridge the gap between the storage and the endpoint services.

The functional requirements of the SV focus on convenience features such as automation, query flexibility (i.e. perform ad-hoc queries) and data sharing features to fulfil the more "Data-Mesh" oriented use-cases, enabling end-users to share their data with other clients through a number of filters and privacy-preserving techniques, and disabling access to unauthorized users outside of the relevant domain.

ID	Actor	Source	Description
FR-SV-001	System	REQ-1	The system shall allow for the automation of data transit (pushing, pulling) for external services
FR-SV-002	System	REQ-1	The system relays data from the Storage Layer through a message queue
FR-SV-007	Data Manager	REQ-1	The data manager can configure data transit for new external services.
FR-SV-008	System	PRE	The system shall present specific views of the data in the SL according to the user's access level/domain.

Table 5.8: Sample Functional Requirements for Serving Layer (SV).

Non-functional requirements for this layer focus on the qualities of **Performance**, for high speed querying and data fetching/pulling, and in **Security**, **Traceability** and **Consistency**, to ensure that user access is done according to the relevant rules and domain lock-outs, and that all user accesses can be traced for auditing purposes.

QAS-SV-003 - *Security* - REQ-1
Source - Data Manager **Stimulus** - The data manager wishes to configure new data views for a tenant's users. **Environment** - At runtime, under normal operations. **Artifact** - Tenant Interface
Response - The new views are available to the relevant end-users/domains/tenant. **Response Measure**: Propagation of new views takes at most 1 hour. Non-authorized users cannot access this view.

Table 5.9: Sample Quality-Attribute Scenario for the Serving Layer (SV). In this case, the quality-attribute scenario relates to Security.

5.3.4 Administration Layer

The administrative side of the framework will focus mostly on the traceability and auditability aspects of the use of data within the framework.

Functional requirements for this layer include the management of teams, domains and users and also the logging of user activity. This management is done also on the level of maintaining an external **data catalog**, which serves as a ledger for all the operations and metadata attached to the data within the system. This catalog may also be used to initiate data sharing contracts with external clients, by allowing them to browse the catalog and see, via the metadata, what data may be of use to them, without compromising on its privacy and security.

ID	Actor	Source	Description
FR-AL-001	Data Manager	REQ-1	The data manager can perform access control operations, defining who can access which domains through an IMS (Identity Management System)
FR-AL-005	System	REQ-1	The system shall perform global user interaction monitoring using daily logs pertaining to user actions (read, write, modify)
FR-AL-006	System	REQ-1	The system shall snapshot logs on a configurable basis and save them to LTS for auditability purposes.

Table 5.10: Sample Functional Requirements for Administration Layer (AL).

Non-functional requirements of this layer involve the maintenance of **Security**, **Privacy** and **Isolation**, key qualities of a compliant system, and with some interest put on issues of **Usability** and **Consistency** to ensure that operating this administrative environment is easy, and that its effects on the system are felt in a consistent way.

QAS-AL-004 - *Privacy* - REQ-1
Source - Tenant X **Stimulus** - Tenant X wishes to access data from Tenant Y's catalog. **Environment** - At runtime, under normal operations. **Artifact** - Management Dashboard
Response - If Tenant Y has created self-serve views, these may be used. If not, a request is made for the Data Manager to enable access. **Response Measure**: Tenant X cannot access Tenant Y's data beyond the views which Tenant Y provides (i.e. if Tenant Y does not enable the viewing of the first column of a DB, Tenant X will receive that DB without said column).

Table 5.11: Sample Quality-Attribute Scenario for the Administration Layer (AL). In this case, the quality-attribute scenario relates to Privacy.

5.3.5 Orchestration Layer

With the distributed system in execution, and with the organizational side of the framework being accounted for, what remains is the management of the infrastructure, which relies on logging, monitoring and alarm, to ensure a quick reaction to any adverse conditions.

Functional requirements for this layer generally revolve around the monitoring of the many components, the health checking and associated analytics and metrics. With this information, the process manager can automate tasks and backups, set up alarms, triggers and actions to ensure the harmonious operation of the framework. Additionally, some consideration is allotted to disaster recovery, ensuring that backups occur and that recovery is possible.

ID	Actor	Source	Description
FR-OL-002	System	REQ-1	The system shall perform status monitoring on all components
FR-OL-003	System	REQ-1	The system shall support the building of images/snapshots of log data and save them to LTS.
FR-OL-007	Process Manager	REQ-1	The process manager can access platform health data, status information and component metrics.
FR-OL-009	Process Manager	REQ-1	The process manager can configure alarms, triggers and actions, and automate them via connection to specific component(s).

Table 5.12: Sample Functional Requirements for Orchestration Layer (OL).

The non-functional requirements of this layer revolver generally around **Scalability**, **Performance** and **Availability**, key factors in the system's response to changes in the operating conditions, as well as matters of **Consistency** in how often the system should execute its backup operations.

QAS-OL-006 - *Consistency* - REQ-1
Source - Process Manager **Stimulus** - The process manager wishes to configure automated snapshotting for the system health logs. **Environment** - At configure time. **Artifact** - Orchestration Layer
Response - The logs are saved to LTS via snapshotting at configured time intervals. **Response Measure**: Discrepancies of time between snapshots are no greater than 0.1%.

Table 5.13: Sample Quality-Attribute Scenario for the Administration Layer (AL). In this case, the quality-attribute scenario relates to Consistency.

Chapter 6

Architecture Design

Following the description of the requirements, and the creation of a characterization which encompasses the functions and qualities of the system, the architecture process can begin. This process involves the decision-making and corresponding validation that will eventually result in a properly justified, correct and validated specification featuring the components, functionalities and qualities expected of the target system.

During this project, three iterations of the ACDM process were undertaken:

- **Iteration #0** - Draft/Notional Architecture
- **Iteration #1** - First refinement, *Ingestion* focused.
- **Iteration #2** - Second and final refinement, *Storage* and *Serving* focused.

The iteration logs presented in this chapter have the following structure:

1. **Analysis** - Critical review of the artifacts of the previous iteration, its issues and any changes which may have occurred.
2. **Refinement** - Description of the steps taken to evolve the architecture into its next form. Refinement is performed by *updating the requirements*, then *updating the architecture*, and finally *consolidating the current architecture*, its alternatives and design considerations. This step is called "Creation" for Iteration #0.
3. **Review & Validation** - Detailed analysis of the architecture based on the RAID issue tracker's output. Presented as a summary of the identified issues and the mitigation plan for each issue.
4. **Experiments** - Aggregated logs of the experiments performed during the iteration, presenting a description of each experiment, the experimental scenario/configuration and the experimental results.
5. **Result** - Summarised view at the iteration's outcome, presented as a layer-by-layer analysis along with a reduced architecture diagram.

6.1 Iteration #0 - Notional Architecture

Following the first draft of the requirement specification, enough information had been gathered to create a draft of the architecture. Because there is not yet a base for refinement, this iteration serves mainly to provide a quickly put together foundation which informs on a general level, giving a very basic understanding of that possibilities may arise as the architecture is further refined.

Three steps are undertaken in this initial iteration: the creation of the basic architecture layout, the review of this simple design, and a description of its general functionality.

During this time in the development process, the ACDM's strict and structured review processes had not yet been fully adhered to, which resulted in a very incomplete and crude verification, based only on documentation review and not using any kind of experimental validation. As such, the descriptions within this section will be far less detailed than those in the subsequent iterations.

It should also be noted that, at this stage, the Requirement Specification was still in a preliminary state, pending verification and feedback from Altice Labs. Due to time constraints related to the project's deadlines, it had to be used as a backbone regardless, and this resulted in some marked insufficiencies within the notional draft, which had to be corrected in the subsequent iterations.

6.1.1 Analysis

Based on the preliminary documents provided by Altice Labs, some information was already present to tackle the initial design of the architecture:

- A three layer setup - "Streaming" (Ingestion) Layer, Storage Layer and Serving Layer - was used.
- A number of technologies were present in the supplied diagrams, with the main ones being identified and categorized in the following sets:
  - **Ingestion** - *Kafka, StreamSets, Flume, Spark*.
  - **Storage** - *HDFS, S3, PostGres*.
  - **Serving** - *Impala, Presto, Flink*.

With this information, as well as the drivers and requirements extracted during the ADEW, it was possible to create a Requirement Specification, which informed on the system's functionalities and qualities, to a certain extent. The Requirement Specification had used an architecture partitioning system which relates to the logical structures within a Big Data architecture - *ingestion, storage* and *serving*. This would be the nomenclature that was carried forward, to maintain consistency between documents.

6.1.2 Creation

Using the supplied information as a supporting resource, an architecture draft was produced that used **Kafka** as its main *data streaming tool*, **HDFS** as its main *storage component*, and a mixed use of **Flume**, **Presto** and **Spark** for *querying and serving*. This layout was quite basic however, and did not meet all functional requirements (namely those related to task scheduling and automation, or ETL operations).

Following this analysis, and, after some additional research, the addition of a **Spark** module was added to *perform ETL* prior to data storage into the Hadoop cluster. As a supporting structure, and based on research regarding the use of **Airflow** as an automation support tool, the use of Airflow *with* Spark's ETL abilities was considered. Within HDFS, the addition of **Hudi** as a "Lakehouse" component, adding metadata management capabilities, and **Hive** as a "Warehouse" component adding ACID-compliant high-speed transactional queries allowed the draft to now be able to meet all the currently identified functional requirements (based on the listed features within the software's documentation).

Figure 6.1 expresses a simplified data flow diagram of the system's draft.

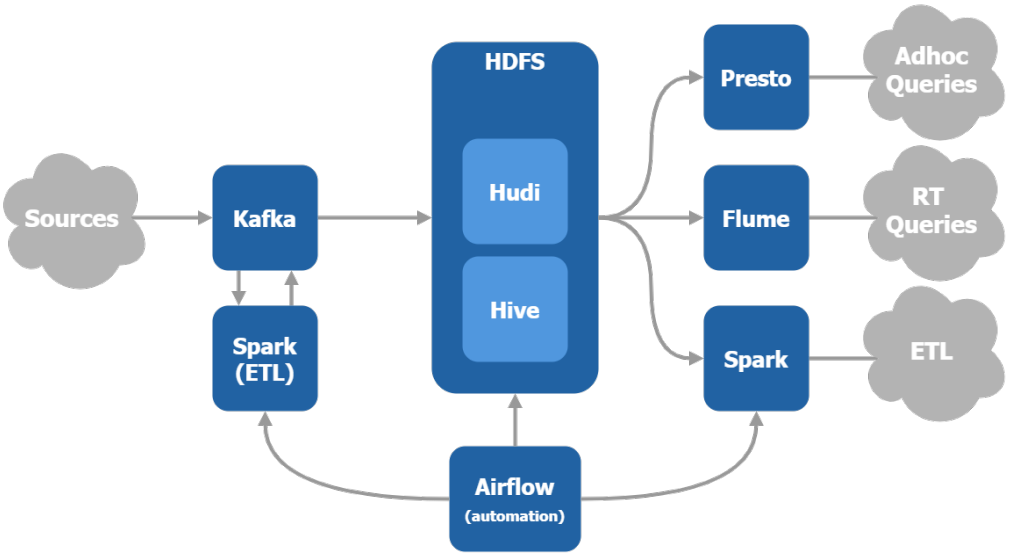

Figure 6.1: Conceptual data flow diagram of the notional architecture draft. The main components are visible and their data flows are represented by arrows.

6.1.3 Review & Validation

As previously mentioned, at this point the rigorous validation process of the architecture review (as defined in the Methodology chapter) was not yet being used, as the notional architecture required only a superficial documentation-based validation. In its stead, a simple and brief documentation-analysis based approach was used to perform the gist of the validation, looking at the requirements and comparing the various components' documentation to assert, albeit only on a surface level, if the requirement *could* potentially be met.

The process involved going through each of the layers, and verifying if the documentation presented sufficient evidence that the various components was provisioned with the required functionality/quality to meet the requirements. This description uses the requirements as they were defined at the start of the project, presenting various differences from the final version.

Ingestion Layer

The Ingestion Layer (IL) components met all the functional requirements, as expected. The documentation showed them as possessing all the necessary features which, up to that point, had been identified for the IL. Next, it was important to verify if the components were within possibility of meeting Quality Attribute (QA) requirements, by looking at benchmarks and documentation and comparing them to the metrics exposed within the QAs. Table 6.1 presents this "Pass-Fail" analysis:

ID	Description	Pass (w. desc)
QA-IL-001	System must be able to capture **600M** events/day in regular operation.	**PASS** - Kafka's limitation comes from lack of hardware. Some instances have been configured to process trillions of events per day.
QA-IL-002	The system *(handles)* 40k files per day under regular operation. *(Processed)* in under two minutes.	**PASS** - Ingestion time is one of the measurable and configurable settings for a Kafka cluster, allowing fine-tuning of specific streams for these files.
QA-IL-003	Each file has between 100MB and 200MB and will represent 50GB daily.	**PASS** - By scaling up resources, Kafka can instantiate multiple brokers to process up to 10GB/s (in ideal conditions).
QA-IL-004	Data acquisition must be continuously initiated at 6 minute intervals.	**PASS** - Through Airflow scheduling and automation, Kafka streams can be instantiated on-demand.
QA-IL-005	The system can handle near real-time data acquisition (...).	**PASS** - Kafka latency can be made as low as 10ms.
QA-IL-006	The system can handle 100k events per second from 10k Video Boxes.	**PASS** - Due to Kafka's scalable nature, it can easily handle 100k events, by scaling up the amount of internal clusters.
QA-IL-007	The system supports both continuous and periodic data acquisition processes.	**PASS** - Kafka streams are continuous, but through Airflow scheduling and automation, it can be made to be on-demand or periodic.
QA-IL-008	The platform shall support exactly-once (EO) message delivery.	**PASS** - Kafka can support EO message delivery using a transaction-based method of operation.

Table 6.1: QA Evaluation table for draft components of the Ingestion Layer.

Storage Layer

The Storage Layer (SL) components met all the functional requirements, having support for both streaming and batch data, possessing features such as data compression, compaction, etc. and having inter-operability features for data integration, due to running on the HDFS ecosystem, which provided native compatibility between all the selected components. The subsequent validation of the QAs produced results indicative of a good fit, showing that through analysis of feature-sets, HDFS, Hudi and Hive would possibly be able to meet all the relevant quality attributes. Table 6.2 presents the "Pass-Fail" analysis for the SL:

ID	Description	Pass (w. desc)
QA-SL-001	Aggregated data snapshots must be refreshed 24 times daily, in 1-hour intervals.	**PASS** - HDFS supports snapshotting natively, and through Airflow this process can be automated.
QA-SL-002	The system must cope with 20TB of data in 1st year and 10TB of yearly growth.	**PASS** - This growth is limited by hardware only, as HDFS is very flexible, even up to the petabyte scale.
QA-SL-003	Data must not be deleted in any circumstance, although lossless data compression (...).	**PASS** - Hadoop, Hudi and Hive all support lossless compression and backups.
QA-SL-004	The system must handle 10 to 20 thousand database tables under regular operation.	**PASS** - The Hive documentation recommends no more than 500.000 active tables, providing ample room for normal functioning and even expansion.
QA-SL-005	The system shall support Long-term storage (LTS).	**PASS** - HDFS, Hive and Hudi allow for backups and HDFS provides an advanced LTS feature natively.
QA-SL-006	The system shall support storage of structured, semi-structured and unstructured data.	**PASS** - Using Hive and Hudi both types of data are supported.
QA-SL-007	The system shall categorize incoming data automatically (...)	**PASS** - Airflow, running on a Spark analytics cluster, will allow for realtime data analysis and categorization, with optional injection of metadata into the files.

Table 6.2: QA Evaluation table for draft components of the Storage Layer.

Serving Layer

The SV's functional requirements were met through the use of Hive and Presto to satisfy querying operations, providing flexibility and options when configuring queries (ad-hoc and preset). The use of Airflow allows for the same flexibility when it comes to sending and receiving data as the other layers possess. Push/Pull operations can be configured, and data enrichment can be performed by automating data-source mixing tasks, and sending the data back to the IL.

The current state of affairs regarding the SV was quite insufficient, with very few Quality Attribute requirements defined, and with the use of components not being very clear. Regardless, the SV's components' ability to meet the identified quality attributes was evaluated, once again by researching within the documentation. Table 6.3 shows these results.

ID	Description	Pass (w. desc)
QA-SV-001	The system must enable auditing and non-repudiation of user's access to data.	**PASS** - Presto supports authentication and auditing natively, saving queries and usage metrics to a file (which may be stored safely in the SL).
QA-SV-002	The system ensures data privacy among clients *(in)* a multi-tenant architecture.	**PASS** - HDFS and all mounted components support role-based security and lockouts, enabling data privacy.

Table 6.3: QA Evaluation table for draft components of the Serving Layer.

General Requirements

Additionally, some of the functional and non-functional requirements that related to the globality of the system were analysed. Functional requirements were not defined for the globality of the system as each layer presented vastly different functional characterizations.

As far as the general qualities of the system, some questions of availability, maintainability and throughput were presented, and the analysis of the documentation, mainly of HDFS and Kafka, presented them both as potential candidates for meeting these requirements. Table 6.4 presents the results of this analysis:

ID	Description	Pass (w. desc)
QA-G-001	The system remains online during maintenance or update.	**PASS** - HDFS and possesses CI/CD capabilities.
QA-G-002	When a data system becomes unresponsive, data must never be lost and the connection is eventually re-established.	**PASS** - Kafka buffers support Exactly-once message delivery, using buffers to store data until it is sent successfully, and Airflow supports automating the retrying procedures/handshakes.
QA-G-003	Peaks (...) are typical, and the system remains in operation and does not lose data or functionality (...).	**PASS** - Kafka and HDFS are good pair since both are natively scalable and elastic. All other services mounted atop this ecosystem are, by virtue of integration, similarly capable.

Table 6.4: QA Evaluation table for globality of the system in the draft.

6.1.4 Outcome

While this iteration had a very troubled execution, namely due to project management related difficulties, the lack of adherence to strict validation standards and the general lack of experience and knowledge evidenced by the lackluster design process, it nonetheless resulted in a foundation which could be improved. This foundation is described in a condensed container diagram (Fig. 6.2).

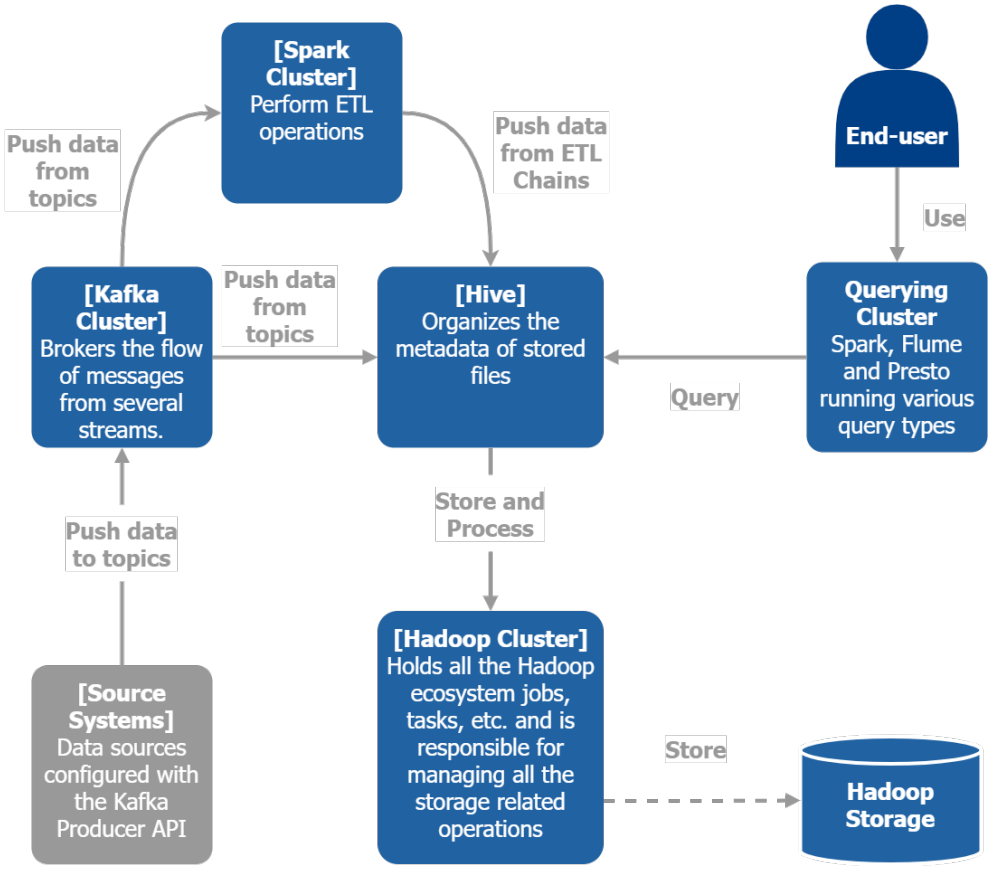

Figure 6.2: Condensed container diagram for the framework architecture as of the first iteration. Each block represents an application or micro-service executing in its own environment.

The draft had three main components which served as the core of the architecture: Kafka, HDFS and Spark/Presto. While these components were not yet experimentally validated, they served as a starting point for a more rigorous approach.

6.2 Iteration #1 - First Refinement, Ingestion Layer

The first iteration's objective was to build a more concrete architecture from the notional architecture produced in Iteration 0. The main problems identified with the draft included missing *management* and *monitoring* scopes, which had to be integrated and accounted for, and some important structural changes to the architecture (mainly the lack of a *Data Catalog* component).

Considering these changes, the areas where the requirements were validated and more refined were the target of focused refinement and experiments which sought to validate mainly performance and scalability-related metrics of the Ingestion Layer components - in this case, Apache Kafka. Alternatives to Kafka were mapped and compared, eventually cementing Kafka as the correct choice for this architecture.

The process of analysing, refining and validating these changes is described in detail in the following sub-sections.

6.2.1 Analysis

The analysis was performed internally, with the professors, and externally, through meetings with Altice Labs where the draft was presented, and some more requirement/driver related information was requested. Through this process, some areas were identified as being underdeveloped:

1. Two crucial scopes were missing from the requirement specification and architecture draft: **management of data access** (access control, data governance) and **management of the infrastructure** (health monitoring, network monitoring)

2. **Actors** were poorly defined in the requirements, with unclear roles such as "Administrator", "User" or roles which were too specific such as "Data Scientist" - that while not necessarily incorrect, didn't result in fundamentally distinct requirements/drivers.

3. **Metadata-related governance requirements** were poorly defined, and the architecture did not have a metadata catalog component, which was identified as a key requirement.

4. **Failure scopes** were not included in the Requirement Specification. Quality Attributes mostly accounted for normal operations and healthy system states and non-functional attributes were presented poorly, in a basic format with little detail (not quality-attribute scenarios)

These issues were tracked in the **RAID** sheet, and handling plans were put in place to begin the refinement and validation processes. For this iteration, most of the issues were related to lacking architectural drivers (requirements, essentially) which required re-working and clarification.

6.2.2 Refinement

The first stage in the refinement process was the update of the requirements according to the feedback obtained during the analysis stage. This process consisted in the re-writing of requirements with the following objectives:

- **Improve the quality of the requirement specification** – Make definitions clearer, avoid overly specific definitions and remove redundant requirements. Quality-Attributes converted to six-part scenarios; roles reworked (Data Manager, Process Manager and End-user).

- **Centralise infrastructure management requirements** – These will be encompassed in a new layer – the **Orchestration Layer (OL)**. Includes everything related to logging and health monitoring/alarms.

- **Centralise access control/privacy requirements** – These will be encompassed in a new layer – **the Administration Layer (AL)**. Includes everything related to access control and traceability.

With these new requirements, new technologies were identified as potential candidates for these two new layers and for the new data access stack:

Lakehouse - Data Access Stack

In order to make the Lakehouse a reality, it is necessary to ensure the existence of a metadata store, as well as a service to use this store.

Lakehouse interfaces which have currently been identified (**Hudi** and **Iceberg**) seem to meet the functional requirements for the system, but they have not been fully validated, and their feature-set is too extensive for a fully-encompassing testing scope.

Metadata Catalog

The currently most promising candidate for the Metadata Catalog on the on-prem conception of the system is **Apache Atlas**, which is an out-of-the-box data catalog and governance solution for HDFS-based systems. It is heavily tied to the Hadoop ecosystem, presenting native integration with *Hive, Kafka and Spark*.

There are also cloud-native alternatives which offload the metadata management and data access to managed services. **Cloudera**, for example, ships with Apache Atlas natively integrated; **Alation** is a managed data catalog which can hook onto services such as S3/Min.IO and **AWS Glue** is able to provide data catalog capabilities to AWS-based storage.

Orchestration Layer

In this layer, it was necessary to design a monitoring stack that could extract metrics from all the currently identified components, and generate opportunities for high-performance analytics. For the on-prem conception of the architecture, **Nagios** was identified as a possible centerpiece for this layer, to monitor connectivity and general system health through direct connections to the numerous servers (for example, the Kafka services, HDFS back-end, etc.) . Additionally, the use of **Log4j** as a technical bridge for the numerous Java-based components enables the use of **Spark** to run continuous real-time analytics on the large log flows generated during everyday operations.

Cloud-native solutions generally rely on integrated infrastructure management dashboards and configurations. As a result, if the architecture migrates to the cloud-native managed service route, then the management and monitoring is integrated, for example, with **AWS Cloudwatch** in the **AWS** stack.

Administration Layer

The administration of the system essentially relies on the underlying technology stack. For each of the alternatives (on-prem, cloud-native) several options are available. HDFS based systems can use **Apache Ranger** (which features native integration with **Apache Atlas'** dashboard) or **Apache Sentry**, which governs direct accesses to HDFS. Meanwhile, cloud-native solutions, much like the management scope, often come with their own identity management and governance solutions. **AWS IAM** is the component in the AWS Suite which can enforce role-based security on the entire ecosystem.

Consolidation

Additionally, work was performed to organize the current components, and their respective alternatives, layer-by-layer. It is possible to look at the architecture as being not just a single possible path, but a set of modular building blocks which can be used to reach the goal of having a functional framework. By analysing the compatibility between the identified technologies, it is possible to, for example, leverage the power and availability of cloud-native *tech* while also maintaining the fine-grain control of self-hosted and designed serving solutions.

On the *Ingestion Layer*, Kafka was defined as a strong candidate due to its simplicity, power and compatibility. Despite this, it was necessary to perform a validation pass to ascertain if it truly was the most adequate fit, before experiments could proceed.

The result indicates that, for an on-prem, open-source solution, Kafka holds the lead when compared to the alternative message systems - ActiveMQ, RabbitMQ and, on the managed side, Kinesis/Confluent Kafka (these losing priority mainly due to the costs associated with them). The results of this comparison are presented in Table 6.5.

Feature	Apache Kafka	Confluent Kafka	AWS Kinesis	AWS SQS	ActiveMQ	RabbitMQ
Scalability	Horizontal, auto	Horizontal, auto	Horizontal, auto	Horizontal, auto	Replicable	Not natively
Cost Control	Can throttle consumer	Yes	Yes	Yes	Throttled externally	Can throttle consumer
Throughput	Very High	Exceptional	Exceptional	Exceptional	Low	Low
At-Least Once	Yes	Yes	Yes	Yes	Yes	Yes
Message Retention	Policy-based	Policy-based	Policy-based	Policy-based	Delete after ack.	Delete after ack.
Cost Model	Open-source	Managed (SaaS)	Managed (SaaS)	Managed (SaaS)	Open-source	Open-source
Availability	Very High	Exceptional	Exceptional	Exceptional	Medium, High	Medium, High
Pluggable Metrics	Yes	Yes	Yes	Yes	Yes	Yes
Monitoring	Highly compatible	Proprietary	Proprietary	Proprietary	Highly compatible	Highly compatible
Message Ordering	Yes	Yes	Yes	No	No	No
Message Consumption	Pull-type	Pull-type	Pull-type	Push-type	Push-type	Push-type
Latency	Low	Low	Low	Low	Very Low	Very Low
Error Detection	Checksum	Checksum	Not natively	Optional Checksum	Not natively	Not natively
Topology	Pub/Sub	Pub/sub	Pub/Sub	Queue	Queue	Queue
Direct ETL Support	No	No	Yes	No	No	No

Table 6.5: Comparison of the main Data Ingestion components under analysis. Components are sorted from left (most suited) to right (least suited) based on how well they meet the requirements through a documentation-analysis based approach, to set expectations and create a knowledge-base.

While these features do not directly express quality attributes, they are inherently tied to them. Through these metrics it is possible to assert that quality-attributes *may be met*, and informs on which component may be best suited for the task, serving as a preliminary step to the validation processes which take place in the next stage.

To inform on the current state of the architecture, a simplified Container diagram was created, to illustrate the architecture's main operating blocks (Figure 6.3. For the C4-model presentations, the layers which are used in the design process (IL, SL, SV, AL, OL) are not utilised.

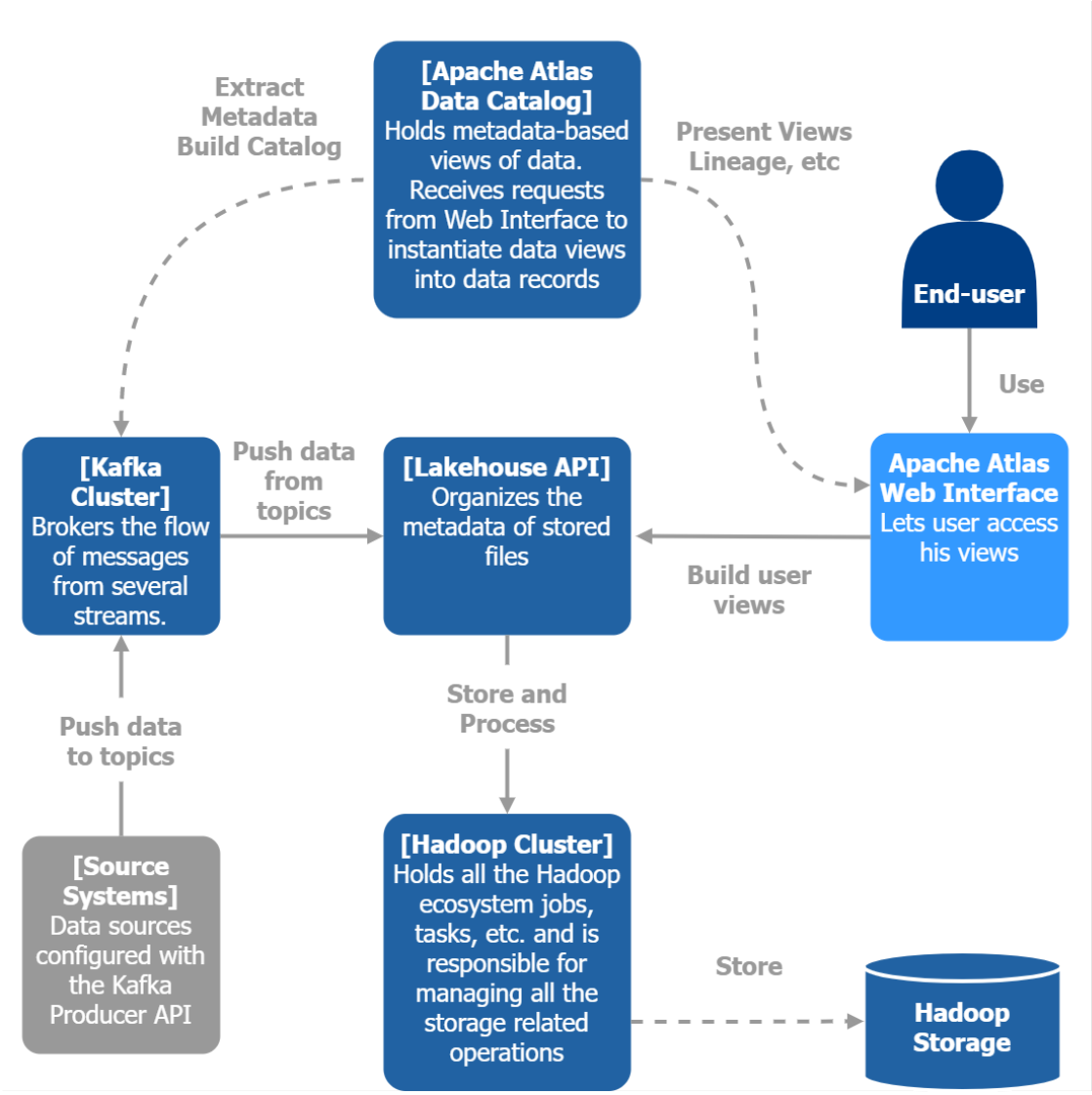

Figure 6.3: Condensed container diagram for the framework architecture as of the first iteration. Each block represents an application or micro-service. Dashed lines indicate metadata flows.

In the current definition of the architecture, data flows from the source systems through a Kafka cluster, entering the Hadoop Storage through the use of a Lakehouse API (which is yet undefined), and going through operations, transformations and integration within a Hadoop cluster (which may house a number of internal components such as Spark, Hive, etc.). Parallel to this, the Apache Atlas data catalog is being built with entries that originate through a hook onto the Kafka cluster, cataloguing data as it comes into the system, and eventually making it available through a Web Interface, to which the authenticated user will have access to.

6.2.3 Review and Validation

For the review, and considering that some large changes were still pending validation from Altice Labs (mainly the changes to the Serving Layer, Orchestration and Administration layers and all the supporting structures), the focus of the review was on the **Ingestion Layer** and the **Storage Layer**.

In the **Ingestion Layer, Kafka** presented some unique challenges:

- Performance requirements could not be verified by analysing research or documentation alone, experimental validation was required to verify if it could meet these requirements, even on sub-par hardware.
- Availability and Integrity related requirements raised some concerns since Kafka's only mechanisms to preserve harmonious operation stem from replication and redundancy – documentation states that *checksums* are applied but the process is not transparent. Could Kafka be relied upon to deliver all sent messages, and could it do so without errors?
- Kafka would be experimentally analysed to validate the following metrics, in order to verify if it meets all the relevant quality-attribute requirements:
  - End-to-End Latency (<100ms in 99.99% of messages)
  - Message Loss Rate (<0.1%)
  - Message Out-of-Order Rate (<0.01%)
  - Message Error Rate (<0.01%)
  - Throughput (informative measure only)
  - Maximum Recorded Downtime (Must be <30s)

As for the Storage Layer, Hadoop/HDFS also presented some difficulties, mainly due to how extensive the software is:

- Due to its large scope, HDFS was only understood on a basic level. The ecosystem presented so many technologies that a more comprehensive analysis was due to ensure that it was viable. Documentation, online experiments and the general industry's long-term support make it a seemingly viable candidate in all things related to scalability (and such requirements cannot be tested without sufficient hardware).
- It is then necessary to adapt the requirements of the Storage Layer to focus more on the usability of the storage components that exist within HDFS or any other storage solution, rather than attempt to analyse the performance and scalability characteristics of the storage solution itself, as testing would be impossible within the hardware's limitations.

The remaining aspects which were not validated were passed onto the next iteration, pending the feedback from Altice Labs.

The output of the validation of the first iteration was summarised through a condensed representation of the **RAID Issue Tracker** table (Type 0 issues - "no action required" - were omitted for brevity) presented in Table 6.6:

ID	Issue Description	Type	Mitigation Plan	Status
I-001	Monitoring is insufficiently defined.	4	Re-structure the architecture to account for a larger degree of monitoring and traceability.	*NEXT*
I-002	Catalog not included in the architecture draft.	3	Research Data Catalogs, rework the Storage Layer and Serving Layer.	*NEXT*
I-003	Alarms/actions are not defined.	4	Research monitoring/orchestration solutions/engines	*NEXT*
I-004	No provisions for data lineage tracking	4	Related to I-002, research Data Catalogs/Lakehouse.	*NEXT*
I-005	No provisions for access control, views, etc.	4	Research traceability/access control solutions compatible with the new catalog/data solution.	*NEXT*
I-007	Kafka's execution profile is unclear.	5	Prepare experiments: INDIV-001 and INDIV-002.	OPEN
I-009	Connection of a new data stream may take more time than expected.	5	Research shows that new sources take a bit more to start (20-30 seconds). Is this a problem? Would this condition the use of Kafka?	OPEN
I-010	Kafka's error detection abilities are unclear	5	Experiment on Kafka error rates (INDIV-002).	OPEN
I-011	Kafka latency may be too high for RT/NRT.	5	Experiment on Kafka latency (INDIV-001, INDIV-002).	OPEN
I-012	HDFS is very crudely understood.	3	Research more on HDFS fault tolerance, data replication, availability metrics.	CLOSED
I-017	Requirements scoped under failure scenarios are underrepresented	2	Rework QAS descriptions to also include failure modes and possible faults.	CLOSED

Table 6.6: Condensed view of the RAID issue tracker table as of Iteration 1, displaying each issue, its description, type and mitigation plan. Issues of type 0 were omitted, and text was condensed to fit the short-form presentation approach. The "Status" column indicates that state of the issue at the end of the iteration. Issues with closed status will be omitted from future iterations of the table.

6.2.4 Experiments

In this iterations two experiments were performed on Apache Kafka, using its default configuration, to analyse its performance in normal operations, highly concurrent (high-load) operation under partial failure. The experiments identified in Table 6.7 were executed using the configuration described in Table 6.8.

ID	Target	Description	Metrics	Status
INDIV-001	Kafka	Profile Kafka's normal operation	Performance, Integrity	PASS
INDIV-002	Kafka	Profile Kafka under partial failure	Availability, Integrity	PASS

Table 6.7: Experiment specification related to Iteration 1's validation process.

ID	System	CPU	Clock	Cores	Memory	OS
A	ASUS X571GT 79B15PL1	Intel Core i7-9750H	4.5GHz	6	12GB	Windows 10 22H2

Table 6.8: Technical specification of the system for the experimental setups described for the first iteration.

The software was deployed locally in System A through a set of Java-based applications using a threaded implementation of the Kafka Consumer and Producer API (version 3.4.0).

Configuration

The Kafka configuration which was used in this set of experiments consisted in a simple three-broker implementation, serving a single Kafka topic, with a varying number of producers (single or concurrent), as described in Figure 6.4:

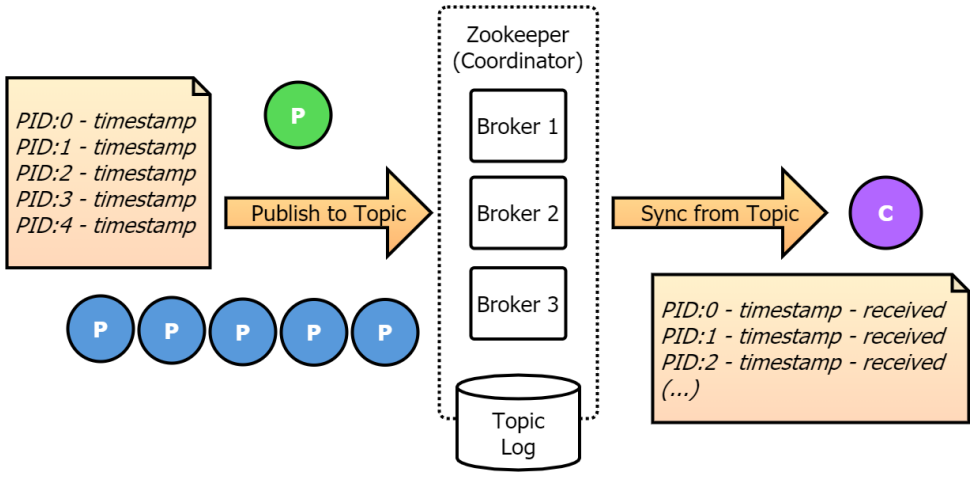

Figure 6.4: Experimental setup for the Kafka experiments of Iteration 1. In green is the "single producer" setup, and in blue is the "concurrent producer" setup.

The message structure (described in Figure 6.4) included the Process ID (PID) of the producer responsible for writing the message to the topic; a serial/sequence number for the message and the timestamp of the commitment of the message to the topic.

This structure will enable the computation of metrics related to integrity, performance, consistency and availability. Upon arrival, the messages will be sorted by their PID to assert ordering, errors and loss/failure. The following metrics will be analysed:

- **End-to-End Latency** - Mean of end-to-end latency. 99th percentile will also be analysed.
- **Message Loss Rate** - Number of undelivered messages (expressed as a percentage).
- **Message Out-of-Order Rate** - Number of inconsistencies in message order (expressed as a percentage).
- **Message Error Rate** - Number of inconsistencies in message content (expressed as a percentage).
- **Throughput** - Computed number of messages sent per hour.
- **Maximum Recorded Downtime** - Time elapsed since system failure (broker outage) until service re-start.

The following internal identifiers will be used in this experiment-set to categorize the different scenarios:

- **NC** - Normal Conditions
- **PF** - Partial Failure
- **SPr** - Single Producer
- **CPr** - Concurrent Producers

And the following conditions are defined for the experimental scenarios that were used in the experiments:

- **NP SPr/CPr** (Normal Conditions, Single and Concurrent Producers) - 100.000 events (processed by consumer), brokers remain active throughout. Mean results calculated over 10 runs.
- **PF SPr/CPr** (Partial Failure, Single and Concurrent Producers) - 100.000 events (processed by consumer), single broker failure induced programatically at 30.000 messages. Mean results calculated over 10 runs.

INDIV-001 - Kafka Metrics - NC SPr/CPr

Trait	NC SPr	NC CPr
Mean end-to-end latency (ms)	1.09ms	2.45ms
Message Loss Rate (LR)	0%	0%
Message Out-of-Order Rate (OOR)	0%	0%
Message Error Rate (ERR)	0%	0%
Throughput (TP)	3.5 million msg/hour	5 million msg/hour
Maximum Recorded Downtime (MD)	-	-

Table 6.9: Experimental results for experiment INDIV001 showing Kafka under normal conditions in the two analysed load scenarios.

Results - Kafka shows excellent performance. Performance, Consistency and Integrity requirements were met.

Observations - The message transmission occurred without issues, and the execution profile seems to indicate that Kafka possesses extraordinary throughput potential, with a very high level of performance, namely considering the near-realtime latency results. Kafka's internally managed scaling profile is also evident, as when the concurrent producer setup began flooding the topic with messages, Kafka scaled up its resources and increased its throughput to match the increased load.

INDIV-002 - Kafka Metrics - PF SPr/CPr

Trait	PF SPr	PF CPr
Mean end-to-end latency (ms)	54.27ms	92.53ms
Message Loss Rate (LR)	0%	0%
Message Out-of-Order Rate (OOR)	0.08%	0.29%
Message Error Rate (ERR)	0%	0%
Throughput (TP)	1.02 million msg/hour	2.32 million msg/hour
Maximum Recorded Downtime (MD)	22s	19.4s

Table 6.10: Experimental results for experiment INDIV002 showing Kafka under partial failure (broker outage) in the two analysed load scenarios.

Result - Kafka maintains an acceptable level of performance, meeting the specified quality attribute metrics even under partial failure.

Observations - Kafka's latency and maximum downtime were affected by broker failure, as it incurred a relatively fixed downtime (configuration file defines 18 seconds). There were messages ordering faults during the load re-balancing, as some of the messages which were passed by the lost broker were re-transmitted.

Experiment Results and Conclusions

Kafka performed to expectation, meeting all latency requirements, loss rate and error rate requirements and even downtime requirements without much difficulty.

However, some questions were noted that, despite not making Kafka ineligible for its place in the architecture, deserve some type of consideration:

- **Out-of-order requirement is not met during partial failure** - It may be important to implement an ordering and de-duplication mechanism or explore Kafka's options for broker-consumer synchronization.
- **Scalability is observed but not linear** - Kafka's scaling profile, for a five-fold increase in load, was only a 42% increase in throughput. It may be that this was a factor limited by the under-powered hardware (as Kafka typically runs on machines with a lot more parallel computing power and memory), but it nonetheless requires a note that there may be some merit in performing a full analysis of the scaling profile for Kafka's processing under high loads, in the same vein as some Kafka studies which create progressive load profiles [58].

Due to the limitations of the project in both scope and time, the experimental validation of Kafka for this iteration is limited to these experiments. However, these considerations are carried forward in the issue tracker, where an issue will reside regarding Kafka's scalability profile and regarding the need of a robust ordering mechanism (either correct or tolerate).

6.2.5 Outcome

With the closing of the first iteration, the results of the undertaken processes were summarised (layer-by-layer):

- **Ingestion Layer**
  - Technologies identified (Kafka, RabbitMQ, ActiveMQ, Kinesis, SQS) and features mapped and compared.
  - Selection of Kafka as the main ingestion component. Other components present some challenges (lack of scalability, complexity, costs) and are not prioritized.
- **Storage Layer**
  - Technologies identified for Storage back-end (HDFS, S3/Min.IO) and analysed.
  - Current selection is HDFS for the storage back-end. Revision by Altice Labs is required.
- **Serving Layer**
  - Data Catalog component is now under analysis. Technologies are currently being identified.
  - The role of the metadata catalog and the Lakehouse components are not fully clear and should be researched and better understood going into the next iteration.
- **Orchestration and Administration Layers**
  - Monitoring and Management are now scoped, and components have been selected, albeit in a tentative way.
  - Dedicated requirements and components have been created, and technologies have been identified, as well as proprietary/cloud-native alternatives. Despite this, more research is due, as the area is relatively recent and documentation is scarce for a number of use-cases.
- **Requirement Specification**
  - Requirements were much more developed and concrete, and now include failure and maintenance scopes.
  - Feedback indicates that some level of stability had been reached with regards to the requirements. Some details left to clarify, but major decisions were now easier to make.

6.3 Iteration #2 - Storage and Serving Layers

The second iteration involved a great deal of change, mainly due to a number of requirement changes that occurred after Iteration 1. This iteration focused on adapting the previous version to these new requirements, as well as continuing the active exploration of new solutions in the realm of data governance and management.

After the validation of the Ingestion Layer, there was an opportunity to explore the drivers of the Storage Layer and, due to its proximity and dependency, the Lakehouse component of the Serving Layer, due to the aforementioned requirement changes. This iteration also focused on validating the compatibility between all the selected components thus far.

6.3.1 Analysis

Following the presentation of Iteration #1's outcome, and the presentation of the iteration's developments to Altice Labs, some new information was introduced to clarify questions regarding the Ingestion Layer's experimental stages, resulting in changes to requirements.

Additionally, some information was forwarded regarding the Lakehouse component, which, up until this point, had been undefined as the relevant technologies were still under research; and some important requirement changes were communicated regarding the infrastructure management components, namely that the focus of the validation should switch to a compatibility study rather than a full experimental analysis.

These external changes can be described, layer-by-layer:

- **Ingestion Layer**
  - Some newly defined requirements. **RT/NRT streaming** are possible but not essential; **audio and video ingestion** are core features for the ingestion pipeline.
  - Alternatives to **Kafka** must be analysed further to see if there is any alternative powerful enough for high-throughput scalable ingestion, however, Kafka has massive benefits as a choice – cost of entry is essentially null since Altice Labs S.A. is already using it at scale and has evidence of adequate performance, scalability, availability and workflow-fit.
- **Storage Layer**
  - The primary choice for the **Data Lake** component at Altice Labs is shifting away from **HDFS**, with a transition towards object-storage based solutions such as **S3/Min.IO**. This transition should be prioritized. However, HDFS can still be considered as a secondary option to be analyzed and included if necessary.

- **Orchestration Layer**
  - This layer has been de-prioritized in favor of focusing on the functional side of the framework, mostly due to the existing knowledge of the team regarding management and monitoring technology, in order prioritize the development of core features - namely the *Data Lakehouse* and *Catalog* solutions.
- **Serving Layer**
  - The use of PrestoDB, Spark and Flink for data processing/serving is highly recommended due to familiarity within Altice Labs and will be used as a reference for comparisons and pre-validation screening of compatibility.

In parallel, internal research was conducted to identify new technologies for the **Metadata Catalog**. The previous conception of the data catalog - a data access interface - was incorrect, as this feature will be covered by the Lakehouse component.

The role of the metadata catalog is much more abstract and independent from the data access stack, without inherently providing any means to access the data to which it refers to, instead relying on the analysis of the metadata surrounding the system's many data flows and entries from a purely governance and management-oriented perspective. In summary, the internal changes can be described as:

- **Serving Layer**
  - **Data Catalog** no longer in the *Serving Layer*, moved to *Administration Layer* due to re-structuring of requirements as well as new research and clarifications.
  - *Serving Layer* now encompasses the **Lakehouse** component - a metadata-based system for ACID-compliant, reliable and fast data access on mixed distributed storage data in variable formats - as well as the querying logic and any related components.
- **Administration Layer**
  - Now houses the **Metadata Catalog**, which tracks the entire data lifecycle, providing lineage views, traceability and auditability through visualisation-based dashboards.
  - Previous iteration presented Atlas as a solution, but its native connectors are no longer the best fit considering the shift away from HDFS and toward S3/Min.IO.
  - New technologies identified as alternatives to Apache Atlas, perhaps covering some of the identified problems – **OpenMetadata**, **DataHub**.

And, in an effort to improve the understanding of the Metadata Catalog's function in a data management framework's compliance goals, a meeting was scheduled with another member of the POWER Project, specializing in Data Governance, to obtain some insight into the requirements associated with a metadata-based governance system.

- **Data provenance** and **origin tracking** are top priority.
- The use of a **business glossary** can improve inter-operability between domains.
- **Metrics and dashboard analytics** for metadata flows is also a desirable feature.
- **DOI (Digital-Object Identifiers)** for metadata is a requirement.
- **Filter systems** and **search-engine optimizations** are also desirable.

6.3.2 Refinement

After the analysis, steps were taken to improve the architecture. Following the same pattern that was previously defined, the requirements were updated followed by a revision of the architecture's layout and components. Following these changes, the architecture was consolidated and all the current alternatives, design considerations, etc. were mapped.

The requirement specification was updated in the following areas:

- **Clarify the Lakehouse components/Metadata Catalog components** - Putting them in their appropriate layers (Lakehouse as a Serving Layer component, Catalog as an Admin. Layer component) and clarifying their role in the architecture.
- **Rework the requirements of the Orchestration Layer** - Reflect the shift toward compatibility-based validation rather than direct performance/feature validation.
- **Rework the requirements of the Administration Layer** - This layer, which now includes the Metadata Catalog, should incorporate the feedback from Altice Labs and from the internal research and meeting with the data governance project's responsible.

And, following the implementation of these changes, the architecture itself required some re-structuring, as well as the addition of some new technologies (namely the previously unresolved Lakehouse component). These new technologies focused mainly on the Serving layer, in both the Lakehouse scope and in the context of the query engines that will support the functional use of the platform.

Lakehouse

The Lakehouse components which were aggregated and studied were the previously mentioned **Hudi** and **Iceberg**, as well as the **Delta Lake** solution, which had been absent in the previous iteration. These three technologies were looked at and analysed comparatively to assess their compatibility with the Kafka/Min.IO stack. This would serve as a preliminary selection prior to the validation based on the quality-attributes and requirements.

In this evaluation, Hudi was selected as the most fit for the current layout of the system because it met all the functional requirements and seemed fit (based on documentation analysis and *benchmarking*) to meet all the quality attribute requirements.

Feature	Hudi	Iceberg	Delta Lake	AWS LakeFormation
ACID Compliant	Yes	Yes	Yes	Yes
Automated Data cleansing	Yes	Yes	Yes	Yes
Time-travel queries/snapshots	Yes	Yes	Yes	Yes
Data retention policies	Yes	Yes	Yes	Yes
Data lineage	Yes	Possible		Yes
Audit History	Yes	Possible	Yes	Yes
Schema Evolution/Enforcement	Yes	Yes	Yes	Yes (schema crawler)
Open Source	Open Source	Open Source	Open Source	Closed Source
Preffered use-case	Efficient Cloud-Native "Kappa" analytics	Large table .Lakehouses	Massive data pipelines	AWS S3 based Lakehouses
OOtB Compatibility with Kafka	Yes, very complete	Not yet, community driven	Yes, very complete	Yes, although not natively
OOtB compatibility with S3/MinIO	Yes, very complete	Yes, very complete	Yes, community driven	Yes, S3 "one-click"
OOtB Compatibility with Presto	Yes, very complete	Yes, reasonably complete	Yes, very complete	Yes, very complete
OOtB Compatibility with Spark	Yes, very complete	Yes, very complete	Yes, very complete	Yes, preferably through Amazon EMR.
OOtB Compatibility with Flink	Yes, very complete	Yes, very complete	Yes, reasonably complete	Yes, preferably through Amazon EMR.

Table 6.11: Comparison of the main Lakehouse components under analysis. Components are sorted from left (most suited) to right (least suited) based on how well they meet the requirements through a documentation-analysis based approach.

Query Engines

The refinement of the query engine solutions was a necessary step given the amount of change the Storage and Serving layers went through in this iteration. Considering the choice of Hudi as the component of choice, it was now interesting to see what native integrations are supported by Hudi, and select those that are a better fit for the target architecture, prior to a more in-depth validation.

Based on the Apache Hudi documentation [26], there is a large number of compatible query engines:

- **Presto and *Trino*** - For interactive querying and analytics.
- **Hive, Spark and *Impala*** - For batch analytics and data ops.
- **Flink** - For streaming analytics.

This makes Hudi a very promising solution, especially due to the native **PrestoDB, Spark and Flink** compatibility.

Metadata Catalog

Another area that needed changes was the Administration Layer's data catalog. With Hudi being selected for use within the architecture, it was possible to look among the alternatives and find which one was best suited for the task of overlooking the operation of this Lakehouse.

The three technologies under analysis for the refinement were **OpenMetadata**, **Acryl DataHub**, **Apache Atlas** and **AWS Glue**. The comparative analysis (initial approach) is described in Table 6.12, with Acryl DataHub being the component which provides the more promising set of features, especially the out-of-the-box (OOtB) compatibility with the currently identified stack (Kafka and S3/Min.IO).

Feature	Acryl Datahub	OpenMetadata Catalog	AWS Glue Data Catalog	Apache Atlas
Data Discovery	Yes	Yes	Yes	Yes
Metadata Management	Yes	Yes	Yes	Yes
Data Lineage/Provenance	Yes	Yes	Yes	Yes
Glossary	Yes	Yes	Yes	Yes
Data Quality pipeline	Yes	Yes	Yes	Yes, basic
Documentation	Good	Very good	Very good	Good
Open Source	Open-source	Open-source	Closed-source	Open-source
SaaS available	Yes	Yes	Yes, exclusively	Yes
Integration Method	Programmatic	UI-Based	UI-Based	Programmatic
CLI	Yes	Yes	Yes, as of AWS CLI 2	No, through python
Docker/Kubernetes friendly	Yes	Yes	No	No, Hadoop/YARN
Web UI	Yes	Yes	Yes, AWS Dash	Yes
Access Control	Yes	Yes	Yes	Yes
User Groups	Yes	Yes	Yes	Yes
Unique Data Object ID	Yes	Yes	Yes	Yes
Dashboard with Metrics	Yes	Yes	Yes	Yes
OOtB Compatibility with Kafka	Yes	Yes	Yes	Yes
OOtB Compatibility with S3/MinIO	Yes	Yes	Yes	Yes
OOtB Compatibility with Hudi	Yes	No	No	No
OOtB Compatibility with Presto	Yes	No	No	No

Table 6.12: Comparison of the main Metadata Catalog components under analysis. Components are sorted from left (most suited) to right (least suited) based on how well they meet the requirements through a documentation-analysis based approach.

Consolidation

As with the previous iteration, a consolidation took place to summarise the current architecture through a simplified container diagram (Figure 6.5).

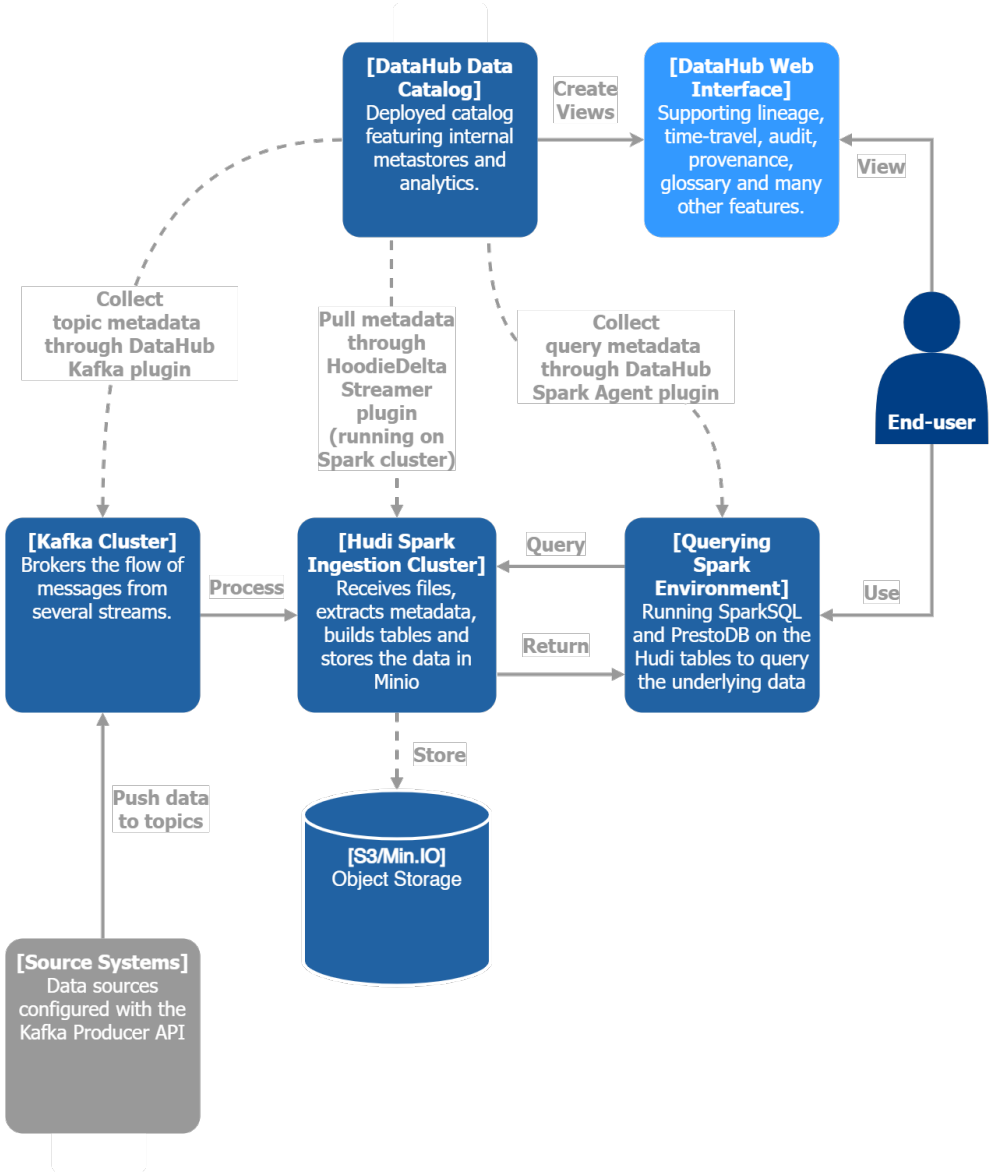

Figure 6.5: Condensed container diagram for the architecture as of the second iteration. Each block represents an application or micro-service. Dashed lines indicate metadata flows.

In this iteration, the architecture retains its usage of **Kafka** as the main ingestion system, but now uses a **Spark Cluster** to perform the storage of data, as it is performed **through the Hudi Lakehouse system**, which relies on Spark jobs to extract and label metadata in Hudi's table.

Typically, one Spark application (deployed in the cluster) will serve a Kafka topic and encompass that entire data flow, enabling standardized categorization through Hudi's metadata table.

This system inserts data into S3/Min.IO using the S3 API while simultaneously building the **Hudi metadata table** which enables **ACID-compliant and high-speed querying** on the underlying data stored the object storage. To query this data, a **dedicated Spark environment** is used, hosting SparkSQL (for Hudi-specific query types) and PrestoDB (for high-performance distributed querying) deployed on the Spark back-end.

While data moves through the platform, in all three major steps (ingestion in Kafka, storage through Hudi, and queries through Spark), metadata is extracted and passed to DataHub, which builds a dashboard (web interface) that allows for the users to observe data lineage analysis, data provenance checking, auditing and more through global perspectives (on the totality of data in the system) or a more focused analysis (targeting either the ingestion, storage or serving streams in particular).

6.3.3 Review and Validation

With the previous iteration focusing on the data ingestion processes, and the drastic changes which took place during the start of the iteration, this review focused on the **Storage** and **Serving Layers**, and how the Lakehouse paradigm can be implemented within the framework using current technologies. It was in this stage that HDFS was de-prioritized and S3/Min.IO was moved to the forefront of the Storage back-end, which arose some concerns regarding the compatibility of the chosen components.

For the Storage and Serving Layers:

- Min.IO compatibility must be verified for the currently selected components (namely on the data ingestion side)
- Lakehouse Components were analysed and Hudi was selected. Can we verify that Hudi works with the currently selected components?
- Serving Components - The currently selected components (Spark and Presto) should be analysed to verify compatibility with Hudi and Min.IO. Presto is natively integrated with both, and presents a lot of documentation in its use

Additionally, aspects of the **Administration Layer** were also tackled by analysing the metadata catalog components which were selected. For the Administration Layer:

- Acryl Datahub seems to be the most featured and appropriate choice for the desired use-case. However, it is a very recent software solution, and the current documentation is insufficient to ensure compatibility with the current components. Some tests are in order.

As previously discussed, the issues pertaining to the Orchestration Layer have been de-prioritized and reformulated to be essentially framed as compatibility

verification, and are carried over to a possible third iteration. Table 6.13 holds the condensed RAID issue tracker for the iteration.

ID	Issue Description	Type	Mitigation Plan	Status
I-018	Monitoring has been de-prioritized.	4	Rework requirements for compatibility-checks. Pass issues to next iteration.	*CLOSED*
I-019	Monitoring components need to be feature-mapped and compared.	3	Research their features and create a table.	*NEXT*
I-021	HDFS has been de-prioritized.	4	Rework architecture, focus on Min.IO/S3.	*CLOSED*
I-022	Apache Atlas incompatible with Min.IO/S3.	4	Research other data catalog solutions.	*CLOSED*
I-023	Lakehouse Component alternatives must be studied.	2	Document them and analyse their feature set to find the best fit.	*CLOSED*
I-024	Kafka->Hudi->Min.IO chain must be experimentally verified for compatibility	5	Attempt to connect them all and run a data ingestion process.	*OPEN*
I-025	Spark/Presto serving processes (queries) must be verified on Hudi	5	Attempt to query Hudi/Min.IO/S3.	*OPEN*
I-026	DataHub's interface and setup are not well understood	5	Attempt to set-up DataHub and run it.	*OPEN*
I-027	DataHub's ingestion for Kafka sources seems difficult to set-up	5	Run DataHub on a Kafka Stream.	*OPEN*
I-028	Can the entire system execute in harmony?	5	Run all components and monitor the data through the entire system.	*OPEN*

Table 6.13: Condensed view of the RAID issue tracker table as of Iteration 2, displaying each issue, its description, type and mitigation plan. Issues of type 0 were omitted. The "Status" column indicates that state of the issue at the end of the iteration.

6.3.4 Experiments

The experiments of this iteration focus on establishing that all the selected systems thus far are compatible, inter-operable, and that an end-to-end data streaming scenario can be established. Additionally, it was relevant to attempt to connect DataHub to the framework and witness its operation.

So, in this case, five experiments were planned. However, only three were performed fully, as the fourth experiment was unfinished, and the fifth experiment needed more system resources than those that were available at the time:

ID	Target	Description	Metrics	Status
INTER-001	Kafka, Min.IO	Place data in Min.IO through Kafka	Compatibility, Interoperability, Usability	PASS
INTER-002	Hudi, Min.IO, Spark	Place data in Min.IO through Hudi	Compatibility, Interoperability	PASS
INTER-003	Kafka, Spark, Hudi	Place data in Min.IO through Hudi via Kafka	Compatibility, Interoperability	PASS
INTER-004	DataHub, Kafka	Setup DataHub, gather from Kafka	Compatibility, Interoperability	SKIP
SYSTEM-001	Full System	Run and observe fully connected system behaviour	Compatibility, Interoperability	SKIP

Table 6.14: Experiment specification related to Iteration 2's validation process.

ID	System	CPU	Clock	Cores	Memory	OS
A	ASUS X571GT 79B15PL1	Intel Core i7-9750H	4.5GHz	6	12GB	Windows 10 22H2
B	Subsystem for Linux	Intel Core i7-9750H	4.5GHz	6	4GB	Docker Engine Linux
C	Virtual Machine	Unknown	Unknown	8	32GB	Ubuntu 20.04

Table 6.15: Technical specification of the systems used in the experimental setups described for the second iteration.

This setup required the use of **Docker** [56], a program that leverages OS virtualisation[1] to create containers - software packages that execute in isolation, running on the **Windows Subsystem for Linux** (system B), a virtual machine that runs natively on Windows. Following the delay of the dissertation, a **Virtual Machine** was reserved for testing with the Department of Informatics Engineering to account for hardware limitations of systems A and B - which unfortunately did not successfully resolve the identified issues.

[1]The creation of different parallel execution instances at the OS level, simulating an entirely separate computer within a computer's execution state.

Configuration

For this set of experiments, the goal was to create a compatibility verification process which enabled the flow of data from the ingestion components all the way to the serving components. To ensure this, the following applications were necessary:

- A working instance of **Min.IO** - *version 8.4.3*
- A working instance of **Kafka** - *version 3.4.0* - using the following libraries:
  - *apache.hudi.utilities_2.11* - For the Hudi connector
  - *apache.spark.sql_2.13* - For the Spark connector and session tools
  - *apache.hadoop.aws 3.3* - For the connection between Spark and Min.IO
  - *amazonaws.aws.java.sdk 1.12* - For the connection to the S3 API
- A working **Spark Cluster** -*version 3.3.2*
  - Using Hadoop "winutils" version 3.3 for compatibility with Windows

The Kafka instance from the first iteration was re-used, once more with the default settings. Min.IO was installed locally, and also used the default settings.

For the Spark Cluster, a Docker container setup was launched, which created the back-end for the Spark Shell to execute, and enabled the registration of Spark jobs for Hudi Ingestion, SparkSQL queries and PrestoDB queries.

INTER-001 - Min.IO/Kafka Test

For this experiment, the Kafka consumer was programmed to place its data within the Min.IO "kafka-test" bucket. The same configurations from the experiments in Iteration 1 were used, with a single producer under normal conditions sending 100.000 messages through the cluster to the consumer.

Result - Compatibility, inter-operability and usability requirements met.

Observations - All files were placed in the correct bucket, and were readily accessible. Because files were saved directly as text files, they were open-able, and presented the correct data within when viewed externally.

INTER-002 - Min.IO/Hudi/Spark Test

For this experiment, the Spark Shell was used to insert sample data into Min.IO through Hudi. Spark ships with some sample datasets which can be readily inserted in Min.IO. This experiment used an official Hudi guide [21] to provide information regarding on which commands to run.

A session was created, using Hudi as a metadata management system and Min.IO as the storage back-end, using the following configuration:

```
spark-shell
    --packages org.apache.hudi:hudi-spark3.3-bundle_2.12:0.13.1
    --conf 'spark.serializer=org.apache.spark.serializer.KryoSerializer'
    --conf 'spark.sql.catalog.spark_catalog= \\
                org.apache.spark.sql.hudi.catalog.HoodieCatalog'
    --conf 'spark.sql.extensions= \\
                org.apache.spark.sql.hudi.HoodieSparkSessionExtension'
    --conf 'spark.hadoop.fs.s3a.access.key=admin'
    --conf 'spark.hadoop.fs.s3a.secret.key=password'
    --conf 'spark.hadoop.fs.s3a.endpoint=http://127.0.0.1:9000'
    --conf 'spark.hadoop.fs.s3a.aws.credentials.provider= \\
                org.apache.hadoop.fs.s3a.SimpleAWSCredentialsProvider'
```

In essence, this session initiates a spark-shell job with Hudi as the catalog system, which uses SparkSQL to create the metadata entries before forwarding the data to the S3 API used by Min.IO with the provided credentials at the defined endpoint (in this case, exposed at 127.0.0.1, port 9000).

By submitting data through SparkSQL in this session, Hudi incrementally builds a metadata store within the S3 bucket. Data is submitted through the following commands:

```
val tableName = "hudi_spark_test"
val basePath = "s3a://hudi/hudi_spark_test"
val dataGen = new DataGenerator
val inserts = convertToStringList(dataGen.generateInserts(10))
val df = spark.read.json(spark.sparkContext.parallelize(inserts, 2))
df.write.format("hudi").mode(Overwrite).save(basePath)
```

These commands generate a dataset through the Spark "DataGenerator" module, converting it into a serializable data format (dataframe), setting the *dataframe* format to a Hudi compatible schema, and then writing it to the Hudi path. With the data inside, it was now time to attempt a query through SparkSQL.

```
val tripsSnapshotDF = spark.read.format("hudi").load(basePath)
spark.sql("select   _hoodie_commit_time, _hoodie_record_key, \\
                    _hoodie_partition_path, \\
                    rider, driver, fare \\
                    from  hudi_trips_snapshot").show()
```

This query produced a response that featured a table containing the sample data (*rider, driver and fare columns*) as well as relevant Hudi metadata (*commit time, record key, partition path*). Time-travel queries are also available (track data as of a certain commit time)

Result - Pass, Hudi and Min.IO are compatible and the SparkSQL query system implemented by Hudi meets all requirements.

Observations - Following this, the new bucket "hudi_spark_test" was available, and the data was visible within the bucket. SparkSQL queries performed in an independent Spark session produced the correct and expected return.

INTER-003 - Kafka/Spark/Hudi Ingestion Test

The first step is to initiate a Spark session with the correct packages attached, namely the SparkSQL-Kafka package.

```
spark-shell
    -- packages org.apache.spark:spark-sql-kafka-3-4_0\\
  org.apache.hudi:hudi-spark3.3-bundle_2.12:0.13.1
    (... same configuration as before)
```

At this stage, before any Spark connectors are deployed, Kafka is launched, with the topic "hudi_kafka_test_topic" being the destination for the messages to be sent by the producer. No consumers are required, as the Spark job for writing data to Hudi from Kafka encapsulates the consumer logic. The next step is to create a stream reading component within Spark that can be attached to a Kafka topic through connection with its bootstrap servers (in this case, exposed at 127.0.0.1, port 9092). The reader is configured to start at the "earliest" submission *which it does not yet know* - this will ensure that the data streams encompass the entire topic while avoiding duplicates.

```
val dataStreamReader = spark.readStream.format("kafka").
option("kafka.bootstrap.servers", "http://127.0.0.1:9092").
option("subscribe", "hudi_kafka_test_topic").
option("startingOffsets", "earliest")
```

Next, a Hudi writer was created, that builds data-frames based on the data which it receives from Kafka. This is the job which will be executed alongside the Kafka program, triggered to run every 30 seconds (30000 milliseconds) and ingest data from the Kafka topic, writing it to Hudi's table. The data is written in Min.IO through Kafka itself, and not through this Spark session.

```
val writer = df.writeStream.format("org.apache.hudi")
val tableName = "hudi_kafka_test"
val basePath = "s3a://hudi/hudi_kafka_test"
writer.trigger(new ProcessingTime(30000)).start(basePath);
```

Now, with the writer in execution, triggering ingestion into Hudi every 30 seconds, it is possible to launch the producer, which begins sending its programmed 100.000 messages. The process takes approximately 4 minutes, and data is streamed in in 30 second intervals.

Result - Passed, compatibility is assured, Kafka can stream data into Min.IO using Hudi with the help of a Spark job.

Observations - Data is correctly streamed in with the appropriate time-stamps. Future experiments could be targeted at the "writer" job execution timer, to see if it incurs in any significant overhead if a real-time (<10ms) use-case was considered. It is worth noting that a much simpler option is currently under development using Kafka Connect [106] which should greatly simplify the ingestion process.

INTER-004 - DataHub Setup

This experiment could not be concluded as the minimum system requirements for executing DataHub's docker containers were 10GB of RAM - this was not feasible for execution on the System B, using the Windows Subsystem for Linux (WSL) which possessed only 4GB of RAM, or System A, which in total possessed 12GB, but used 2GB for the OS, and needed an additional 2GB for the JVM[2] running Kafka.

Despite this setback, an attempt was still made to set-up DataHub and go through the execution process. This process involved downloading the container images, and deploying them using Docker. The following services were deployed, as part of DataHub's back-end:

- **Deployed Applications**
  - **Kafka, Zookeeper** - For the messages handled within DataHub.
  - **Elasticsearch** - For the search indexing and search engine used to browse the metadata catalog.
  - **MySQL** - To host the data and metadata.
  - **Neo4j Graph Database** - To build the underlying graphs that connect the metadata.
- **Internal Modules**
  - **DataHub Generalised Metadata Service (GMS)** - Ingestion and metadata management process
  - **DataHub React Frontend** - Web user interface for the DataHub catalog.

During the booting process, Elasticsearch and MySQL presented a number of errors. Kafka started up normally, but when attempting to pass messages between the numerous services, the brokers continuously disconnected and failed. Additionally, the DataHub **GMS**, a REST API deployed by DataHub to act as a processing unit for metadata arrivals, was not able to establish a connection to the MySQL service, thus never actually started up, and the DataHub front-end was never instantiated. Upon browsing the documentation, it seems that the hardware limitations of System B were a significant limiting factor, as the processes could not operate with under half of their minimum requirement. System C was not usable as nested virtualisation (required for running Docker containers in a Virtual Machine) was not enabled for the reserved VM.

Result - Experiment skipped due to hardware limitations.

Observations - While the experiment was not successful, documentation points to this use-case (Kafka ingestion with DataHub) using a very commonly used, successful and well documented/developed feature. References for this experiment are the *Metadata Ingestion Guide* [50] and the *Kafka Ingestion Guide* [51] from the official DataHub documentation.

[2]Java Virtual Machine

SYSTEM-001 - Full System Test

This experiment would inform on the qualities on **Compatibility**, **Interoperability** and **Performance** (end-to-end latency to test feasibility of RT/NRT scenarios). Under similar circumstances as the previous experiment, this analysis could not be concluded due to the hardware limitations of System A and System B.

The configuration would have been as described in Figure 6.6:

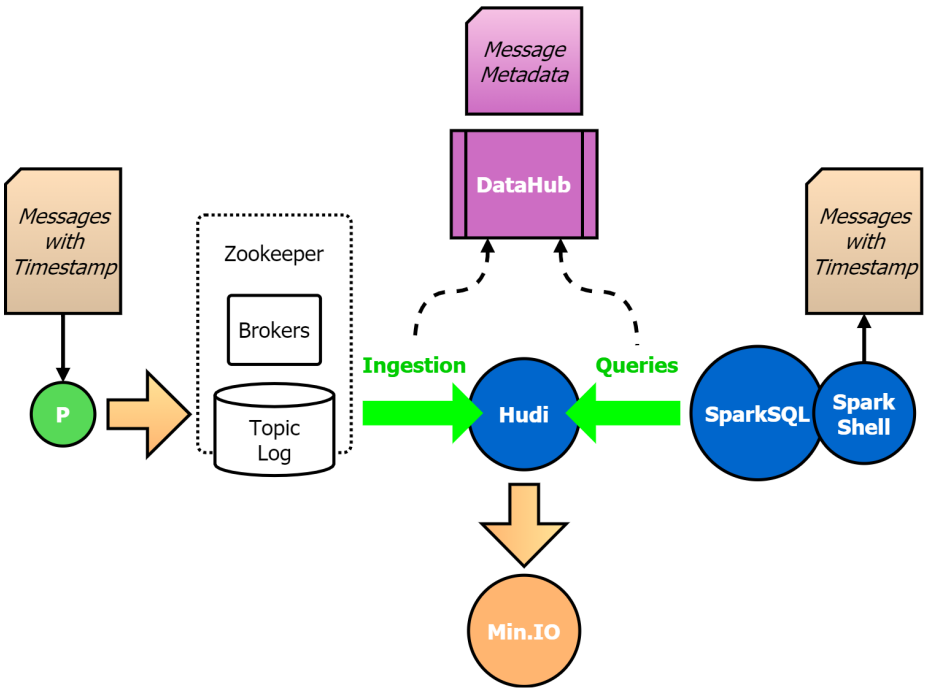

Figure 6.6: Experimental setup for SYSTEM-001. The Kafka Producer (Green P) would post messages to the topic which Hudi would ingest from, storing in Min.IO. SparkSQL would query it and compute end-to-end latency. DataHub would monitor the entire lifecycle.

The experiment's intent was to monitor the ingestion of data and the querying of data, and seeing the information populate the DataHub dashboard. Additionally the possibility of computing end-to-end latency for the entire framework would have enabled performance metric extraction. Through the previous experiments, it is possible to infer that the functional side of the framework is capable of operating simultaneously (as both Kafka ingestion through Hudi and SparkSQL Hudi queries were able to run in their own independent tests), however, to run all of the components together, more hardware resources would be necessary. Additionally, the execution of the DataHub catalog on top of the remaining system was not feasible for the reasons presented in the previous experiment log. The previously identified issue with nested virtualisation in System C once again made the execution of this experiment unfeasible.

Result - Experiment skipped due to hardware limitations.

Observations - DataHub's integrations with Kafka and Spark are quite well documented and seem to be relatively easy to set-up provided that the execution environment can support them.

Experiment Results and Conclusions

Due to hardware limitations, this experimental suite was not completed. However, some relevant information was extracted and the understanding of how the system operates in production was greatly increased.

Kafka, Hudi and Min.IO were proven to be a compatible coupling, with the flexibility and power of Kafka's clusters being readily configurable to use Hudi as a Lakehouse component, organising the incoming data and preparing an easily query-able metadata table. The flexibility of SparkSQL Hudi queries was also exposed, showing how it can be used to meet compliance goals related to data snapshotting, retention and lineage tracing.

Avenues for future work include:

- **Study the inter-connectivity of the whole system** - Through the use of more powerful hardware, the entire system can be executed and performance, scalability and consistency metrics can be gathered.
- **Experiment using the Kafka Connect Hudi Sink** - The new technology is still in development, but it shows a great deal of promise in the quick management of streaming Kafka pipelines.
- **Experiment with the Flink-centric ingestion system** - Due to time constraints, only one of the two major ingestion systems for Hudi was analysed (Spark). Flink is a very promising solution which can possibly make the Hudi ingestion process more optimized, simpler and powerful, by combining it with Flink's streaming data transformation pipelines.

6.3.5 Outcome

This iteration, despite its setbacks in the experimental stage, resulted in a considerable improvement of the requirement specification as well as a number of large changes in the architecture that brought it closer to a viable and possible solution for Altice Labs' use cases. The results of these processes can be summarised (layer-by-layer) as:

- **Ingestion Layer**
  - Integration of Spark into the Ingestion Layer for Hudi/Kafka bridging.
- **Storage Layer**
  - Experimental validation of S3/Min.IO as the storage solution, followed by a compatibility analysis with all currently identified technologies.
- **Serving Layer**
  - Technologies identified (Hudi, Iceberg, Delta Lake, LakeFormation), and features mapped and compared.
  - Selection of Hudi as the most promising candidate, followed by experimental validation with Kafka, S3/Min.IO and Spark.
- **Administration Layer**
  - Technologies identified (OpenMetadata, DataHub, Glue, Atlas), and features mapped and compared.
  - Selection of DataHub as the most promising candidate, experimental validation started but unfinished.
- **Orchestration Layer**
  - De-prioritization and reduction in scope.
  - Currently identified components (Nagios + Spark + Log4J) are compatible with all the currently identified technologies, lacking only experimental validation.
- **Requirement Specification**
  - Re-organization to account for the corrected Lakehouse component definition.
  - Re-structuring of the monitoring and infrastructure management requirements to account for the reduced scope.

As there was no time to perform a third iteration, this was the last step in the design of this architecture. The resulting information and requirement-to-technology mappings would be used to create the diagrams of the architecture as it stands.

Chapter 7

Final Architecture

With the iterative process concluded, the architecture's current state was documented for the final submission of the dissertation project and for delivery to Altice Labs S.A.. The specification uses the C4 model, described in Chapter 3 and presents three of the four views: **Context, Container, Component and Code**. The last level of abstraction in the C4 model (Code) was not developed.

The development process resulted in an architecture with an experimentally validated **data processing stack**, as well as a partially validated **data governance solution**. The **monitoring solution**, although de-prioritized as per Altice Labs' wishes, was analysed and composed for compatibility with the Prometheus toolset.

While the architecture is not ready for production scenarios, it is sufficiently developed for larger scale prototyping and presents evidences that the target framework can be functionally achieved with the selected components. Future work should ideally focus on utilising more powerful hardware to complement the experimental validation under more demanding environments and to better profile the scalability of the network under real conditions

The following sections will approach the three developed levels of the C4 model. Starting with an overview, then progressing to the C4 views: the *context* view, *container* view and then moving onto the several different *components*, presenting a diagram for each view, as well as a textual explanation clarifying the functionalities and qualities supported by each of the different parts of the architecture.

Alternatives are presented for the data pipeline section of the framework, identified through the state-of-the-art research and iterative process. Additionally, a section dedicated to the future work is presented, presenting the avenues for further development of this framework.

7.1 Overview

The final architecture can be summarised as a system built upon the use of a **Apache Hudi Lakehouse** on top of a **Min.IO/S3** object store (or Data Lake), which creates an environment fit for high-speed, highly scalable data serving through engines like **Spark** or **Presto**. This data store is fed by a multitude of **Apache Kafka** streams, either directly or after curating *ETL* via **Spark**. The framework's many data flows are audited, recorded and traced by the **Acryl DataHub** metadata catalog, and the infrastructure is constantly under health monitoring by a **Prometheus**-based monitoring dashboard.

7.1.1 Ingestion, Storage and Serving

This framework ingests data from a number of source systems through the creation of designated **Kafka Producers** that gather data (in push or pull configurations) and send them through a network of brokers onto the **Hudi Spark cluster**, which extracts metadata and categorizes data before storing it in a **Min.IO/S3 bucket**. This chain is built around *scalability*, *fault tolerance* and *high-throughput*, and allows for many ingestion styles under the *Kappa architecture pattern* (real-time and batch using the same component, Kafka)

Using **Spark and Presto**, the flexibility of the querying systems is maximised, as **Spark** enables the use of Hudi-specific audit-related queries (such as the *time-travel* or *data lineage* query) while **Presto** allows for extremely performant ad-hoc or preset querying. The use of **Apache Airflow** enables the automation of queries on the serving side by automating Spark or Presto jobs. Additionally, Presto can be executed on a Spark cluster, further exploiting the ability to execute these technologies in parallel. Both Spark and Presto come provisioned with *credential-based access control* which can be connected to external identity management software.

7.1.2 Administration and Governance

To establish an audit log and traceability for entire framework, the use of the **Acryl DataHub** metadata catalog helps build a *graph-based representation* of the data within the framework, showing origin, destination, accesses, modifies and lineage for any data object within the framework. This catalog is hooked to all the components which handle data within the framework, and will, over time, create a robust audit log to ensure data governance, safe data sharing and compliance are all possible within the framework.

7.1.3 Infrastructure Control

All the identified components present ways to extract metrics and data which can be interpreted by the **Prometheus** monitoring and alerting software, which allows for alerts and management of the system's infrastructure.

7.2 Context View

In the context view (Fig. 7.1) for the system, it is possible to see the three main actors involved with the operation of the framework:

- **The Process Manager** - Who is responsible for the health monitoring and configuration of the framework's components. Uses an external dashboard which receives data posted by the framework.
- **The End User** - Who uses an interface to interact with the data inside the framework. The interface lets the tenant/user see the data which he is authorized to see, query it, and process it through the use of the components described further in the chapter.
- **The Data Manager** - Who has direct access to the internal governance tools of the framework, and can manage tenant access and authorizations through an external IAM.

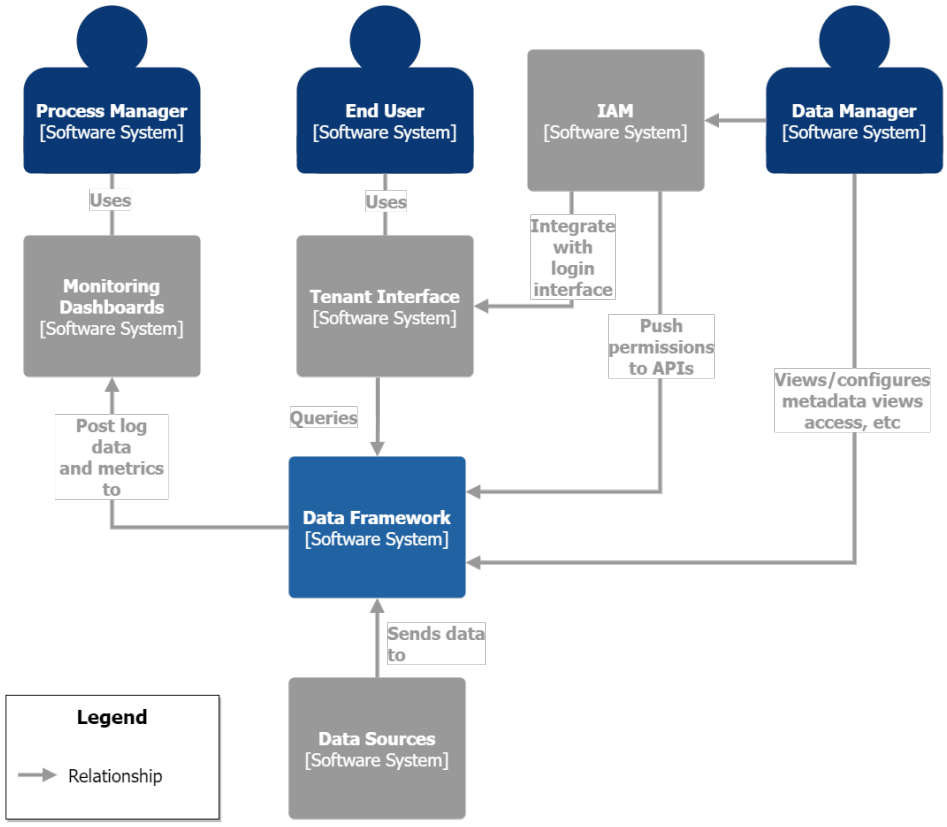

Figure 7.1: Context diagram for the architecture. Displayed are the three relevant actors and the systems they interact with.

The tenant interface, IAM and monitoring dashboard systems were not developed in this project and remain as avenues for future work.

7.3 Container View

In the container view (Fig. 7.2) the various executable parts of the framework are visible. Each of these containers is executed either on Docker/Kubernetes or on bare-metal. It is also possible to operate some of the containers via cloud-based deployment (namely the Min.IO storage component).

Here, the main data flows are exposed - the flow of data *from source* systems *to end-users*, the metadata governance flows and the infrastructure monitoring flows.

As previously described, the system offers a central path for data to travel through and from - the **Kafka Ingestion Cluster** receives data from outside sources, passing it onto the **Lakehouse Hudi/Spark Cluster**, which saves it to **Min.IO/S3**, and then passes data along to the **Serving Spark/Presto Cluster** whenever required by the tenants/end-users. Data may also be served from the Serving Cluster back to the Ingestion cluster for re-processing, ingestion of newly generated data or model saving. These three layers support the main qualities demanded of the framework, in regards to the function of data ingestion, processing, storage and serving:

- **Scalability** - All components were selected with scalability and cloud-native operations in mind, and experimental validation took place, ensuring they are inter-operable with each-other, albeit in a small, low resource scenario.
- **Performance** - The main points of data traffic - Kafka and Spark - are highly performant, and, due to their scalable, distributed and expansible nature, can guarantee performance at varying load levels.
- **Cost Control** - All components were selected with cost control in mind, and, aside from being open-source, include measures for rate control, performance limiting and resource management.

Along this path, the **Prometheus deployment** extracts metrics, logs and health data in order to maintain a record of system execution, and potentiate reactive and predictive maintenance efforts to ensure the long-term stability and scalability of the system (as some of the components require some level of manual adjustment to scale correctly).

For each of the significant points of data movement (*Ingestion, Storage and Serving*) the **DataHub Catalog** receives metadata transactional records, keeping a complete audit log of all data movements within the system. This allows for the verification of data lineage, provenance and access for any given data object in the framework, at all points of its lifecycle.

The DataHub catalog ships with an interface which enables the Data Manager's use cases of regulating data discovery, sharing and auditing activities. This catalog interface is also usable by the end-users, who have their own views of the system's data, typically filtered by their own authorization level as defined by the Data Manager.

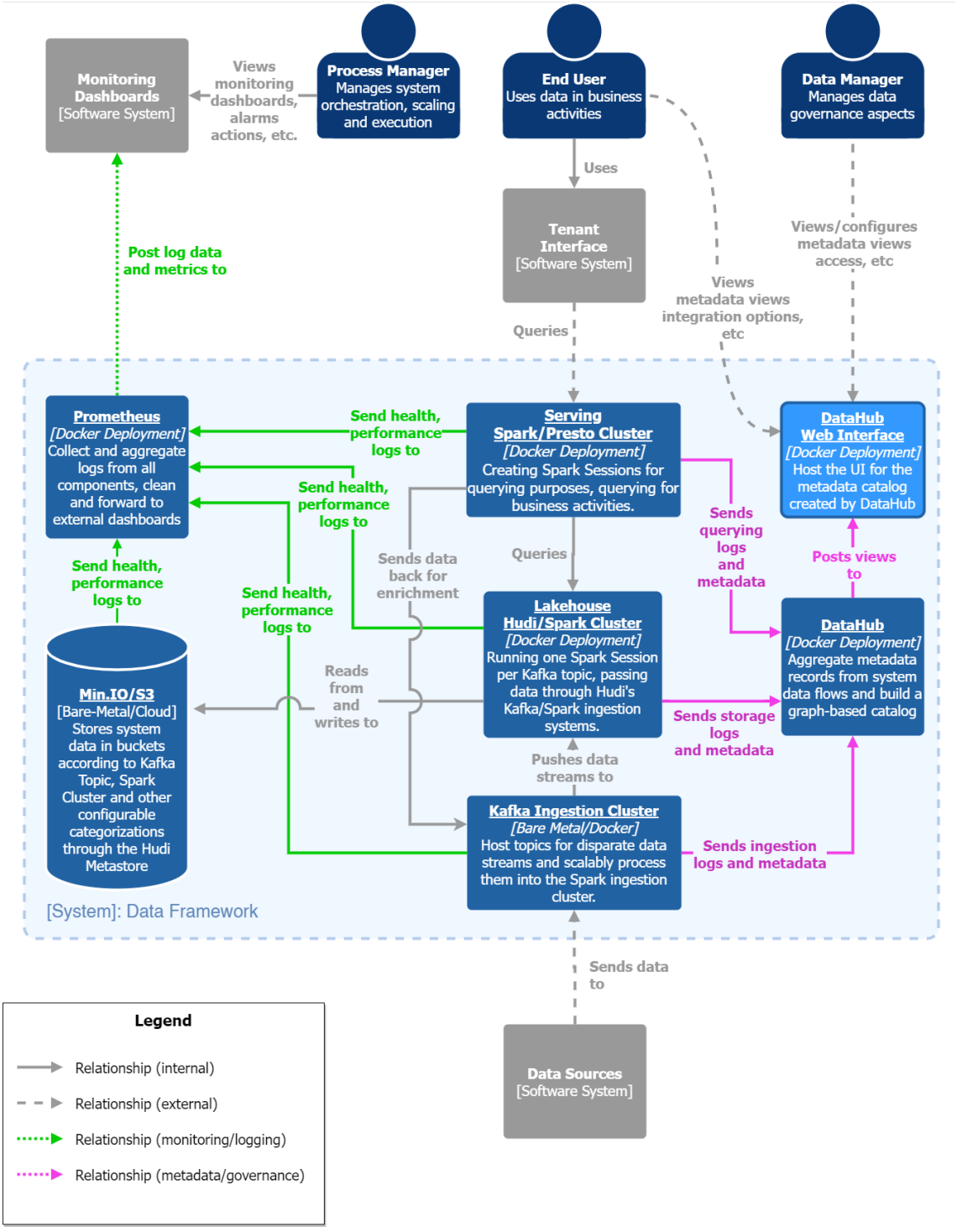

Figure 7.2: Container View of the Data Framework system. Grey arrows indicate data flows, green arrows indicate log flows and pink arrows indicate metadata flows.

7.4 Component Views

For each of the previously identified containers in the Data Framework system, a component diagram will be presented, exposing the inner components of each container as well as their connections/relationships to other containers.

7.4.1 Kafka Ingestion Cluster

As the container responsible for the entry of data into the system, the **Kafka Ingestion Cluster** (Fig. 7.3) relies on the creation of paths for data to travel to the framework in scalable way, fit for large volumes of data and a high-throughput execution profile. This cluster can be deployed on-prem or through cloud-based Kafka SaaS deployments like Confluent.

Functionalities

This container uses Kafka streams to serve the data sources outside of the system. They consist of a Producer-Broker-Consumer set that can be configured in push, pull or even a batch-style ingestion that aggregates data before passing it to storage. This is done through explicit programming of the logic within the Kafka Producer modules (where they can be connected to any API, legacy system or external data source).

Additionally, there's a separate Kafka stream for *enrichment* and *data integration* purposes. This stream receives data from the end-users (through the Serving Spark/Presto Cluster) and inserts it into the framework, making it possible for data enrichment and integration practices to have their own ingestion path, which can even include intermediate processing if necessary.

The Kafka clusters and components are all monitored through the metrics and logs extracted from its coordinator - the Kafka Zookeeper - and they are passed onto the Prometheus monitoring software for log analytics, automated alarms and even actions. The several Kafka topics are connected to a DataHub plugin, syncing the data, along with its metadata, into the DataHub catalog.

Qualities

Through the use of Kafka's native scalability, these clusters can maintain their performance even under highly concurrent, high-traffic workloads (through presizing for the intended loads) and more clusters/streams can be instantiated to serve tenant's needs in an elastic way.

Experimental validation indicates that Kafka can serve in this scenario with very good results in *integrity*, *scalability* and *performance*. Its flexible implementation also satisfies the requirements of *modifiability*, enabling a nearly limitless palette of external connections.

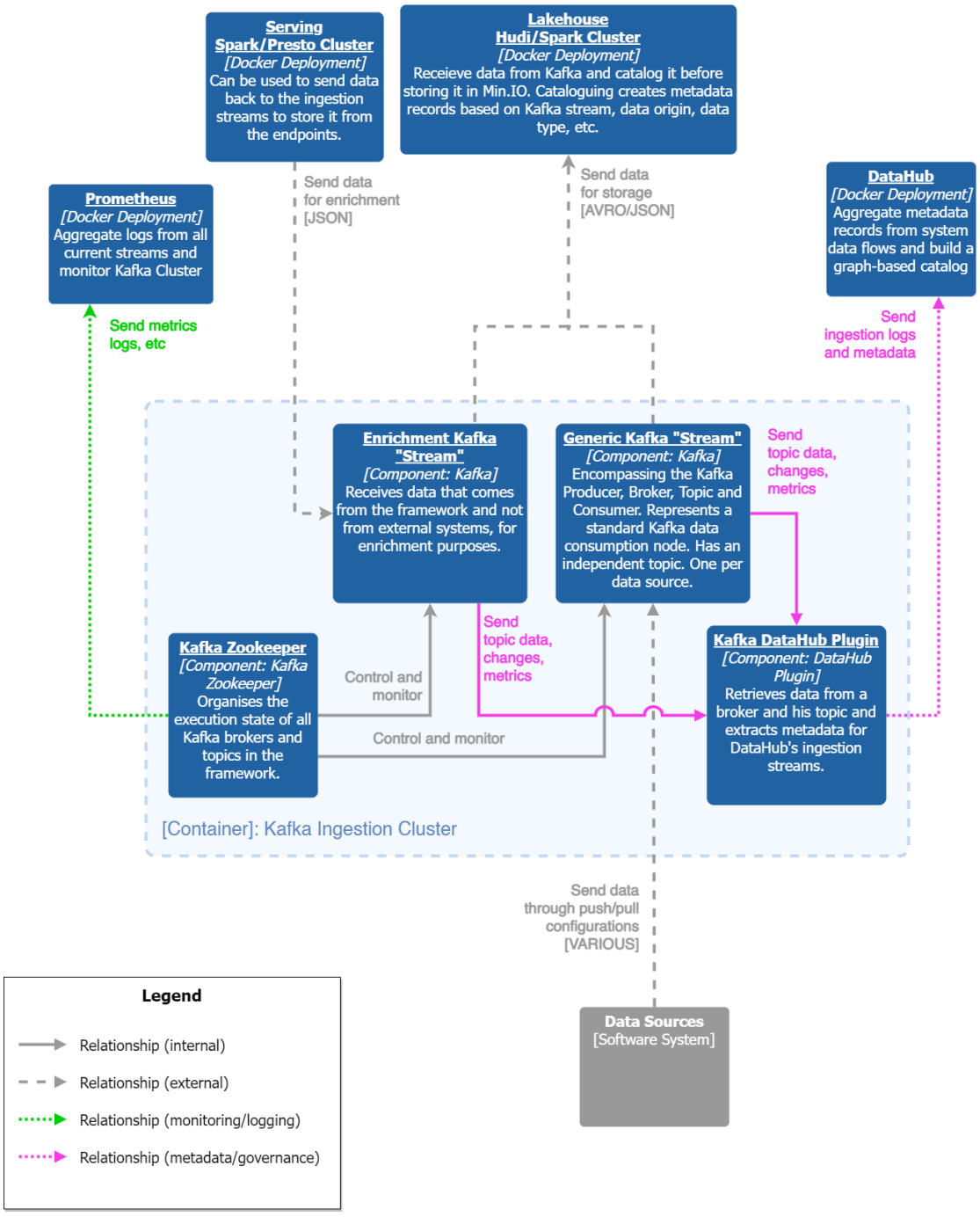

Figure 7.3: Component View of the Kafka Ingestion Cluster component. Grey arrows indicate data flows, green arrows indicate log flows and pink arrows indicate metadata flows. Additionally, dashed grey lines indicate data flows from/to outside the component.

7.4.2 Lakehouse Hudi/Spark Cluster

Once data has entered the system it is the job of the **Lakehouse Hudi/Spark Cluster** (Fig. 7.4) to categorize the data, store it, and serve it to the numerous services which rely on it. The tools used to create this container are extensively documented, namely in the identified pairings (Hudi/Min.IO and Spark/Presto), and rely on open-source technology, with both on-prem and cloud-based deployments available.

Functionalities

Through the use of Hudi as a Data Lakehouse platform, it is possible to empower the Min.IO/S3 object storage (typically used as a Data Lake) a build a more flexible and performant system. The Hudi/Min.IO combination is able to serve the required functionalities by enabling:

- **Lineage**, **time-travel** and **audit** queries (via SparkSQL).
- Build **custom categorizations** for data streams.
- Create **data retention policies** for specific data types/categories.

Hudi also natively supports two very powerful query engines: the aforementioned SparkSQL and Presto, with a large array of native integrations. Additionally, because Hudi runs on a Spark cluster, it is possible to create Spark applications for ETL jobs and other associated data transformations/integrations that can be configured to run automatically on designated input streams (functionally creating a curated data path).

The Prometheus dashboard is connected to Hudi's Spark cluster, where metrics are exported to monitor the performance and health of this cluster. Data flows are monitored and logged through the use of metadata extraction via the Spark DataHub agent, which is configured to automatically parse Spark metadata transactions and log them in the Catalog.

Qualities

The Hudi/Min.IO/S3 system is a robust, *performant*, *scalable* and *highly available* set-up, featuring many options to maximise these qualities, such as parallel execution, redundancy and multi-node operations with error correction.

Experimental validation ensures compatibility between the two, as well as successful data transit into the open file formats used by Hudi, as well as the reverse path of re-building from these formats into easily processed JSON files through the Hudi query system, fulfilling specified requirements of data *integrity*.

The ability to perform lineage and time-travel queries also fulfils non-functional requirements of *auditability* and *security*, which aids in adoption of compliant strategies.

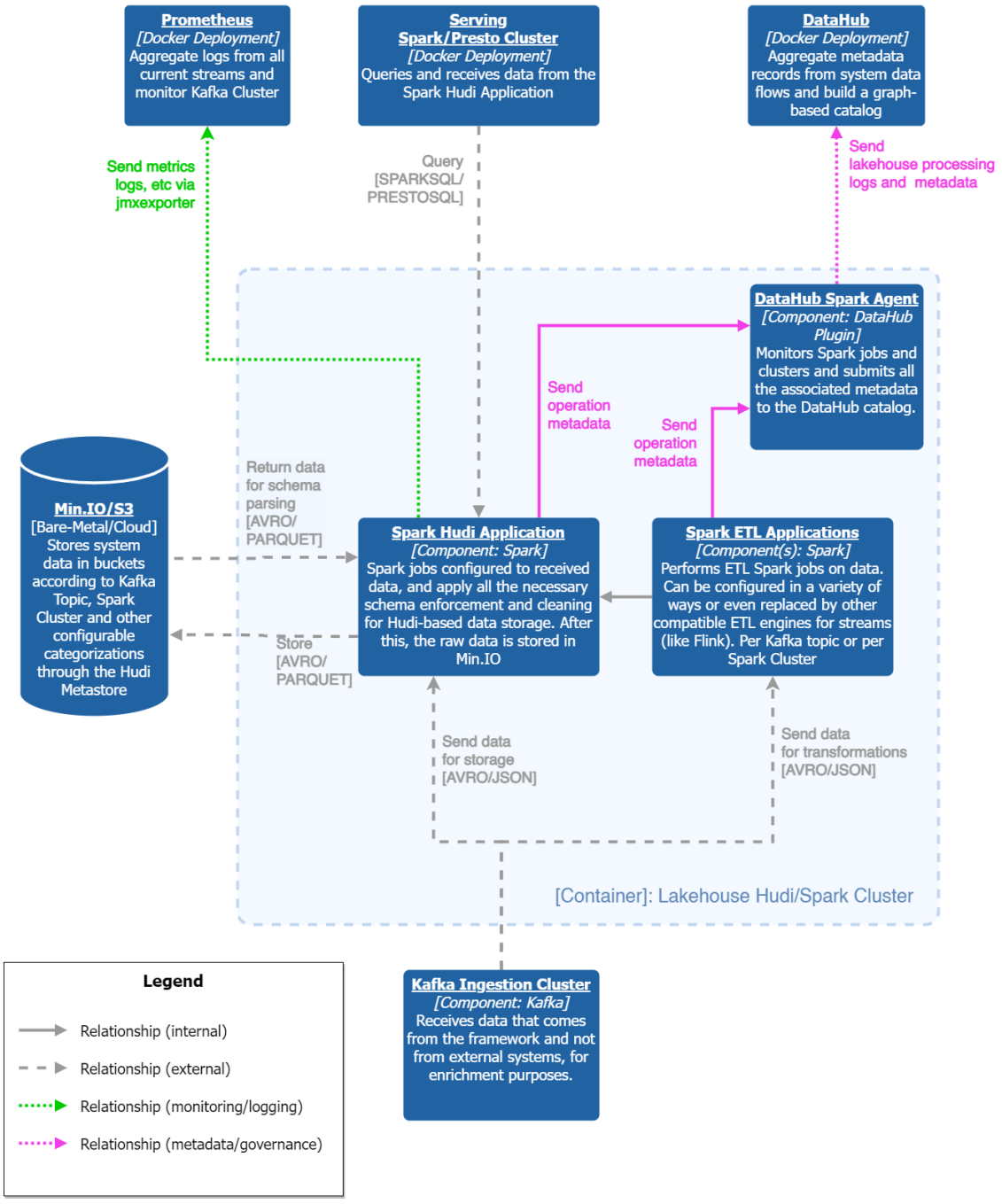

Figure 7.4: Component View of the Lakehouse Hudi/Spark Cluster component. Grey arrows indicate data flows, green arrows indicate log flows and pink arrows indicate metadata flows. Additionally, dashed grey lines indicate data flows from/to outside the component.

7.4.3 Serving Spark/Presto Cluster

To interact with the stored data, the **Serving Spark/Presto Cluster** (Fig. 7.5) container leverages the Spark execution environment to create a performant and flexible querying layer, featuring open-source and highly documented technologies fit for on-prem and cloud-based deployment.

Functionalities

The container relies on the use of the SparkSQL and Presto to query the Hudi service, allowing for a variety of query types (ad-hoc, preset, lineage, time-travel) and for a considerable degree of control over the authorization and access of the end-user. This is performed through the use of interfacing APIs which may connect to an external *Identity and Access Management* (IAM) software, providing access control to tenant queries.

To send data into the framework, a serving-side API enables data to be routed back through the Kafka ingestion cluster, and go through the categorization processes of the Hudi lakehouse.

The use of an Airflow scheduling component enables end-users to automate queries to their external services, and it ships with access control associated to its central dashboard. This feature enables the automation of data presentation to external AI/ML models, and also the snapshotting and saving of models/model checkpoints to long-term storage.

Much like the previous containers, the DataHub catalog tracks all uses of data into the metadata catalog, while Prometheus manages the health and execution state of the container's components.

Qualities

The use of Presto and Spark as querying engines allows the system's queries to be very fast, executing in a highly *performant* way, even in ad-hoc scenarios.

By routing the accesses to data through APIs it is possible to ensure a large degree of control, interfacing with external IAM software and enabling domain lock-out. Spark, through combination with Hudi, also enables credentials to be propagated through the querying process, returning only the data that the user has access to, omitting fields that they are not authorized to see.

This, along with the recording of all data movements in DataHub fulfils the qualities of *security*, *traceability* and *privacy* which are required for a safe data handling and sharing environment, potentiating the evolution to a Data Mesh scenario.

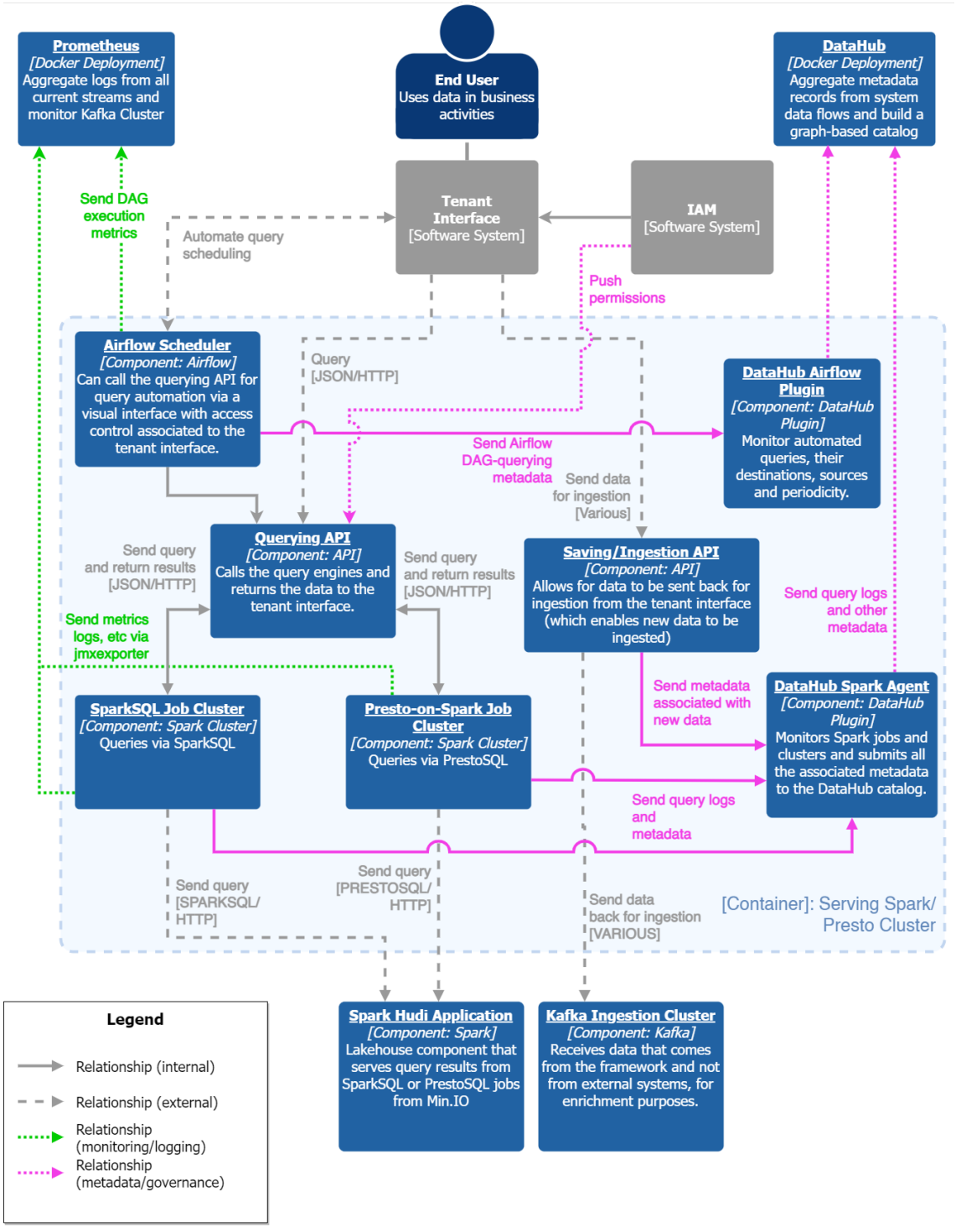

Figure 7.5: Component View of the Serving Spark/Presto Cluster component. Grey arrows indicate data flows, green arrows indicate log flows and pink arrows indicate metadata flows. Additionally, dashed grey lines indicate data flows from/to outside the component.

7.5 Alternatives

Along with the selected technologies for the final architecture, some of the researched components are, based on the selection criteria described in the previous chapter, possible viable alternatives for the functionalities of the main data pipeline.

As research could not cover or experiment on the identified metadata catalog alternatives, this section will not present any alternatives for the data governance side of the framework, as DataHub is both the most featured and most well documented choice in this area.

Additionally, some considerations can be made on the management and monitoring components, although mostly these rely on the use of the built-in management tools provided by cloud-native service providers.

The alternatives presented in this section provide the same feature-set as the selected components, but were not selected as the exclusion process undertaken in the refinement stages favoured a more suitable alternative. Nonetheless, they will be presented as they may present a better fit if new conditions arise in the future of the Data Framework Architecture project.

7.5.1 Ingestion

For Ingestion, as a replacement of Kafka, few components present as much flexibility and performance. However, there are options, namely the ones offered by the **AWS Kinesis Suite**, which present very good performance and scalability features and an unparalleled level of availability. They can be described as follows:

- **AWS Kinesis Data Streams** - Real-time streaming service which serves as a drop-in replacement for Kafka, featuring a lot of the same principles (records, re-transmission and "topic"-based streams). Like Kafka, the scalability is manually performed by pre-sizing and adding more "topics" or, as Kinesis documentation refers, *shards*. It specializes in real-time streaming featuring low latency by default.
- **AWS Kinesis Data Firehose** - Batch, un-managed data streaming solution that focuses on high throughput, while sacrificing some the low latency provided by a more structured service. Latency with Firehose is often above 60ms for a given data record's transmission, making it less suitable for real-time workloads, and more useful for massive transfers of data under less strict requirements.

It should be noted that, while these services provide similar functionalities, they are more advantageously used when the rest of the chosen technology stack is also within the AWS Suite, as it ends up providing native, "one-click" integration, as well as a connected and unified management dashboard system.

Alternatively, it is possible to offload the management of the Kafka cluster to an external service provider, maintaining the exact same functionality while gaining the benefit of high availability and improved latency.

Both the AWS and externally managed Kafka services will, however, incur in costs that can become exacerbated by the multitude of streams demanded by a multi-tenant architecture.

7.5.2 Storage and Serving

The storage functionalities and qualities, provided in the architecture by S3/Min.IO with Hudi as the Lakehouse component, can be achieved using other platforms, albeit with significant changes being necessary to the serving stack. As such, these alternatives are grouped together.

AWS S3-centric

By using S3 and choosing to use the AWS managed cloud-native solution, it is possible to also use **AWS LakeFormation** and **AWS Glue** to achieve the Lakehouse functionality. However, the compatibility with the currently chosen Serving stack is not clear, and instead, it is recommended that this use is paired with other AWS services, as these integrate seamlessly without the need for extensive modification and untested pairings.

As per the research conducted and presented in Chapter IV, the use of **AWS Athena** and **Redshift** can, in combination, result in a powerful interactive query engine which can satisfy the requirements of the Serving layer.

HDFS-centric

These solutions typically present some challenges, as the Lakehouse components typically rely on tailor made connectors fit for more recent object storage services (such as S3, Azure, Google Cloud).

However, in this case, Apache Hudi presents an HDFS connector which may enable it to resolve this problem, however, experimentation would be necessary to ensure compatibility of the stack. If possible, this would make the current serving stack compatible, and the requirements would likely still be met under the HDFS-based platform.

7.6 Future Work

With the work taken as far as the dissertation project's constraints would allow, the avenues for future work were identified to be delivered alongside the architecture and requirement specifications.

Three main areas were identified:

- **Conclude the experimental validation** - While the main data pipeline was validated experimentally, the conditions for the tests were simply meant for rapid prototyping and used significantly reduced hardware specifications in their execution. Additionally, the hardware limitations made it impossible to test the usability of the metadata catalog correctly, and thus this could also be worth exploring. The two main tasks here are:
  - Test the framework with more resources, using distributed configurations, virtualization and create synthetic scenarios closer to the estimated production workload.
  - Finish testing the governance aspects (catalog metadata visualization) with a suite of tests designed around usability.
- **Refine the monitoring solution** - As the monitoring was indicated as an area which should not be focused on, as internally, Altice Labs S.A. already had significant experience in the area and had proposed existing tooling (the Prometheus toolset), it may be valuable to explore the monitoring and management drivers more and see how Prometheus can be potentially used in the PaaS/IaaS scenario.
- **Design the interfaces for the users** - This was outside of the scope of the project, but, nonetheless, remains a very important part for the production architecture, as some of the concepts associated with managing a multi-tenant architecture can not be explicitly solved through architectural decisions, but rather through interface design processes.

And, while there are still some areas where improvement is possible, implementation could feasibly begin on the data pipeline side, as the technologies which were presented, as well as their data flows and component relationships will allow for the identified functional and non-functional requirements to be met.

Chapter 8

Conclusion

The objective of this project was to develop an architecture fit for Altice Labs S.A.'s goal of developing a data processing framework fit for a multi-tenant IaaS or PaaS scenario and incorporating innovative data management principles.

To achieve this, a requirement engineering process was followed by an iterative architecture design methodology, which took the initial preliminary drafts to a more refined, well-rounded architecture, validated not only through documentation and literature review, but also through experimental processes that ensure that it can meet its identified requirements. These processes were backed up by a significant research effort into state-of-the-art data management models, approaches and frameworks, as well as a large amount of contextual information regarding data processing paradigms.

The dissertation project concluded with the second iteration of the architecture development process. Through these two passes, the architecture achieved significant coverage of the requirements, with experimentally validated components, namely those related to the handling of the system's data and its adaptation into a multi-tenant, scalable IaaS/PaaS-capable system. While some ancillary aspects of the framework were not covered by experimentation due to time and hardware constraints, they nonetheless were the target of significant exploration and documentation, laying out the groundwork for future work on the project.

Future work can be focused on implementing the data processing pipelines for production testing, for which there is already a stable and well-documented validation, further refining the architecture, continuing the iterative process with a focus on the areas which were de-prioritized in Iteration #2, or on performing a design process for the tenant interface.

And, from a learning perspective, the project provided an opportunity to explore the processes of systematic software development, demonstrating the power of a consistent and organized effort to explore requirements, architectural drivers and research avenues to build the product most suited to the needs of a client. Over the course of the project numerous exchanges with Altice Labs S.A. provided the experience of working with external clients, communicating change, risks and progress - skills which will prove invaluable in future endeavours.

References

[1] Rene Abraham, Johannes Schneider, and Jan vom Brocke. Data governance: A conceptual framework, structured review, and research agenda. *International Journal of Information Management*, 49:424–438, 12 2019. ISSN 0268-4012. doi: 10.1016/J.IJINFOMGT.2019.07.008.

[2] Acryl. The #1 Open Source Data Catalog - DataHub, 2023.

[3] Actian. DataConnect - Map Connectors - Data File Formats, 2020. URL https://docs.actian.com/dataconnect/11.5/index.html#page/User/Data_File_Formats.htm.

[4] ActiveMQ. ActiveMQ, 2023. URL https://activemq.apache.org/.

[5] Alluxio. Caching - Alluxio v2.9.3 (stable) Documentation, 2022. URL https://docs.alluxio.io/os/user/stable/en/core-services/Caching.html?utm_source=prestodb&utm_medium=prestodocs#configuring-alluxio-storage.

[6] Alluxio. Alluxio - Data Orchestration for the Cloud, 2023. URL https://www.alluxio.io/.

[7] Altice Labs S.A. Altice Labs – Enabling Digital Society, 2023. URL https://www.alticelabs.com/.

[8] Amazon Web Services. Cloud Computing Services - Amazon Web Services (AWS), 2023. URL https://aws.amazon.com/?nc2=h_lg.

[9] Amazon Webservices. Build a Lake House Architecture on AWS | AWS Big Data Blog, 2022. URL https://aws.amazon.com/blogs/big-data/build-a-lake-house-architecture-on-aws/.

[10] Amazon Webservices. Data Catalog and crawlers in AWS Glue - AWS Glue, 2023. URL https://docs.aws.amazon.com/glue/latest/dg/catalog-and-crawler.html.

[11] Amazon Webservices. Amazon API Gateway, 2023. URL https://aws.amazon.com/api-gateway/.

[12] Amazon Webservices. Interactive SQL - Serverless Queries - AWS Athena, 2023. URL https://aws.amazon.com/athena/.

[13] Amazon Webservices. Fast NoSQL Key-Value Database – Amazon DynamoDB – Amazon Web Services, 2023. URL https://aws.amazon.com/dynamodb/.

[14] Amazon Webservices. Secure and resizable cloud compute – Amazon EC2 – Amazon Web Services, 2023. URL https://aws.amazon.com/ec2/.

[15] Amazon Webservices. Process and Analyze Streaming Data – Amazon Kinesis – Amazon Web Services, 2023. URL https://aws.amazon.com/kinesis/.

[16] Amazon Webservices. Serverless Computing - AWS Lambda, 2023. URL https://aws.amazon.com/lambda/.

[17] Amazon Webservices. Data Lake Governance - AWS LakeFormation, 2023. URL https://aws.amazon.com/lake-formation/.

[18] Amazon Webservices. Cloud Object Storage – Amazon S3 – Amazon Web Services, 2023. URL https://aws.amazon.com/s3/.

[19] Amazon Webservices. Fully Managed Message Queuing – Amazon Simple Queue Service – Amazon Web Services, 2023. URL https://aws.amazon.com/sqs/.

[20] Samuil Angelov, Paul Grefen, and Danny Greefhorst. A classification of software reference architectures: Analyzing their success and effectiveness. *2009 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA 2009*, pages 141–150, 2009. doi: 10.1109/WICSA.2009.5290800.

[21] Apache Hudi. Spark Guide | Apache Hudi, 2022. URL https://hudi.apache.org/docs/quick-start-guide/.

[22] Apache Software Foundation. Apache Hadoop, 2006. URL https://hadoop.apache.org/.

[23] Apache Software Foundation. Apache Hive, 2010. URL https://hive.apache.org/.

[24] Apache Software Foundation. Apache Kafka, 2011. URL https://kafka.apache.org/.

[25] Apache Software Foundation. Apache Spark, 2014. URL https://spark.apache.org/.

[26] Apache Software Foundation. Apache Hudi, 2016. URL https://hudi.apache.org/.

[27] Apache Software Foundation. Apache Flume, 2022. URL https://flume.apache.org/.

[28] Apache Software Foundation. Welcome to The Apache Software Foundation!, 2023. URL https://www.apache.org/.

[29] Apache Software Foundation. Apache Atlas, 2023. URL https://atlas.apache.org/#/.

[30] Apache Software Foundation. Apache Flink® — Stateful Computations over Data Streams | Apache Flink, 2023. URL https://flink.apache.org/.

[31] Apache Software Foundation. Apache Iceberg, 2023. URL https://iceberg.apache.org/.

[32] Michael Armbrust, Ali Ghodsi, Reynold Xin, Matei Zaharia, and Uc Berkeley. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2021.

[33] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture In Practice*. 1 2003. ISBN 978-0321154958.

[34] Len Bass, John Bergey, Paul Clements, Paulo Merson, Ipek Ozkaya, and Raghvinder Sangwan. A Comparison of Requirements Specification Methods from a Software Architecture Perspective. 2006.

[35] P Baxendale and E F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 6 1970. ISSN 15577317. doi: 10.1145/362384.362685. URL https://dl.acm.org/doi/10.1145/362384.362685.

[36] Andrew L. Beam and Isaac S. Kohane. Big Data and Machine Learning in Health Care. *JAMA*, 319(13):1317–1318, 4 2018. ISSN 0098-7484. doi: 10.1001/JAMA.2017.18391. URL https://jamanetwork.com/journals/jama/fullarticle/2675024.

[37] Blend Berisha, Endrit Mëziu, and Isak Shabani. Big data analytics in Cloud computing: an overview. *Journal of Cloud Computing (Heidelberg, Germany)*, 11(1):24, 12 2022. ISSN 2192113X. doi: 10.1186/S13677-022-00301-W. URL /pmc/articles/PMC9362456//pmc/articles/PMC9362456/?report=abstracthttps://www.ncbi.nlm.nih.gov/pmc/articles/PMC9362456/.

[38] Piero A. Bonatti and Sabrina Kirrane. Big Data and Analytics in the Age of the GDPR. *Proceedings - 2019 IEEE International Congress on Big Data, BigData Congress 2019 - Part of the 2019 IEEE World Congress on Services*, pages 7–16, 7 2019. doi: 10.1109/BIGDATACONGRESS.2019.00015.

[39] Simon Brown and Thomas Betts. The C4 Model for Software Architecture, 6 2018. URL https://www.infoq.com/articles/C4-architecture-model/.

[40] Cambia. The Lens - Free \& Open Patent and Scholarly Search, 2023. URL https://www.lens.org/.

[41] Carlos J. Fernández Candel, Diego Sevilla Ruiz, and Jesús J. García-Molina. A Unified Metamodel for NoSQL and Relational Databases. *Information Systems*, 104, 5 2021. doi: 10.1016/j.is.2021.101898. URL http://arxiv.org/abs/2105.06494http://dx.doi.org/10.1016/j.is.2021.101898.

[42] Alberto Hernandez Chillon, Diego Sevilla Ruiz, Jesus Garcia Molina, and Severino Feliciano Morales. A Model-Driven Approach to Generate Schemas for Object-Document Mappers. *IEEE Access*, 7:59126–59144, 2019. ISSN 21693536. doi: 10.1109/ACCESS.2019.2915201.

[43] CISUC. POWER - Empowering a digital future - CISUC, 2021. URL https://www.cisuc.uc.pt/en/projects/power.

[44] Clairvoyant. Big Data File Formats, 2021. URL https://www.clairvoyant.ai/blog/big-data-file-formats.

[45] Cloudera. Cloudera | The hybrid data company, 2023. URL https://www.cloudera.com/.

[46] Confluent Cloud. Confluent: Apache Kafka Reinvented for Multi-Cloud Data Streaming, 2023. URL https://www.confluent.io/.

[47] Carlos Costa and Maribel Yasmina Santos. Big Data: state-of-the-art concepts, techniques, technologies, modelling approaches and research challenges. *IAENG International Journal of Computer Science*, pages 285–301, 2017. ISSN 1819-656X. URL https://hdl.handle.net/1822/46855.

[48] Databricks. What is a Medallion Architecture?, 2023. URL https://www.databricks.com/glossary/medallion-architecture.

[49] Databricks. Data Lakehouse Architecture and AI Company | Databricks, 2023. URL https://www.databricks.com/.

[50] DataHub. Introduction to Metadata Ingestion | DataHub, 2023. URL https://datahubproject.io/docs/metadata-ingestion/#recipes.

[51] DataHub. Kafka Ingestion | DataHub, 2023. URL https://datahubproject.io/docs/generated/ingestion/sources/kafka/.

[52] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *6th Symposium on Operating System Design and Implementation (OSDI 2004)*, 2024.

[53] Zhamak Dehghani. *Data Mesh - Delivering Data-Driven Value at Scale*. O'Reilly Media, first edition, 2021. ISBN 978-1-492-09232-2.

[54] Delta Lake. Home | Delta Lake, 2023. URL https://delta.io/.

[55] James Dixon. Pentaho, Hadoop, and Data Lakes | James Dixon's Blog, 2010. URL https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/.

[56] Docker Inc. Docker: Accelerated, Containerized Application Development, 2023. URL https://www.docker.com/.

[57] Athanasios S. Drigas and Panagiotis Leliopoulos. The Use of Big Data in Education. *IJCSI International Journal of Computer Sciences*, 11(5):58–63, 9 2014. ISSN 1694-0784.

[58] Stefan Dzalev and Marjan Gusev. Evaluation of Scalability and Multitenancy: A Use-Case. *2021 29th Telecommunications Forum, TELFOR 2021 - Proceedings*, 2021. doi: 10.1109/TELFOR52709.2021.9653373.

[59] Lisa Ehrlinger and Wolfram Wöß. Towards a Definition of Knowledge Graphs. In *Joint Proceedings of the Posters and Demos Track of 12th International Conference on Semantic Systems - SEMANTiCS2016 and 1st International Workshop on Semantic Change & Evolving Semantics (SuCCESS16*, 6 2016.

[60] Enrique de Argaez. World Internet Users Statistics and 2022 World Population Stats, 2022. URL `https://www.internetworldstats.com/stats.htm`.

[61] European Commission. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance). 2016.

[62] European Commission. Reform of EU data protection rules - Rules for business and organisations, 2018. URL `https://commission.europa.eu/law/law-topic/data-protection/reform/rules-business-and-organisations_en`.

[63] European Union. A European approach to artificial intelligence | Shaping Europe's digital future, 2023. URL `https://digital-strategy.ec.europa.eu/en/policies/european-approach-artificial-intelligence`.

[64] FBI Fortune Business Insights. Big Data Analytics Market Size, 2022. URL `https://www.fortunebusinessinsights.com/big-data-analytics-market-106179`.

[65] Corinna Giebler, Christoph Gröger, Eva Hoos, Holger Schwarz, and Bernhard Mitschang. Leveraging the Data Lake: Current State and Challenges. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11708 LNCS:179–188, 2019. ISSN 16113349. doi: 10.1007/978-3-030-27520-4{_}13/COVER. URL `https://link.springer.com/chapter/10.1007/978-3-030-27520-4_13`.

[66] Abel Goedegebuure, Indika Kumara, Stefan Driessen, Dario Di Nucci, Geert Monsieur, Willem-jan van den Heuvel, and Damian Andrew Tamburri. Data Mesh: a Systematic Gray Literature Review. 4 2023. doi: 10.1145/nnnn. URL `https://arxiv.org/abs/2304.01062v1`.

[67] Mert Onuralp Gokalp, Kerem Kayabay, Mohamed Zaki, Altan Kocyigit, P. Erhan Eren, and Andy Neely. Open-Source Big Data Analytics Architecture for Businesses. *1st International Informatics and Software Engineering Conference: Innovative Technologies for Digital Transformation, IISEC 2019 - Proceedings*, 11 2019. doi: 10.1109/UBMYK48245.2019.8965572.

[68] Paola Gómez, Rubby Casallas, and Claudia Roncancio. Automatic schema generation for document-oriented systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture*

Notes in Bioinformatics), 12391 LNCS:152–163, 2020. ISSN 16113349. doi: 10.1007/978-3-030-59003-1{_}10/COVER. URL https://link.springer.com/chapter/10.1007/978-3-030-59003-1_10.

[69] Ian Gorton and John Klein. Distribution, data, deployment: Software architecture convergence in big data systems. *IEEE Software*, 32(3):78–85, 5 2015. ISSN 07407459. doi: 10.1109/MS.2014.51.

[70] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 12 1983. ISSN 15577341. doi: 10.1145/289.291. URL https://dl.acm.org/doi/10.1145/289.291.

[71] IBM. What is a data fabric? | IBM, 2023. URL https://www.ibm.com/topics/data-fabric.

[72] IEEE. IEEE Recommended Practice for Software Requirements Specifications. *IEEE Std 830-1998*, pages 1–40, 10 1998.

[73] IEEE. ISO/IEC 25010:2011 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. *ICS : 35.080*, 2011. URL https://www.iso.org/standard/35733.html.

[74] Joseph. Ingeno. *Software Architect's Handbook : Become a Successful Software Architect by Implementing Effective Architecture Concepts.* Packt Publishing Ltd, 2018. ISBN 9781788624060. URL https://www.packtpub.com/product/software-architects-handbook/9781788624060.

[75] William H. Inmon. *Building the Data Warehouse Third Edition*. Fourth edition, 2002.

[76] Instituto de Telecomunicações. Instituto de Telecomunicações, 2023. URL https://www.it.pt/.

[77] Instituto Pedro Nunes. IPN - Instituto Pedro Nunes, 2023. URL https://www.ipn.pt/.

[78] Veit Jahns. Data Fabric and Datafication. *ACM SIGSOFT Software Engineering Notes*, 47(4):30–31, 9 2022. ISSN 0163-5948. doi: 10.1145/3561846.3561854. URL https://dl.acm.org/doi/10.1145/3561846.3561854.

[79] Rick Kazman, Mark Klein, and Paul Clements. ATAM: Method for Architecture Evaluation. 2000.

[80] Jay Kreps. Questioning the Lambda Architecture - O'Reilly Radar, 2014. URL http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html.

[81] Krish Krishnan. *Data Warehousing in the Age of Big Data*. Elsevier Inc., 2013. ISBN 9780124058910. doi: 10.1016/C2012-0-02737-8. URL http://www.sciencedirect.com:5070/book/9780124058910/data-warehousing-in-the-age-of-big-data.

[82] Sara Landset, Taghi M. Khoshgoftaar, Aaron N. Richter, and Tawfiq Hasanin. A survey of open source tools for machine learning with big data in the Hadoop ecosystem. *Journal of Big Data*, 2(1):1–36, 12 2015. ISSN 21961115. doi: 10.1186/S40537-015-0032-1/FIGURES/5. URL https://journalofbigdata.springeropen.com/articles/10.1186/s40537-015-0032-1.

[83] Anthony J Lattanze. Architecture Centric Design Method, 2006.

[84] Anthony J. Lattanze. *Architecting Software Intensive Systems: A Practitioners Guide*. Auerbach Publications, 2008. ISBN 1420045695.

[85] Alexandra L'Heureux, Katarina Grolinger, Hany F. Elyamany, and Miriam A.M. Capretz. Machine Learning with Big Data: Challenges and Approaches. *IEEE Access*, 5:7776–7797, 2017. ISSN 21693536. doi: 10.1109/ACCESS.2017.2696365.

[86] Jimmy Lin. The Lambda and the Kappa. *IEEE Internet Computing*, 21(05): 60–66, 9 2017. ISSN 1089-7801. doi: 10.1109/MIC.2017.3481351.

[87] Inês Araújo Machado, Carlos Costa, and Maribel Yasmina Santos. Data Mesh: Concepts and Principles of a Paradigm Shift in Data Architectures. *Procedia Computer Science*, 196:263–271, 1 2022. ISSN 1877-0509. doi: 10.1016/J.PROCS.2021.12.013.

[88] Rupa Mahanti. *Data quality : dimensions, measurement, strategy, management, and governance*. 2019. ISBN 0873899776. URL https://www.researchgate.net/publication/358523910_Data_Quality_Dimensions_Measurement_Strategy_Management_and_Governance.

[89] Nathan Marz. How to beat the CAP theorem - thoughts from the red planet - thoughts from the red planet, 2011. URL http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

[90] Minio Inc. MinIO | High Performance, Kubernetes Native Object Storage, 2023. URL https://min.io/.

[91] MySQL. MySQL, 2023. URL https://www.mysql.com/.

[92] Athira Nambiar and Divyansh Mundra. An Overview of Data Warehouse and Data Lake in Modern Enterprise Data Management. *Big Data and Cognitive Computing 2022, Vol. 6, Page 132*, 6(4):132, 11 2022. ISSN 2504-2289. doi: 10.3390/BDCC6040132. URL https://www.mdpi.com/2504-2289/6/4/132/htmhttps://www.mdpi.com/2504-2289/6/4/132.

[93] Fatemeh Nargesian, Erkang Zhu, Renée J Miller, Ken Q Pu UOIT, and Patricia C Arocena. Data Lake Management: Challenges and Opportunities. 12(12):1986–1989, 2019. doi: 10.14778/3352063.3352116. URL https://doi.org/10.14778/3352063.3352116.

[94] Thor Olavsrud. 3 keys to keeping your data lake from becoming a data swamp | CIO, 2017. URL https://www.cio.com/article/230163/3-keys-to-keep-your-data-lake-from-becoming-a-data-swamp.html.

[95] OpenMetadata. OpenMetadata: The Best Open Source Data Catalog, 2023.

[96] Oracle. Oracle | Cloud Applications and Cloud Platform, 2023. URL https://www.oracle.com/.

[97] C. L. Philip Chen and Chun Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information Sciences*, 275:314–347, 8 2014. ISSN 0020-0255. doi: 10.1016/J.INS.2014.01.015.

[98] PostgreSQL. PostgreSQL: The world's most advanced open source database, 2023. URL https://www.postgresql.org/.

[99] Presto Foundation and LLC LF Projects. Presto: Free, Open-Source SQL Query Engine for any Data, 2022. URL https://prestodb.io/.

[100] RabbitMQ. Messaging that just works — RabbitMQ, 2023. URL https://www.rabbitmq.com/.

[101] Fatimah Sidi, Payam Hassany Shariat Panahy, Lilly Suriani Affendey, Marzanah A. Jabar, Hamidah Ibrahim, and Aida Mustapha. Data quality: A survey of data quality dimensions. *Proceedings - 2012 International Conference on Information Retrieval and Knowledge Management, CAMP'12*, pages 300–304, 2012. doi: 10.1109/INFRKM.2012.6204995.

[102] Uthayasankar Sivarajah, Muhammad Mustafa Kamal, Zahir Irani, and Vishanth Weerakkody. Critical analysis of Big Data challenges and analytical methods. *Journal of Business Research*, 70:263–286, 1 2017. ISSN 0148-2963. doi: 10.1016/J.JBUSRES.2016.08.001.

[103] SmartNews. How SmartNews Built a Lambda Architecture on AWS to Analyze Customer Behavior and Recommend Content | AWS Big Data Blog. = https://aws.amazon.com/blogs/big-data/how-smartnews-built-a-lambda-architecture-on-aws-to-analyze-customer-behavior-and-recommend-content/, 2016.

[104] Ilkka Tuomi. Data Is More than Knowledge: Implications of the Reversed Knowledge Hierarchy for Knowledge Management and Organizational Memory. *http://dx.doi.org/10.1080/07421222.1999.11518258*, 16(3):103–117, 2015. ISSN 07421222. doi: 10.1080/07421222.1999.11518258. URL https://www.tandfonline.com/doi/abs/10.1080/07421222.1999.11518258.

[105] Andrea Vazquez-Ingelmo, Alicia Garcia-Holgado, and Francisco J. Garcia-Penalvo. C4 model in a software engineering subject to ease the comprehension of UML and the software. *IEEE Global Engineering Education Conference, EDUCON*, 2020-April:919–924, 4 2020. ISSN 21659567. doi: 10.1109/EDUCON45650.2020.9125335.

[106] Jon Vexler and Hudi Development Team. Hudi Kafka Connect Sink - GitHub, 2023. URL https://github.com/apache/hudi/blob/master/hudi-kafka-connect/README.md.

[107] Kai Waehner. Kappa Architecture is Mainstream Replacing Lambda - Kai Waehner, 2021. URL https://www.kai-waehner.de/blog/2021/09/23/real-time-kappa-architecture-mainstream-replacing-batch-lambda/.

[108] Jonathan Stuart Ward and Adam Barker. Undefined By Data: A Survey of Big Data Definitions. 9 2013. doi: 10.48550/arxiv.1309.5821. URL https://arxiv.org/abs/1309.5821v1.

[109] Richard Webber. The evolution of direct, data and digital marketing. *Journal of Direct, Data and Digital Marketing Practice*, 14(4):291–309, 4 2013. ISSN 17460166. doi: 10.1057/DDDMP.2013.20.

[110] Wikipedia. 3G - Wikipedia, the Free Encyclopedia, 2023. URL https://en.wikipedia.org/wiki/3G.

Appendices

Appendix A

Requirement Specification

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

José Miguel Dias Simões

Requirement Specification
POWER Data Framework Architecture

Document produced in the scope of the dissertation "POWER Data Framework Architecture", advised by Prof. Bruno Cabral and Prof. Vasco Pereira and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the scope of the "Dissertation/Internship in Software Engineering" curricular unit of the Master's Course in Informatics Engineering, at the Department of Informatics Engineering, in the Faculty of Sciences and Technology of the University of Coimbra, as a part of the POWER Project, requirements were elicited and specified for the design of a software architecture pertaining to a planned infrastructure for a scalable data processing to be implemented at Altice Labs S.A..

This work and the associated dissertation project are funded by the POWER project (grant number POCI-01-0247-FEDER-070365), co-financed by the European Regional Development Fund (FEDER), through Portugal 2020 (PT2020), and by the Competitiveness and Internationalization Operational Programme (COMPETE 2020).

This chapter will outline the purpose of the document as well as its scope, its layout and relevant information (such as conventions, standards, etc) and lastly, all relevant information regarding the intended audience and any pertinent reading suggestions.

1.1 Purpose

This document's main purpose is to serve as a comprehensive description of the main functionalities, features and qualities of the desired software framework. It will serve as a *requirement specification* - detailing all the functional and non-functional requirements of the software system, which provide the foundations for the development of the final architecture.

The processes through which these requirements were elicited, refined and specified is detailed in full in the dissertation report, where the description focuses on the methodologies employed in the requirement engineering process.

In contrast, this document focuses on the description and specification of the system's requirements in a stand-alone perspective, to serve as a resource for the architecture design process.

1.2 Scope

This document aims to list the requirements and constraints that will influence the design of the architecture of a Big Data framework/platform. The data which will be used in these pipelines is variable, from structured log data to streamed usage statistics and analytics, so flexibility is an important trait. From a general point of view, the final architecture aims to serve as the foundation for a scalable, multi-tenant IaaS/PaaS data framework to serve a number of endpoint services based on data exploration and interpretation, within ever-changing regulatory environment characterized by high security and privacy needs.

This framework will receive data from a number of source systems. These source systems are of varying types, and their internal architecture is not considered. It is, however, expected that the platform will be able to handle any kind of input, by way of transforming data to fit a pattern either on the ingestion side or the storage side.

The projected endpoint services for this platform include Business Intelligence (BI)-based services (generation of data-based visualisations, dashboards, reports), Machine-Learning (ML) oriented services (geared toward exploratory and predictive analysis of high-flow data streams) and alarm or trigger-based services (which operate on log data). For the purposes of this specification, the scope of the architectural decisions will lead up to each of these services, but will not encompass the services themselves, as these should have a flexible implementation.

1.3 Document Structure and Conventions

The requirement specification and general document structure will follow the **IEEE 830-1998 Standard** (IEEE Recommended Practice for Software Requirements Specifications). This standard provides a structure which may be followed to create a fully encompassing requirement specification, as well as some general guidelines for describing requirements.

The contents will begin with a description of the system from a functional and qualitative standpoint, describing how it will be broken down into logical components for the requirement specification (in this case, layers), and presenting the main functionalities of each component, as well as the main qualities associated with it. Following this description, the requirements will be presented (functional and non-functional)

1.4 Intended Audience and Reading Suggestions

While the software requirement specification document is written for a more general audience, this document is intended for individuals directly involved in the development of the **POWER Data Framework** within Altice Labs S.A. and the corresponding dissertation project within the University of Coimbra. Below is an overview of the structure of the document, along with additional information regarding the contents and layout of each Section.

- **Chapter I: Introduction** - In this section, the basic introductory information is exposed, along with the scope of the document and some information regarding the project.
- **Chapter II: System Description** - In this section there is a description of the layout of the system and its five main components (layers).
- **Chapter III: Constraints** - In this section there is a description of the business and technical constraints, separate from the requirements.
- **Chapter IV: Requirements** - In this section, the functional and non-functional requirements of the system and of each individual layer are detailed. Accompanying each layer's requirements is a brief description of its functionalities and expectations.

The requirements and all tabular entries in this document all have a Source ID. In Table 1.1, each ID is described, with information regarding where the requirement originated.

Source ID	**Description**
(*)	Identifies a correction/revision.
PRE	Preliminary elicitation (ADEW).
REQ-0	Preliminary refinement/gap-filling for draft.
REQ-1	Requirements pertaining to the first iteration's refinement.
REQ-2	Requirements pertaining to the second iteration's refinement.

Table 1.1: Source IDs for each of the requirements.

For the analysis of the requirements, some software/requirements engineering concepts are relevant. They are presented below:

- **Constraints** - Fixed restrictions of technical and regulatory nature that must be taken into account during architecture design.
  - **Technical Constraints** typically relate to restrictions imposed by the legacy software which will be connected to the system, or by pre-selected components that the client specifically wants to use.
  - **Business Constraints** relate to the business activity of the client, and must be accounted for and met through correct specification.
- **Functional Requirement (FR)** - A functional requirement is a specific and measurable statement that describes what a component is expected to do, such as a feature, behavior, or task that must be performed by the system to achieve its intended purpose.
- **Non-Functional Requirement (NFR) or Quality Attribute (QA)** - A non-functional requirement is a quality or characteristic that describes how a component should perform or behave, using categories such as reliability, usability, scalability, or security. Generally these requirements describe the qualities of the functionalities and interactions within the system.
- **Quality Attribute Scenario (QAS)** - A six-part quality attribute scenario is a structured format used to describe a QA, which includes six key elements: *stimulus*, *source*, *environment*, *artifact*, *response*, and *response measure*, to describe a scenario with which to verify whether or not a component meets the QA.

Chapter 2

System Description

The target system consists of a centralized data processing platform designed for multi-tenant IaaS/PaaS[1] operational environments. This platform aims to bring together various services within Altice Labs S.A. while also facilitating regulatory compliance. Additionally, it is intended to support future initiatives related to data privacy, security, and governance by offering appropriate tools and facilitate the adoption of organizational changes.

To better organize and structure the requirement specification process, it is necessary to create a well-structured and organized layout, splitting the highly complex and large target architecture into more manageable parts, enabling the isolated analysis of the main "building blocks" of the proposed system.

Each of these components, or "layers" has its own distinct characterization, encapsulating vastly different requirements and needs. As such, it is necessary to formalize the partitioning, define clear boundaries between these parts and proceed with the analysis of each of them in separate, and in a system where they work together to ensure that all identified requirements are met.

In this chapter, the system's architecture is described along with its partitioning into *five* layers: **ingestion**, **storage**, **serving**, and **orchestration & administration**. Each layer has a specific role and objective, and together they enable the system to perform its main functionalities, and support all quality attributes (such as performance, scalability, etc.)

For each layer, a brief overview is provided, describing its main functionalities and motivations, highlighting the key goals that it aims to serve, and the main drivers for its architectural development.

[1]Infrastructure/Platform-as-a-service - operational contexts in which the company (in this case, Altice Labs, supplies either the execution infrastructure or a platform as a service to external parties for their own data management businesses.

2.1 Overview

The system's main objective is to enable the processing of *large quantities of data* in a **multi-tenant** capable, highly-**scalable** and **available** context. Data is received from source systems, with varying types, sizes and formats, and is passed on to storage systems, for later use by the system's various business endpoints (e.g. data analytics, visualization, ML, etc.). To manage this entire process, an orchestration dashboard or control panel monitors the health of the system and handles the logging and configuration of the various components.

Throughout this "functionality" chain, there must be provisions for data quality monitoring, traceability and auditability, as well as considerations for infrastructure monitoring, automation and system health checking. These features will allow the system to adapt to the multi-tenancy context, involving external companies and clients in a compliant manner (especially with regards to data privacy regulations) through strict access control, domain *lock-out*/management and auditability.

2.1.1 Architecture Partitioning

The architecture is partitioned into three views - Functional, Administrative and Orchestration. These views encompass the system's full characterization, and can be summarised as:

- **Functional View** - Those pertaining to the main intended functionality - data processing. This includes a further subdivision:
  - **Ingestion** - The entry of new data into the system.
  - **Storage** - The storage and management of large data volumes.
  - **Serving** - The presentation and consumption of data by external services.
- **Administrative View** - Pertaining to the regulatory constraints and encompassing the components which enable compliance.
- **Orchestration View** - Encompassing the components responsible for monitoring, automation and alerting of the system's execution.

So, to describe the system, a **five-layer** partitioning is performed: *Ingestion Layer*, *Storage Layer*, *Serving Layer*, *Administration Layer* and *Orchestration Layer*, following the aforementioned perspectives. Each of the following sections will target one of these layers, providing a functional view of which tasks and goals the layer is focused on.

2.2 Functional Layers

As previously discussed, this perspective encompasses three layers - the **Ingestion Layer (IL)**, **Storage Layer (SL)** and **Serving Layer (SV)**. These layers are where the majority of the system's data flows will occur, bringing data in from legacy systems and new sources, transforming it, storing and then delivering it to its intended endpoint. Figure 2.1 presents a simplified view of the Functional View, with its main tasks highlighted, layer by layer.

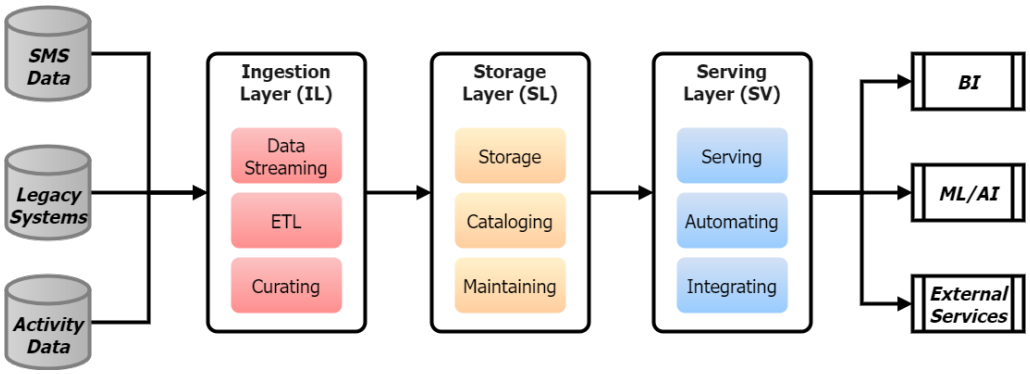

Figure 2.1: Schematic representation of the functional view of the architecture. Data flows are represented by arrows, and within each layer the main functionalities are identified.

Data which is streamed in from external sources is passed through the IL, receiving optional transformations and possibly even being curated based on its quality or other metrics.

Afterwards, the data is passed to the SL, where storage will take place. This storage should include some form of cataloguing as per the Lakehouse pattern, to allow for mixed data storage with warehouse-level performance.

Finally, in the SV, data is pulled from storage. This layer supports automation of pulling and querying, and provides the necessary tools for data integration, connecting the data to external services and data sharing. The SV will also encompass access control requirements, by virtue of being the external interface of the system.

The main drivers of these layers can be identified as:

- **Ingestion Layer** - *Performance* and *Scalability* in the data streaming operations to handle the variable loads and peaks of daily operations, especially in the multi-tenant environment.
- **Storage Layer** - *Availability* and *Integrity* of the storage systems, and the ability to ensure that data is never compromised even under hardware failure.
- **Serving Layer** - *Security* and *Traceability* of user actions, to create a stable and compliant data-sharing environment.

2.3 Administration Layer

On top of these functional components, the **Administration Layer (AL)** serves as an all-encompassing set of components that will enable the management of data throughout the entire lifecycle. Figure 2.2 provides a simplified look at the main tasks supported by the AL.

By leveraging modern technologies such as AI, Knowledge Graphs and more, it is possible to use metadata derived from all the layers of the system to build a characterization of the many data flows for auditability and traceability-related tasks.

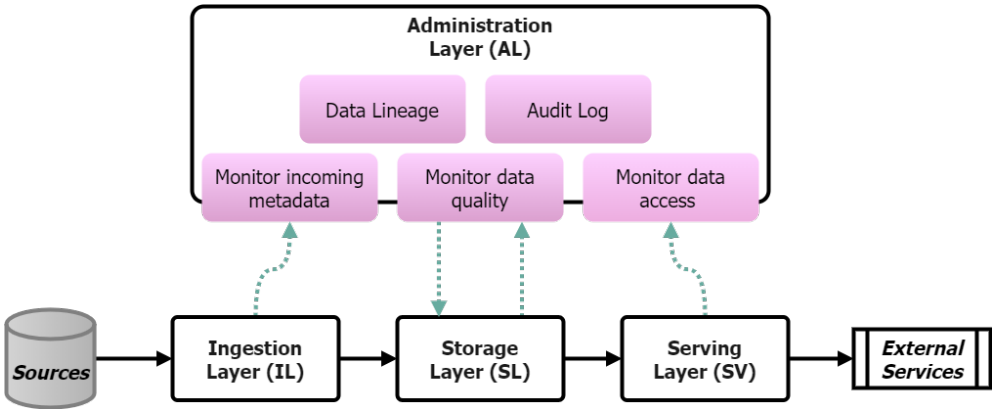

Figure 2.2: Schematic representation of the administrative view of the architecture. Metadata flows are represented with dotted green arrows. The data sources and endpoint external services were joined into a single component for this view.

Through these functionalities, it is possible to maintain an extensive audit log and use it to ensure compliance, as well as assist in tasks related to data quality and understanding how data changes throughout its lifecycle by monitoring all changes made to data objects and integrating them into a comprehensive lineage view.

The main drivers for this layer are **Isolation**, **Privacy** and **Usability**:

- **Isolation and Privacy** - In ensuring that data is not widely available to anyone who seeks it, and that each domain/team/user has their own set of data and access is controlled.
- **Usability** - In creating an environment that can facilitate integration, data sharing and inter-operability between domains in a safe, compliant way.

2.4 Orchestration Layer

In parallel with the management of the data, there must also be a feature-set for managing the system's execution, and to ensure that scalability, performance and productivity does not suffer during the various states of operation the system may be subjected to.

In the **Orchestration Layer (OL)** components work together to ensure the system's health, through a combination of logging, log analytics and automated maintenance routines. Figure 2.3 provides a simplified view of the main activities that take place in the OL.

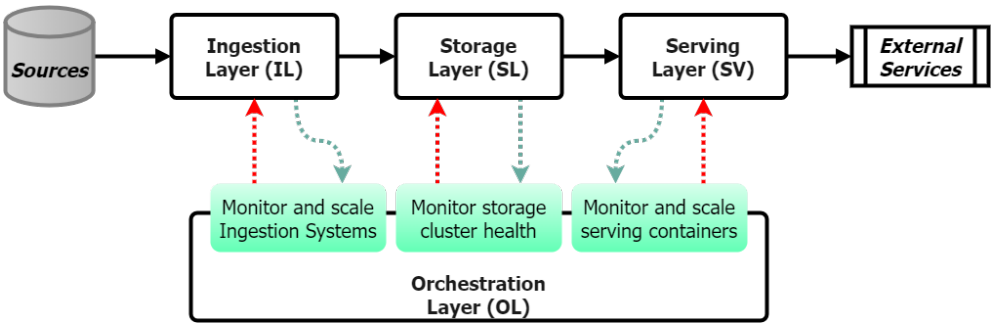

Figure 2.3: Schematic representation of the functional view of the architecture. Data flows are represented by arrows, and within each layer the main functionalities are identified.

To support these functionalities, distinct message queues/pathways must be defined, to ensure long-term scalability and flexibility of the monitoring and management solution. The potential for this layer is significant, since it may use its role as overseer to significantly automate the management and health checking of the system through the configuration of alarms, automated actions and reactions to system events. Taking this into account, the key drivers for this layer are **Consistency**, **Modifiability** and **Scalability/Performance**:

- **Consistency** - In logging, monitoring and maintaining the harmony of the system through constant, real-time analysis of system health data.
- **Modifiability** - In adding new monitoring components, new analytics operations, action automation and even registering new components into the monitoring suite. These needs must be met, allowing for a resource-conservative approach to management and maintenance.
- **Scalability/Performance** - In ensuring harmonious operation at all load levels, the monitoring, rate-limiting and automation must be able to perform, and scale up/down whenever necessary, maintaining an elastic profile.

Chapter 3

Constraints

This chapter seeks to detail the constraints to which the project as a whole is subjected too, and that must be considered when designing the architecture. These constraints include business-related limitations (such as regulatory constraints, limits inherent to the business activities of the company, etc.) and more technical-related limitations (such as integration with existing technology, on-prem hardware, etc.).

The mapping of these constraints will give more perspective to the design process by defining the rigid and soft limits in the design decisions which will be made in the architecture design process.

This chapter presents both business and technical constraints. They are presented in a tabular format, including an ID, a Category, a Description and a Source-ID (mapped previously in Chapter 1, Section 1.4 - Intended Audience and Reading Suggestions).

3.1 Business Constraints

The business constraints identified for this project can be summarised as the normal limitations attached to a data management platform (i.e. regulatory compliance, GDPR requirements, etc.) paired with the need to accommodate a growing, scalable business model and all its requirements (cost control, client management, privacy, etc).

Table 3.1 identified the main business constraints identified thus far for the project. As with other requirements/constraints in this document, an **ID-Description** pair is used for each entry. The business constraints here also have a **Category** identifier, which gives some general information as to which specific topic the constraint refers to.

ID	Source	Category	Description
BC-001	PRE	General	The system must evolve into a multi-tenant cloud-based architecture, with one deployment for several client provider.
BC-002	PRE	Privacy	The mandatory security requirements of the GDPR are fully met.
BC-003	PRE	Costs	Software components included in the architecture should prioritize accessibility and focus on low-cost alternatives (free-to-use being especially desirable)
BC-004	PRE*	Costs	Costs should be controllable, either by limiting resource usage or by creating cost ceilings.
BC-005	PRE	Training	Barrier of entry for software component usage/modification should be as low as possible, with good documentation and support.
BC-006	PRE	Costs	Software components should, ideally, use open source software.
BC-007	REQ-1	Privacy	The software manages personal and non-personal data in separate ingestion, storage and serving streams.
BC-008	REQ-1	Privacy	Personal data must be anonymized through Generalisation, Randomisation or Masking.
BC-009	REQ-1	Privacy	Personal data must not be kept for a period longer than useful or necessary for business operations.

Table 3.1: Business constraints identified for the project.

3.2 Technical Constraints

The currently identified technical constraints mostly concern the information and descriptions gathered during the preliminary "Architectural Driver Elicitation Workshop", prior to the dissertation project. As such, the contents of the technical constraints record (Table 3.2) mostly consist of requirements presented from the analysis of existing systems (namely the connections to legacy data sources and existing BI or DS, AI or ML services).

While these requirements may not be directly applicable, they provide some idea of what legacy and existing components may need to be connected to the platform, and as such, represent a base for compatibility-based requirements.

ID	Source	Description
TC-001	PRE	SMS/Phone data is accessed by Cube Navigator/MicroStrategy
TC-002	PRE	SMS/Phone data is stored in Greenplum
TC-003	PRE	SMS/Phone data is accessible via REST API and MongoDB
TC-004	PRE	SMS/Phone data is monitored via Prometheus Alertmanager
TC-005	PRE	BI data is received from CSV and Oracle databases
TC-006	PRE	BI data must be available via SQL interface
TC-008	PRE	Internal clients (helpdesk, etc) receive data from a Kafka MQ
TC-009	REQ-2	Components (general) are compatible with the Prometheus monitoring toolset.

Table 3.2: Technical constraints identified for the project.

Chapter 4

Requirements

This chapter outlines the functional requirements and non-functional requirements of the system, layer by layer. As previously detailed, the architecture has been partitioned into **five** layers (IL, SL, SV, OL and AL), and this chapter will detail all the requirements associated with each of these logical components.

Functional requirements describe what the system must do to fulfill its purpose and meet the needs of its users and stakeholders. They specify the features, capabilities, and behaviors of the system in terms of inputs, processing, outputs and user interaction, providing a detailed description of the system's expected functionality.

Adding to this description, non-functional requirements describe the characteristics and qualities of the system which are crucial to its overall performance, usability, reliability, scalability, etc. These provide a set of criteria against which the system's success can be measured and help ensure that the system meets the needs and expectations of its stakeholders, while taking the various constraints into consideration.

Firstly, the general requirements will be outlined. These encompass the globality of the system and are applicable to all components. After this general view, each of the layers' requirements are detailed separately, in order to provide a comprehensive characterization of each component.

4.1 General View

From a general perspective, the system's requirements are defined loosely, but give some indication on how the system's components should generally function in different settings.

Functionally (Table 4.1), integration is identified as a major factor. The ability to seamlessly integrate new technologies and data streams is considered due to the open-ended nature of the multi-tenancy context which the framework is intended to serve. Adding to this, the matter of functionally implementing scalable design is also considered, as the system must possess the ability to change dynamically and monitor itself to ensure its adaptability to adverse conditions.

In terms of qualities (Table 4.2, the system has a major focus on the usual factors that drive distributed systems: Availability, Reliability, Performance and Scalability. Communication between components, general availability metrics and performance requirements can be defined (albeit loosely), and some general scalability requirements can be defined as significant for all components.

ID	Source	Description
FR-G-001	REQ-1	The administrator can integrate new data streams into the system.
FR-G-002	REQ-1	The system's components must dynamically scale to maintain throughput/other specific production metrics.
FR-G-003	REQ-0	Components collect and internal health checking statistics and data and push it to the CL.

Table 4.1: Functional Requirements for system's components, in a general view.

ID	Source	Category	Description
NFR-G-001	REQ-0	Availability	The system remains online during maintenance or update.
NFR-G-002	REQ-1	Reliability	The system must guarantee that each message is delivered exactly-once, regardless of any adverse conditions (in both data and logging queues).
NFR-G-003	REQ-0	Performance	Peaks of data production and consumption are typical, and the system remains in operation and does not lose data or functionality when peak conditions are met
NFR-G-004	REQ-1	Scalability	The system must instantiate new components or shelf inactive components to maintain steady operation under all conditions
NFR-G-005	REQ-1	Scalability	The system's scalability adjustments must never allow for throughput loss greater than 5%.

Table 4.2: Non-Functional Requirements (Quality Attributes) for the general view.

4.2 Ingestion Layer

As the entry point of data, requirements are more strictly defined, including an array of functionalities pertaining to the connection of new data sources (and the adaptability of the system to varying source data flows), and qualities mainly linked to performance, scalability and reliability (to ensure that data streams and flows are not disrupted in the production environment.

Due to the multi-tenant target for the system, performance qualities must be defined in a scalable way, to ensure that tests can be performed prior to the actual system's production stage (and prior to the concurrent use by various tenants).

4.2.1 Functional Requirements

The functional requirements of the IL focus on the ability to configure the system's data streams through scheduling, duplicating, automating and readily modifying existing streams. By meeting these requirements, the cost of connecting a new data-source or forwarding data to a different place is vastly reduced, resulting in a much more flexible system.

ID	Actor	Source	Description
FR-IL-001	System	REQ-1	The system logs new component additions/connections through an API
FR-IL-002	System	PRE	The system shall accept configurations for both streaming and batch data sources.
FR-IL-003	Process Manager	REQ-1	The process manager can duplicate data streams and forward them to the Serving Layer (SV) through a "speed-layer"
FR-IL-004	Process Manager	REQ-1	The process manager can connect new data sources to the IL's queues
FR-IL-005	Process Manager	REQ-1	The process manager can configure automated pulling from external data sources
FR-IL-006	Process Manager	REQ-1	The process manager can configure scheduled continuous data acquisition (data streaming windows)
FR-IL-007	Process Manager	REQ-1	The process manager can automate actions/ETL/transformations using a scheduling engine
FR-IL-008	Data Manager	REQ-1	The data manager can interact with the scheduling engine to ensure correct integration/enrichment for the "enrichment lane".

Table 4.3: Functional Requirements for Ingestion Layer (IL).

4.2.2 Non-Functional Requirements

This flexibility is complemented by the identified quality-attributes which the components of this layer must hold: **Scalability**, **Availability**, **Performance**, **Integrity** and **Compatibility**.

These are the main traits of this layer's components, and result in a fast, reliable system for moving data into the framework without faults, problems or delays.

QAS-IL-001 - *REQ-1* - Scalability

Source - Source Systems
Stimulus - The number of files streaming into the system increases to peak conditions.
Environment - At runtime, under peak load.
Artifact - Ingestion Layer

Response - Throughput is not severely affected and system operations continue seamlessly.
Response Measure: No throughput loss rate greater than 1% is observed in the ingestion queues as they adapt to the increased load.

QAS-IL-002 - *REQ-1* - Performance

Source - Source Systems
Stimulus - Files are streaming into the system.
Environment - At runtime, under normal operations.
Artifact - Ingestion Layer

Response - Latency is low and files are streamed with minimal overhead.
Response Measure: Latency is not greater than 100ms in message passing operations.

QAS-IL-003 - *REQ-1* - Availability

Source - Ingestion Queue
Stimulus - A cluster of message brokers enters failure state.
Environment - At runtime, under normal operations.
Artifact - Ingestion Layer

Response - Process Manager is notified, new components are instantiated to serve existing streams.
Response Measure: The system does not incur in significant downtime during cluster failure.

QAS-IL-004 - *REQ-1* - Compatibility

Source - Process Manager
Stimulus - A new data source is connected to the system.
Environment - At configure time.
Artifact - Ingestion Layer New source is connected and becomes operational, streaming data into the system.

Response - The source is fully operational and integrated within 1 hour of the change.
Response Measure:

QAS-IL-005 - *REQ-1* - Integrity

Source - Source Systems
Stimulus - Files are streaming into the system.
Environment - At runtime, under normal operations.
Artifact - Ingestion Layer

Response - File corruptions or errors are corrected by the system, and data is verified before being stored.
Response Measure: The system detects message errors 99.99999% of the time.

QAS-IL-006 - *REQ-1* - Reliability

Source - Ingestion Queue
Stimulus - Message transmission isn't successful.
Environment - At runtime, under component failure.
Artifact - Ingestion Layer

Response - Messages are not lost, and are re-transmitted upon service recovery using "E-O" message delivery protocols.
Response Measure: The system has a message loss rate of less than 0.1%.

4.3 Storage Layer

In the Storage Layer (SL), requirements focus on the connections to the other layers and on internal operations regarding the safe, efficient and performant storage of large quantities of both personal and non-personal data.

4.3.1 Functional Requirements

The requirements of the SL focus on features necessary to create a scalable and reliable data store. The ability to snapshot data, organize metadata and even provide transformations and ETL on incoming and outgoing data-streams give ample tools for data management.

ID	Actor	Source	Description
FR-SL-001	System	REQ-0	The system shall provide a functionality to connect incoming data streams to storage.
FR-SL-002	System	REQ-0	The system shall provide optional data transformation capabilities for incoming data streams.
FR-SL-003	System	REQ-1	The system shall provide a functionality to create snapshots of data on-demand.
FR-SL-004	System	REQ-1	The system shall provide the ability to extract aggregated data snapshots.
FR-SL-005	System	REQ-1	The system shall provide a functionality to visualize aggregated data on-demand.
FR-SL-006	System	REQ-1	The system shall provide the ability to store data for a specified period of time.
FR-SL-007	System	REQ-1	The system shall allow the storage of structured, semi-structured and unstructured data.
FR-SL-008	System	REQ-1	The system shall enable scheduling of storage maintenance operations (de-duplication, backup, etc.)
FR-SL-009	System	REQ-1	The system shall allow the configuration of a metadata catalog to virtualise data access.
FR-SL-011	System	REQ-1	The system shall autonomously build categorizations based on incoming data types.
FR-SL-012	Data Manager	REQ-1	A DATA MANAGER may configure custom categorizations for incoming data.
FR-SL-013	Data Manager	REQ-1	A DATA MANAGER may define "lifespans" for data categories.
FR-SL-014	Process Manager	REQ-1	A PROCESS MANAGER may configure new storage components within the SL (databases, "bins", etc)

Table 4.4: Functional Requirements for Storage Layer (SL).

4.3.2 Non-Functional Requirements

Non-functional requirements for the SL focus on **Availability**, **Integrity**, **Security** and **Reliability**, as all the storage system's data must be backed-up, secured and highly available to ensure harmony within the system. To express these qualities, the scenarios were designed around the use of the SL's features, by the Users and by the Data Manager.

QAS-SL-001 - *REQ-1* - Usability

Source - End-user
Stimulus - A user wishes to access a dataset pertaining to his domain.
Environment - At runtime, under normal operations
Artifact - Storage Layer

Response - Data is retrieved and presented to the user.
Response Measure: Data is prepared according to the access level of the user and presents only the data to which he has access to, omitting fields/rows which he is not allowed to see.

QAS-SL-002 - *REQ-1* - Integrity

Source - Data Manager
Stimulus - The manager seeks to ensure that data storage is backed-up and fault-tolerant
Environment - At runtime, under normal operations
Artifact - Storage Layer

Response - The system ensures storage replication, backups and fault-recovery.
Response Measure: Data records are resilient to the loss of their main storage component. Backups are performed daily.

QAS-SL-003 - *REQ-1* - Security

Source - End-user
Stimulus - The user seeks to access data outside of his domain
Environment - At runtime, under normal operations
Artifact - Storage Layer

Response - Data access is denied, user access is logged.
Response Measure: Users are not able to access restricted data outside of their domain.

QAS-SL-004 - *REQ-1* - Auditability

Source - Data Manager
Stimulus - Desire to check the lineage of a dataset
Environment - At runtime, under normal operations
Artifact - Storage Layer

Response - The history of the dataset is displayed, from creation.
Response Measure: All intermediate transformations and steps are recorded.

QAS-SL-005 - *REQ-1* - Reliability

Source - Storage Components
Stimulus - One of the storage components ceases to function and communicate.
Environment - At runtime, under failure conditions.
Artifact - Storage Layer

Response - Failure is detected, operations are handed over to replicated backups. Faulty service is restored.
Response Measure: Data must never be lost, either through image backups or distributed replicas, and the original storage component must be rebuilt and repopulated fully within 1 day.

4.4 Serving Layer

The Serving Layer, as the point of interfacing between the system and the end-users and external services, has a lot of functionalities related to the extraction, presentation and handling of stored data. Qualities are mostly directed at the traits of these connections, namely in terms of how data is presented to external services.

4.4.1 Functional Requirements

The functional requirements of the SV focus on convenience features such as automation, query flexibility (i.e. perform ad-hoc queries) and data sharing features to fulfil the more "Data-Mesh" oriented use-cases, enabling end-users to share their data with other clients through a number of filters and privacy-preserving techniques, and disabling access to unauthorized users outside of the relevant domain.

ID	Actor	Source	Description
FR-SV-001	System	REQ-1	The system shall allow for the automation of data transit (pushing, pulling) for external services
FR-SV-003	System	PRE	The system can perform preset or adhoc queries on the contents of its internal catalog.
FR-SV-004	System	REQ-1	The system shall enable the creation of REST APIs to relay data from the Storage Layer without the use of a message queue.
FR-SV-005	System	PRE	The end-user may configure automated data pulling for ML/AI/DS tasks.
FR-SV-007	Data Manager	REQ-1	The data manager can configure data transit for new external services.
FR-SV-008	System	PRE	The system shall present specific views of the data in the SL according to the user's access level/domain.
FR-SV-009	System	REQ-1	The system shall regulate access to the specific catalog through increasingly granular interfaces
FR-SV-010	End-User	REQ-1	The end-user may access the data views which pertain to his domain
FR-SV-011	Data Manager	REQ-1	The data-manager may configure rules to allow data sharing between domains.
FR-SV-013	Data Manager	PRE	The data manager may connect to the IL's queue to set-up an "enrichment lane" for data integration/enrichment

Table 4.5: Functional Requirements for Serving Layer (SV).

4.4.2 Non-Functional Requirements

Non-functional requirements for this layer focus on the qualities of **Performance**, for high speed querying and data fetching/pulling, and in **Security**, **Traceability** and **Consistency**, to ensure that user access is done according to the relevant rules and domain lock-outs, and that all user accesses can be traced for auditing purposes.

QAS-SV-001 - *REQ-1* - Compatibility

Source - End-user
Stimulus - A user wishes to connect an external analytics engine to an existing data view.
Environment - At runtime, under normal operation
Artifact - Serving Layer

Response - The system allows for the automation of data transformations and subsequent pushing to the external analytics tool.
Response Measure: The system can adjust the data type/schema to a target type using data transformations. These transformations are explictly programmed.

QAS-SV-002 - *REQ-2* - Performance

Source - End-user
Stimulus - A user executes a query for a sub-selection of a dataset.
Environment - At runtime, under normal operation
Artifact - Serving Layer

Response - The system presents the data.
Response Measure: Queries are executed in real-time with minimal latency (<100ms).

QAS-SV-003 - *REQ-2* - Security

Source - Data Manager
Stimulus - The data manager wishes to configure new data views for a tenant's interface.
Environment - At runtime, under normal operation
Artifact - Tenant Interface

Response - The new views are available to the relevant end-users/domains/tenant.
Response Measure: Propagation of new views takes at most 1 hour.

QAS-SV-004 - *REQ-2* - Traceability

Source - End-user
Stimulus - A user executes an operation (read or write)
Environment - At runtime, under normal operation
Artifact - Tenant Interface

Response - The action is recorded in a log.
Response Measure: The log is immutable and replicated. This log is protected from all user interference and stored outside of normal user access.

QAS-SV-005 - *REQ-1* - Consistency

Source - Data Manager
Stimulus - The data manager automates pushing of data to external services for periodic data analytics/ML/etc.
Environment - At runtime, under normal operation
Artifact - Serving Layer

Response - Data pulling from SL and pushing to external services is automated and executed periodically.
Response Measure: Deviation from scheduled action time is not greater than 1 minute. Operations are executed consistently according to their defined rate (i.e. if hourly, 24 times per day).

4.5 Orchestration Layer

Features like this demand a strict logging system, ensuring realtime data flows into an analytics engine which can feasibly respond to the changing conditions of the system. To this end, a connection to the outside (dashboard) will enable a "Process Manager" to configure and prepare the system for its daily operations, be it through automating maintenance, analysing logs manually or pushing new configurations on-the-fly, to effect updates and ensure connectivity throughout the entire system.

4.5.1 Functional Requirements

Functional requirements for this layer generally revolve around the monitoring of the many components, the health checking and associated analytics and metrics. With this information, the process manager can automate tasks and backups, set up alarms, triggers and actions to ensure the harmonious operation of the framework. Additionally, some consideration is allotted to disaster recovery, ensuring that backups occur and that recovery is possible.

ID	Actor	Source	Description
FR-OL-001	System	REQ-1	The system shall allow for the connection of new modules to the monitoring system
FR-OL-003	System	REQ-1	The system shall support the building of images/snapshots of log data and save them to LTS.
FR-OL-004	System	REQ-1	The system shall be configured to perform load balancing.
FR-OL-005	System	REQ-1	The system performs the automatic creation of execution snapshots or checkpoints to enable recovery after catastrophic failure.
FR-OL-007	Process Manager	REQ-1	The process manager can access platform health data, status information and component metrics.
FR-OL-008	Process Manager	REQ-1	The process manager can view and run analytics on log data.
FR-OL-009	Process Manager	REQ-1	The process manager can configure alarms, triggers and actions, and automate them via connection to specific component(s).
FR-OL-010	Process Manager	REQ-1	The process manager can configure scaling profiles (rules for provisioning and orchestration) and automate their implementation,

Table 4.6: Functional Requirements for Orchestration Layer (OL).

4.5.2 Non-Functional Requirements

The non-functional requirements of this layer revolver generally around **Scalability**, **Performance** and **Availability**, key factors in the system's response to changes in the operating conditions, as well as matters of **Consistency** in how often the system should execute its backup operations.

QAS-OL-001 - *REQ-1* - Scalability

Source - System
Stimulus - Resources are insufficient to ensure harmony between producers and consumers.
Environment - At runtime, under high demand.
Artifact - Orchestration Layer

Response - The system scales horizontally to meet demands by executing automated scaling processes.
Response Measure: No throughput loss rate greater than 5% is observed in any component.

QAS-OL-002 - *REQ-2* - Performance

Source - System Component
Stimulus - Sends log data in real-time.
Environment - At runtime, under normal operations.
Artifact - Logging Component

Response - The logs are processed and eventually saved to LTS.
Response Measure: Logs are processed in real-time. Latency is no greater than 10ms.

QAS-OL-005 - *REQ-1* - Consistency

Source - Process Manager
Stimulus - The process manager wishes to configure automated snapshotting for the system health logs.
Environment - At configure time.
Artifact - Orchestration Layer

Response - The logs are saved to LTS via snapshotting at configured time intervals.
Response Measure: Discrepancies of time between snapshots are no greater than 0.1%.

QAS-OL-006 - *REQ-2* - Availability

Source - System Component
Stimulus - A component must be updated
Environment - At runtime, during maintenance operations.
Artifact - Component

Response - The component is updated with no downtime, and its operations are not interrupted.
Response Measure: The system ensures that during component update, its operations are ensured by either replication or load balancing (i.e.e green-blue).

QAS-OL-007 - *REQ-2* - Modifiability

Source - Process Manager
Stimulus - The process manager seeks to automate a new analytics chain on log data from Component X
Environment - At runtime, under normal operations.
Artifact - Log Analytics Engine

Response - The new processing workflow is added to the engine and begins execution.
Response Measure: The integration of this new analytics chain and subsequent presentation of results takes no longer than 30 minutes.

4.6 Administration Layer

This layer will rely on a "Data Manager" to control who has access to the data, how they can view it, and how it can be shared between users. Also available will be a comprehensive logging of all user activity in relation to the data, and the ability to track all interactions in an immutable log to ensure compliance with relevant regulations. This layer is deeply entwined with the Serving Layer, as this is where the user's behaviour will be observable.

4.6.1 Functional Requirements

Functional requirements for this layer include the management of teams, domains and users and also the logging of user activity. This management is done also on the level of maintaining an external **data catalog**, which serves as a ledger for all the operations and metadata attached to the data within the system. This catalog may also be used to initiate data sharing contracts with external clients, by allowing them to browse the catalog and see, via the metadata, what data may be of use to them, without compromising on its privacy and security.

ID	Actor	Source	Description
FR-AL-001	Data Manager	REQ-1	The data manager can perform access control operations, defining who can access which domains through an IMS (Identity Management System)
FR-AL-002	Data Manager	REQ-2	The data manager can associate entities (domains, tenants, user groups) with catalog views.
FR-AL-003	Data Manager	REQ-2	The data manager can extract user interaction logs for analytics.
FR-AL-004	Data Manager	REQ-2	The data manager can configure new user interaction logging processes via APIs.
FR-AL-005	System	REQ-1	The system shall perform global user interaction monitoring using daily logs pertaining to user actions (read, write, modify)
FR-AL-006	System	REQ-1	The system shall snapshot logs on a configurable basis and save them to LTS for auditability purposes.
FR-AL-007	System	REQ-1	The AL is connected to all components and access logs are communicated through APIs integrated into these components.

Table 4.7: Functional Requirements for Administration Layer (AL).

4.6.2 Non-Functional Requirements

Non-functional requirements of this layer involve the maintenance of **Security**, **Privacy** and **Isolation**, key qualities of a compliant system, and with some interest put on issues of **Usability** and **Consistency** to ensure that operating this administrative environment is easy, and that its effects on the system are felt in a consistent way.

QAS-AL-001 - *REQ-1* - Usability

Source - Data Manager
Stimulus - New data views must be configured for Tenant X
Environment - At configure time.
Artifact - Metadata Catalog

Response - A new view is created, and is readily accessible/configurable for Tenant X.
Response Measure: New view is visible within 1 hour of configuration.

QAS-AL-002 - *REQ-2* - Isolation

Source - Unauthorized User
Stimulus - An unauthorised user attempts to access data outside his domain
Environment - At runtime, under normal operations.
Artifact - Tenant Interface

Response - Access is not granted, activity is logged.
Response Measure: Data is not accessible between domains unless explicitly programmed so.

QAS-AL-003 - *REQ-2* - Privacy

Source - Tenant X
Stimulus - Tenant X wishes to create a self-serve view for his data.
Environment - At configure time.
Artifact - Metadata Catalog

Response - The view is created and made available to other tenants. Access is granted by Tenant X or the Data Manager.
Response Measure: Data presented in these self-serve views is updated in real-time when the source data changes.

QAS-AL-004 - *REQ-2* - Consistency

Source - Data Manager
Stimulus - The data manager wishes to configure automated backups of the access log.
Environment - At configure time.
Artifact - Metadata Catalog

Response - The log is automated to be saved to LTS and backed up on a periodic schedule.
Response Measure: Deviations from scheduled time between logs are no greater than 0.1%.

QAS-AL-005 - *REQ-2* - Privacy

Source - Tenant X
Stimulus - Tenant X wishes to access data from Tenant Y's views
Environment - At runtime, under normal operations.
Artifact - Metadata Catalog

Response - If Tenant Y has created self-serve views, these may be used. If not, a request is made for the Data Manager to enable access.
Response Measure: Tenant X cannot access Tenant Y's data beyond the views which Tenant Y provides (i.e. if Tenant Y does not enable the viewing of the first column of a DB, Tenant X will receive that Db without the info on the first column).

Appendix B

Architecture Specification

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

José Miguel Dias Simões

Architecture Specification
POWER Data Framework Architecture

Document produced in the scope of the dissertation "POWER Data Framework Architecture", advised by Prof. Bruno Cabral and Prof. Vasco Pereira and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2023

Contents

List of Figures

Chapter 1

Introduction

In the scope of the "Dissertation/Internship in Software Engineering" curricular unit of the Master's Course in Informatics Engineering, at the Department of Informatics Engineering, in the Faculty of Sciences and Technology of the University of Coimbra, as a part of the POWER Project, this document serves as the final architecture specification for the Data Framework dissertation project, pertaining to a planned infrastructure for scalable data processing to be implemented at Altice Labs S.A..

This work and the associated dissertation project are funded by the POWER project (grant number POCI-01-0247-FEDER-070365), co-financed by the European Regional Development Fund (FEDER), through Portugal 2020 (PT2020), and by the Competitiveness and Internationalization Operational Programme (COMPETE 2020).

This chapter will outline the purpose of the document as well as its scope, its layout and relevant information (such as conventions, standards, etc) and lastly, all relevant information regarding the intended audience and any pertinent reading suggestions.

1.1 Purpose

This document's purpose is to describe the layout, components and connections of the software architecture for the target framework. It will serve as an **architecture specification**, presenting the inner workings of the framework.

This architecture was designed following a formal architecture development methodology, and this process is detailed in full in the dissertation report, where the description is aimed primarily at describing how the architecture was designed and how components were selected based on research, experimentation and an iterative design approach.

In contrast, this document focuses on the description and specification of the system's architecture in a stand-alone perspective, to serve as a resource for future work and implementation efforts.

1.2 Scope

This document aims to describe the architecture using an architecture description language, through diagrams and brief text summaries. From a general point of view, the final architecture aims to serve as the foundation for a scalable, multi-tenant IaaS/PaaS data framework to serve a number of endpoint services based on data exploration and interpretation, within ever-changing regulatory environment characterized by high security and privacy needs.

This framework will receive data from a number of source systems. These source systems are of varying types, and their internal architecture is not considered. It is, however, expected that the platform will be able to handle any kind of input, by way of transforming data to fit a pattern either on the ingestion side or the storage side.

The projected endpoint services for this platform include Business Intelligence (BI)-based services (generation of data-based visualisations, dashboards, reports), Machine-Learning (ML) oriented services (geared toward exploratory and predictive analysis of high-flow data streams) and alarm or trigger-based services (which operate on log data). For the purposes of this specification, the scope of the architectural decisions will lead up to each of these services, but will not encompass the services themselves, as these should have a flexible implementation.

1.3 Document Structure and Conventions

The architecture specification will be described using the the **C4 Model for Visualising Software Architecture**, which will present diagrams related to the target software through a hierarchical, notation-independent approach.

The document will follow a simplified chapter structure based on the one described in the **IEEE 830-1998 Standard** (IEEE Recommended Practice for Software Requirements Specifications). This will provide the generic layout for the presentation of the system's description and features, and will serve as a framing for the C4 model's diagrams.

The content will begin with a description of the system, starting with general features and qualities and then the main functionalities and qualities of each of its separate layers. Following this, the C4 documentation will be used to describe the architecture using four different perspectives (described in the following section).

1.4 Intended Audience and Reading Suggestions

This document is intended for individuals directly involved in the development of the **POWER Data Framework** within Altice Labs S.A. and the corresponding dissertation project within the University of Coimbra, as well as any members of the **POWER** Project or future implementation endeavours associated with this architecture.

Below is an overview of the structure of the document, along with additional information regarding the contents and layout of each Section.

- **Chapter I: Introduction** - In this section, the basic introductory information is exposed, along with the scope of the document and some information regarding the project.
- **Chapter II: System Description** - In this section there is a description of the layout of the system and its five main components (layers).
  - In these descriptions, the concept of "architectural driver" is used. Succinctly, it describes a specific requirement or constraint that significantly influences the design and decisions made during the architecture design process of that specific layer.
- **Chapter III: Architecture Description** - In this chapter, the C4 notation is employed to describe the architecture. This notation starts at a high level (Context Diagram) then *zooms-in* on the system, presenting increasingly granular views (Container Diagram, followed by Component Diagram). The last level of abstraction (Code) will not be utilized for this description.
  - **Section 1** - **Context Diagram** - Detail the system's relationship to external systems and actors to contextualise its operation.
  - **Section 2** - **Container Diagram** - Detail the system's *containers*, i.e. individually executable/deployable units.
  - **Section 3** - **Component Diagrams** - For each of the previous containers, detail the internal components which encompass the container's responsibilities.

The last level of abstraction in the C4 model (Code) is not presented in this document. It is an optional part of the C4 model as it is often closely linked to the actual implementation of the framework, and is generally used when designing at a very fine level (i.e. designing the internal structure of a micro-service, etc). This process was not performed as part of the dissertation project, which focused more on the viability of the technologies used in the architecture, resulting in implementations which consisted focused design prototypes, and not production-grade setups.

Chapter 2

System Description

The target system consists of a centralized data processing platform designed for multi-tenant IaaS/PaaS[1] operational environments. This platform aims to bring together various services within Altice Labs S.A. while also facilitating regulatory compliance. Additionally, it is intended to support future initiatives related to data privacy, security, and governance by offering appropriate tools and facilitate the adoption of organizational changes.

To better organize and structure the requirement specification process, it is necessary to create a well-structured and organized layout, splitting the highly complex and large target architecture into more manageable parts, enabling the isolated analysis of the main "building blocks" of the proposed system. Each of these components, or "layers" has its own distinct characterization, encapsulating vastly different requirements and needs. As such, it is necessary to formalize the partitioning, define clear boundaries between these parts and proceed with the analysis of each of them in separate, and in a system where they work together to ensure that all identified requirements are met.

In this chapter, the system is described, from a high-level perspective, along with its partitioning into *five* layers: **ingestion**, **storage**, **serving**, and **orchestration & administration**. Each layer has a specific role and objective, and together they enable the system to perform its main functionalities, and support all relevant quality attributes[2].

For each layer, a brief overview is provided, describing its main functionalities and motivations, highlighting the key goals that it aims to serve, and its main architecturally significant drivers.

[1] Infrastructure/Platform-as-a-service - operational contexts in which the company (in this case, Altice Labs, supplies either the execution infrastructure or a platform as a service to external parties for their own data management businesses.

[2] A textual representation of a certain quality of a system/component/interaction, such as Security, Scalability, Performance or Privacy, for example.

2.1 Overview

The system's main objective is to enable the processing of *large quantities of data* in a **multi-tenant** capable, **highly-scalable** and **available** context. Data is received from source systems, with varying types, sizes and formats, and is passed on to storage systems, for later use by the system's various business endpoints (e.g. data analytics, visualization, ML, etc.).

Throughout this "functionality" chain, there must be provisions for data quality monitoring, traceability and auditability, as well as considerations for infrastructure monitoring, automation and system health checking. These features will allow the system to adapt to the multi-tenancy context, involving external companies and clients in a compliant manner (especially with regards to data privacy regulations) through strict access control, domain *lock-out*/management and auditability, setting the stage for new paradigms such as the *Data Mesh*[3].

2.1.1 Architecture Partitioning

The architecture is partitioned into three views - Functional, Administrative and Orchestration. These views encompass the system's full characterization, and can be summarised as:

- **Functional View** - Those pertaining to the main intended functionality - data processing. This includes a further subdivision:
  - **Ingestion** - The entry of new data into the system.
  - **Storage** - The storage and management of large data volumes.
  - **Serving** - The presentation and consumption of data by external services.
- **Administrative View** - Pertaining to the regulatory constraints and encompassing the components which enable compliance.
- **Orchestration View** - Encompassing the components responsible for monitoring, automation and alerting of the system's execution.

So, to describe the system, a **five-layer** partitioning is performed: *Ingestion Layer*, *Storage Layer*, *Serving Layer*, *Administration Layer* and *Orchestration Layer*, following the aforementioned perspectives. The following sections will target these layers, providing a functional view of which tasks and goals the layer is focused on.

[3] A shared data processing/integration paradigm that makes each disparate domain in a business enterprise responsible for their data quality, and provides them with the tools to share it with the other domains.

2.2 Functional Layers

As previously discussed, this perspective encompasses three layers - the **Ingestion Layer (IL)**, **Storage Layer (SL)** and **Serving Layer (SV)**. These layers are where the majority of the system's data flows will occur, bringing data in from legacy systems and new sources, transforming it, storing and then delivering it to its intended endpoint. Figure 2.1 presents a simplified view of the Functional View, with its main tasks highlighted, layer by layer.

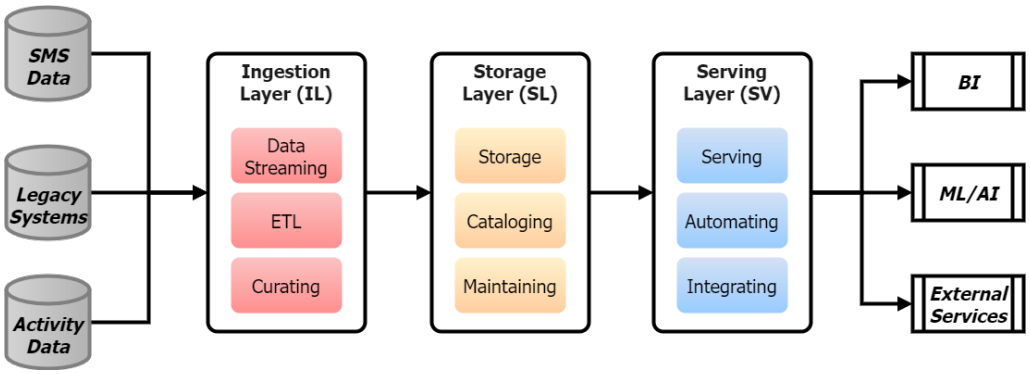

Figure 2.1: Schematic representation of the functional view of the architecture. Data flows are represented by arrows, and within each layer the main functionalities are identified.

Data which is streamed in from external sources is passed through the IL, receiving optional transformations and possibly even being curated based on its quality or other metrics.

Afterwards, the data is passed to the SL, where storage will take place. This storage should include some form of cataloguing as per the Lakehouse pattern, to allow for mixed data storage with warehouse-level performance.

Finally, in the SV, data is pulled from storage. This layer supports automation of pulling and querying, and provides the necessary tools for data integration, connecting the data to external services and data sharing. The SV will also encompass access control requirements, by virtue of being the external interface of the system.

The main drivers of these layers can be identified as:

- **Ingestion Layer** - *Performance* and *Scalability* in the data streaming operations to handle the variable loads and peaks of daily operations, especially in the multi-tenant environment.
- **Storage Layer** - *Availability* and *Integrity* of the storage systems, and the ability to ensure that data is never compromised even under hardware failure.
- **Serving Layer** - *Security* and *Traceability* of user actions, to create a stable and compliant data-sharing environment.

2.3 Administration Layer

On top of these functional components, the **Administration Layer (AL)** serves as an all-encompassing set of components that will enable the management of data throughout the entire lifecycle. Figure 2.2 provides a simplified look at the main tasks supported by the AL.

By leveraging modern technologies such as AI, Knowledge Graphs and more, it is possible to use metadata derived from all the layers of the system to build a characterization of the many data flows for auditability and traceability-related tasks.

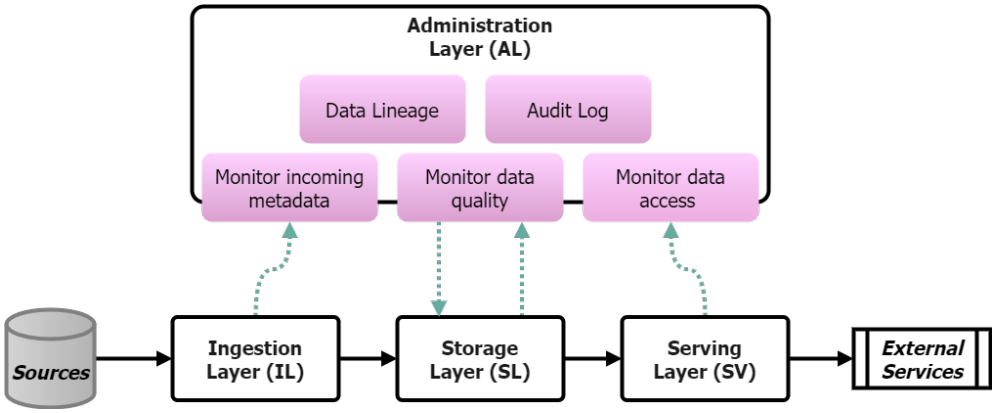

Figure 2.2: Schematic representation of the administrative view of the architecture. Metadata flows are represented with dotted green arrows. The data sources and endpoint external services were joined into a single component for this view.

Through these functionalities, it is possible to maintain an extensive audit log and use it to ensure compliance, as well as assist in tasks related to data quality and understanding how data changes throughout its lifecycle by monitoring all changes made to data objects and integrating them into a comprehensive lineage view.

The main drivers for this layer are **Isolation**, **Privacy** and **Usability**:

- **Isolation and Privacy** - In ensuring that data is not widely available to anyone who seeks it, and that each domain/team/user has their own set of data and access is controlled.
- **Usability** - In creating an environment that can facilitate integration, data sharing and inter-operability between domains in a safe, compliant way.

2.4 Orchestration Layer

In parallel with the management of the data, there must also be a feature-set for managing the system's execution, and to ensure that scalability, performance and productivity does not suffer during the various states of operation the system may be subjected to.

In the **Orchestration Layer (OL)** components work together to ensure the system's health, through a combination of logging, log analytics and automated maintenance routines. Figure 2.3 provides a simplified view of the main activities that take place in the OL.

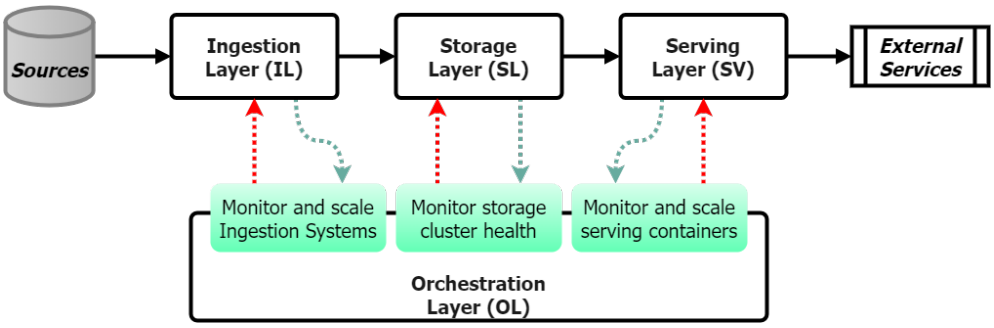

Figure 2.3: Schematic representation of the functional view of the architecture. Data flows are represented by arrows, and within each layer the main functionalities are identified.

To support these functionalities, distinct message queues/pathways must be defined, to ensure long-term scalability and flexibility of the monitoring and management solution. The potential for this layer is significant, since it may use its role as overseer to significantly automate the management and health checking of the system through the configuration of alarms, automated actions and reactions to system events. Taking this into account, the key drivers for this layer are **Consistency**, **Modifiability** and **Scalability/Performance**:

- **Consistency** - In logging, monitoring and maintaining the harmony of the system through constant, real-time analysis of system health data.
- **Modifiability** - In adding new monitoring components, new analytics operations, action automation and even registering new components into the monitoring suite. These needs must be met, allowing for a resource-conservative approach to management and maintenance.
- **Scalability/Performance** - In ensuring harmonious operation at all load levels, the monitoring, rate-limiting and automation must be able to perform, and scale up/down whenever necessary, maintaining an elastic profile.

Chapter 3

Architecture Description

After the identification of the requirements (described in the Requirement Specification), an iterative process was used to develop the architecture from a draft into its final version.

This process refined the architecture, starting with the ingestion processes and components, and ending with the serving, storage and governance components. These were targeted by experimental validation over the course of two iterations, as far as the project's timeline would allow.

With this process concluded, the architecture was documented following the C4 Model, presenting three of the four views: **Context, Container and Component**.

The last level of abstraction in the C4 model (Code) is not presented in this document. It is an optional view as it is often closely linked to the actual implementation of the framework, and is generally used when designing at a very fine level (i.e. designing the internal structure of a micro-service, etc). This process was not performed as part of the dissertation project, which focused on the viability of the technologies used in the architecture, resulting in implementations which consisted focused design prototypes, and not production-grade setups.

The following sections will approach the three developed levels of the C4 model. Starting with an overview, then progressing to the C4 views: the *context* view, *container* view and then moving onto the several different *components*, presenting a diagram for each view, as well as a textual explanation clarifying the functionalities and qualities supported by each of the different parts of the architecture.

Alternatives are presented for the data pipeline section of the framework, identified through the state-of-the-art research and iterative process described in the dissertation report associated with this specification. Additionally, a section dedicated to the future work is presented, presenting the avenues for further development of this framework architecture.

3.1 Overview

The architecture can be summarised as a system built upon the use of a **Apache Hudi Lakehouse** on top of a **Min.IO/S3** object store (or Data Lake), which creates an environment fit for high-speed, highly scalable data serving through engines like **Spark** or **Presto**. This data store is fed by a multitude of **Apache Kafka** streams, either directly or after curating *ETL* via **Spark**. The framework's many data flows are audited, recorded and traced by the **Acryl DataHub** metadata catalog, and the infrastructure is constantly under health monitoring by a **Prometheus**-based monitoring dashboard.

3.1.1 Ingestion, Storage and Serving

This framework ingests data from a number of source systems through the creation of designated **Kafka Producers** that gather data (in push or pull configurations) and send them through a network of brokers onto the **Hudi Spark cluster**, which extracts metadata and categorizes data before storing it in a **Min.IO/S3 bucket**. This chain is built around *scalability*, *fault tolerance* and *high-throughput*, and allows for many ingestion styles under the *Kappa architecture pattern* (real-time and batch using the same component, Kafka)

Using **Spark and Presto**, the flexibility of the querying systems is maximised, as **Spark** enables the use of Hudi-specific audit-related queries (such as the *time-travel* or *data lineage* query) while **Presto** allows for extremely performant ad-hoc or preset querying. The use of **Apache Airflow** enables the automation of queries on the serving side by automating Spark or Presto jobs. Additionally, Presto can be executed on a Spark cluster, further exploiting the ability to execute these technologies in parallel. Both Spark and Presto come provisioned with *credential-based access control* which can be connected to external identity management software.

3.1.2 Administration and Governance

To establish an audit log and traceability for entire framework, the use of the **Acryl DataHub** metadata catalog helps build a *graph-based representation* of the data within the framework, showing origin, destination, accesses, modifies and lineage for any data object within the framework. This catalog is hooked to all the components which handle data within the framework, and will, over time, create a robust audit log to ensure data governance, safe data sharing and compliance are all possible within the framework.

3.1.3 Infrastructure Control

All the identified components present ways to extract metrics and data which can be interpreted by the **Prometheus** monitoring and alerting software, which allows for alerts and management of the system's infrastructure.

3.2 Context View

In the context view (Fig. 3.1) for the system, it is possible to see the three main actors involved with the operation of the framework:

- **The Process Manager** - Who is responsible for the health monitoring and configuration of the framework's components. Uses an external dashboard which receives data posted by the framework.
- **The End User** - Who uses an interface to interact with the data inside the framework. The interface lets the tenant/user see the data which he is authorized to see, query it, and process it through the use of the components described further in the chapter.
- **The Data Manager** - Who has direct access to the internal governance tools of the framework, and can manage tenant access and authorizations through an external IAM.

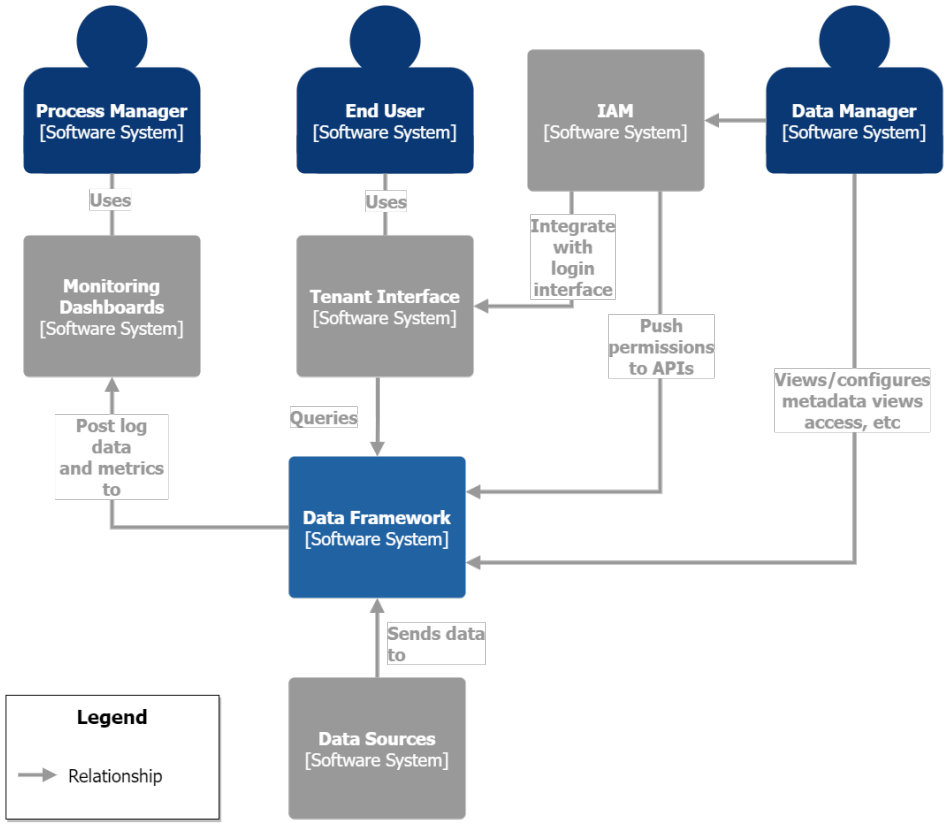

Figure 3.1: Context diagram for the architecture. Displayed are the three relevant actors and the systems they interact with.

The tenant interface, IAM and monitoring dashboard systems were not developed in this project and remain as avenues for future work.

3.3 Container View

In the container view (Fig. 3.2) the various executable parts of the framework are visible. Each of these containers is executed either on Docker/Kubernetes or on bare-metal. It is also possible to operate some of the containers via cloud-based deployment (namely the Min.IO storage component).

Here, the main data flows are exposed - the flow of data *from source* systems *to end-users*, the metadata governance flows and the infrastructure monitoring flows.

As previously described, the system offers a central path for data to travel through and from - the **Kafka Ingestion Cluster** receives data from outside sources, passing it onto the **Lakehouse Hudi/Spark Cluster**, which saves it to **Min.IO/S3**, and then passes data along to the **Serving Spark/Presto Cluster** whenever required by the tenants/end-users. Data may also be served from the Serving Cluster back to the Ingestion cluster for re-processing, ingestion of newly generated data or model saving. These three layers support the main qualities demanded of the framework, in regards to the function of data ingestion, processing, storage and serving:

- **Scalability** - All components were selected with scalability and cloud-native operations in mind, and experimental validation took place, ensuring they are inter-operable with each-other, albeit in a small, low resource scenario.
- **Performance** - The main points of data traffic - Kafka and Spark - are highly performant, and, due to their scalable, distributed and expansible nature, can guarantee performance at varying load levels.
- **Cost Control** - All components were selected with cost control in mind, and, aside from being open-source, include measures for rate control, performance limiting and resource management.

Along this path, the **Prometheus deployment** extracts metrics, logs and health data in order to maintain a record of system execution, and potentiate reactive and predictive maintenance efforts to ensure the long-term stability and scalability of the system (as some of the components require some level of manual adjustment to scale correctly).

For each of the significant points of data movement (*Ingestion, Storage and Serving*) the **DataHub Catalog** receives metadata transactional records, keeping a complete audit log of all data movements within the system. This allows for the verification of data lineage, provenance and access for any given data object in the framework, at all points of its lifecycle.

The DataHub catalog ships with an interface which enables the Data Manager's use cases of regulating data discovery, sharing and auditing activities. This catalog interface is also usable by the end-users, who have their own views of the system's data, typically filtered by their own authorization level as defined by the Data Manager.

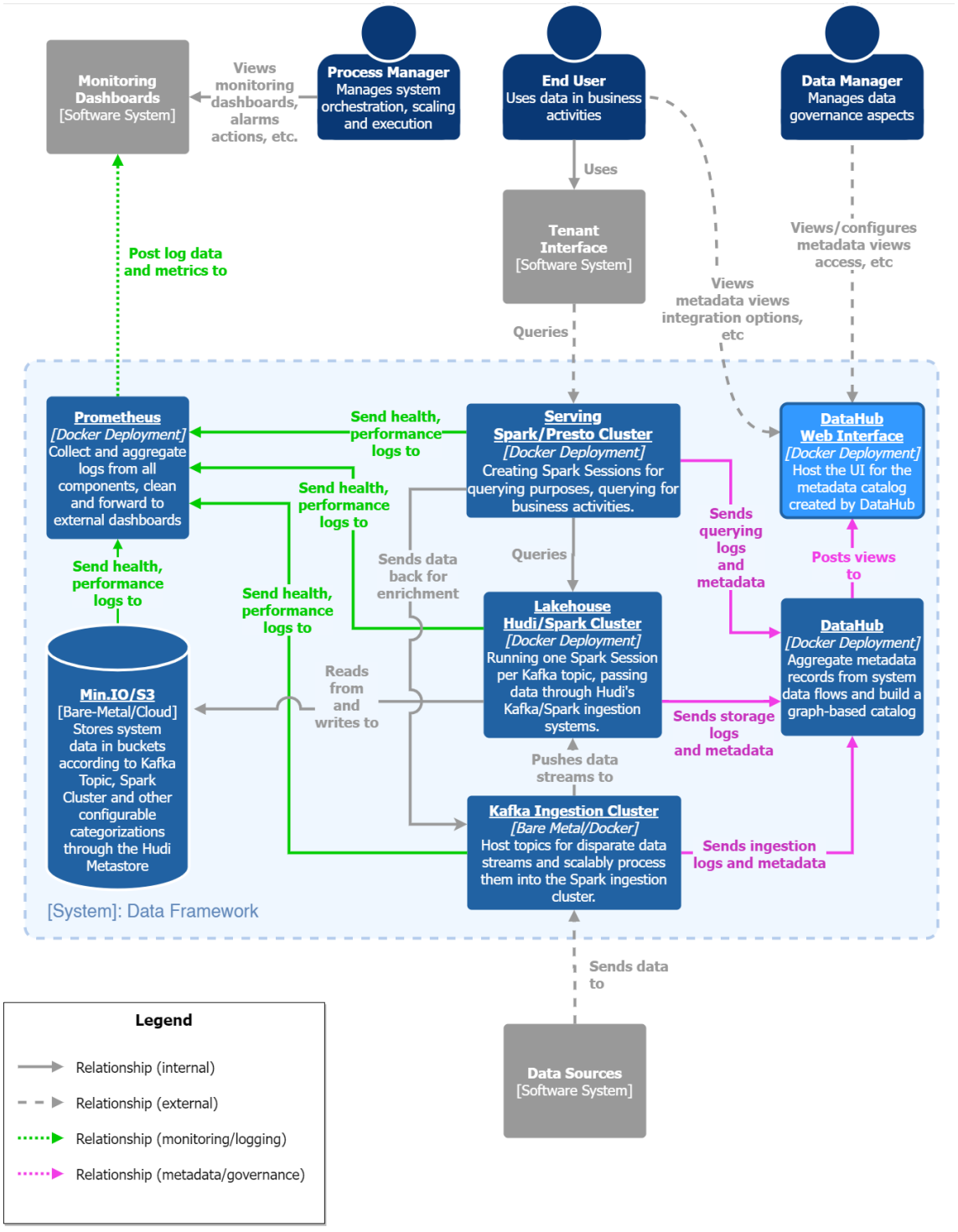

Figure 3.2: Container View of the Data Framework system. Grey arrows indicate data flows, green arrows indicate log flows and pink arrows indicate metadata flows.

3.4 Component Views

For each of the previously identified containers in the Data Framework system, a component diagram will be presented, exposing the inner components of each container as well as their connections/relationships to other containers.

3.4.1 Kafka Ingestion Cluster

As the container responsible for the entry of data into the system, the **Kafka Ingestion Cluster** (Fig. 3.3) relies on the creation of paths for data to travel to the framework in scalable way, fit for large volumes of data and a high-throughput execution profile. This cluster can be deployed on-prem or through cloud-based Kafka SaaS deployments like Confluent.

Functionalities

This container uses Kafka streams to serve the data sources outside of the system. They consist of a Producer-Broker-Consumer set that can be configured in push, pull or even a batch-style ingestion that aggregates data before passing it to storage. This is done through explicit programming of the logic within the Kafka Producer modules (where they can be connected to any API, legacy system or external data source).

Additionally, there's a separate Kafka stream for *enrichment* and *data integration* purposes. This stream receives data from the end-users (through the Serving Spark/Presto Cluster) and inserts it into the framework, making it possible for data enrichment and integration practices to have their own ingestion path, which can even include intermediate processing if necessary.

The Kafka clusters and components are all monitored through the metrics and logs extracted from its coordinator - the Kafka Zookeeper - and they are passed onto the Prometheus monitoring software for log analytics, automated alarms and even actions. The several Kafka topics are connected to a DataHub plugin, syncing the data, along with its metadata, into the DataHub catalog.

Qualities

Through the use of Kafka's native scalability, these clusters can maintain their performance even under highly concurrent, high-traffic workloads (through presizing for the intended loads) and more clusters/streams can be instantiated to serve tenant's needs in an elastic way.

Experimental validation indicates that Kafka can serve in this scenario with very good results in *integrity*, *scalability* and *performance*. Its flexible implementation also satisfies the requirements of *modifiability*, enabling a nearly limitless palette of external connections.

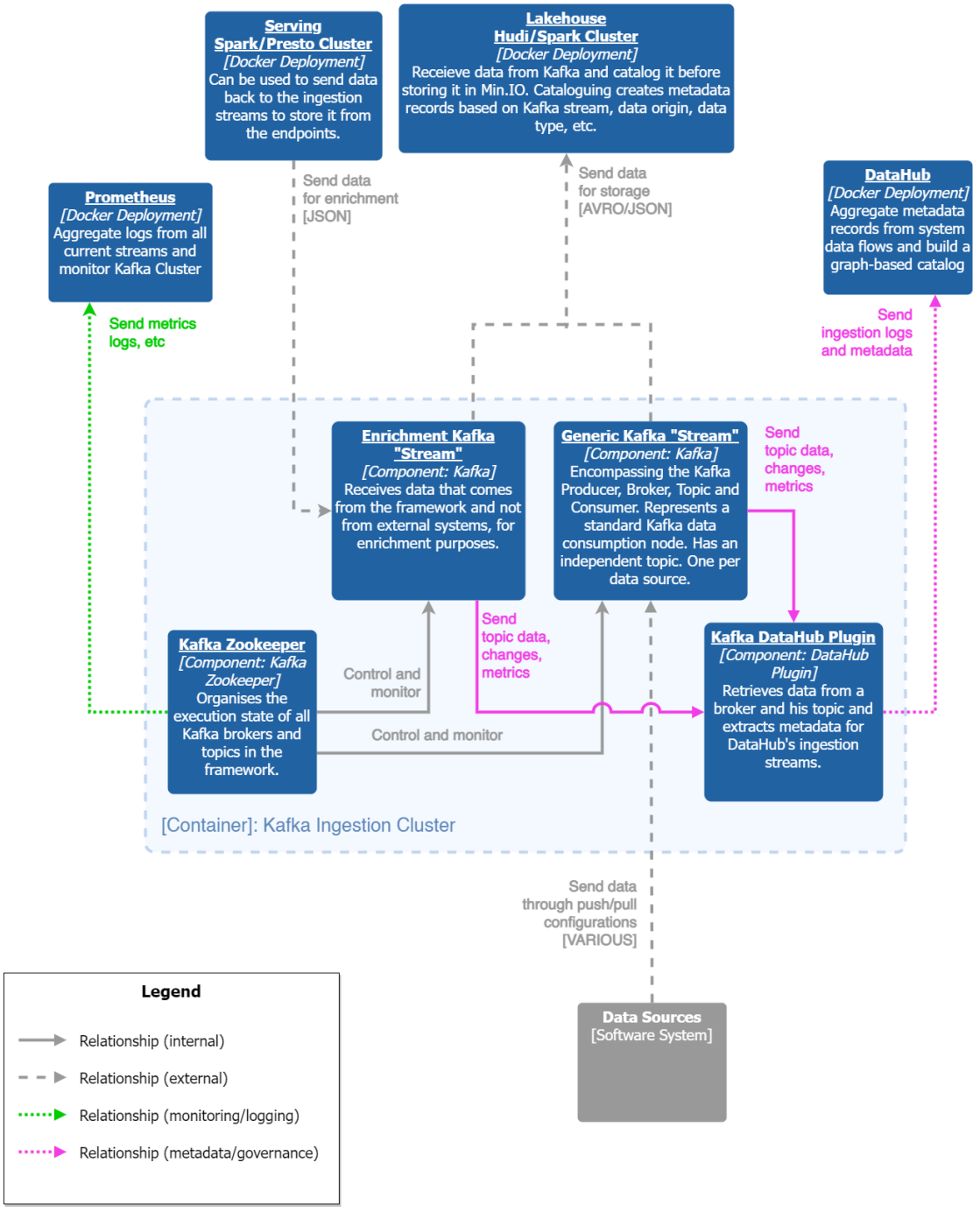

Figure 3.3: Component View of the Kafka Ingestion Cluster component. Grey arrows indicate data flows, green arrows indicate log flows and pink arrows indicate metadata flows. Additionally, dashed grey lines indicate data flows from/to outside the component.

3.4.2 Lakehouse Hudi/Spark Cluster

Once data has entered the system it is the job of the **Lakehouse Hudi/Spark Cluster** (Fig. 3.4) to categorize the data, store it, and serve it to the numerous services which rely on it. The tools used to create this container are extensively documented, namely in the identified pairings (Hudi/Min.IO and Spark/Presto), and rely on open-source technology, with both on-prem and cloud-based deployments available.

Functionalities

Through the use of Hudi as a Data Lakehouse platform, it is possible to empower the Min.IO/S3 object storage (typically used as a Data Lake) a build a more flexible and performant system. The Hudi/Min.IO combination is able to serve the required functionalities by enabling:

- **Lineage**, **time-travel** and **audit** queries (via SparkSQL).
- Build **custom categorizations** for data streams.
- Create **data retention policies** for specific data types/categories.

Hudi also natively supports two very powerful query engines: the aforementioned SparkSQL and Presto, with a large array of native integrations. Additionally, because Hudi runs on a Spark cluster, it is possible to create Spark applications for ETL jobs and other associated data transformations/integrations that can be configured to run automatically on designated input streams (functionally creating a curated data path).

The Prometheus dashboard is connected to Hudi's Spark cluster, where metrics are exported to monitor the performance and health of this cluster. Data flows are monitored and logged through the use of metadata extraction via the Spark DataHub agent, which is configured to automatically parse Spark metadata transactions and log them in the Catalog.

Qualities

The Hudi/Min.IO/S3 system is a robust, *performant*, *scalable* and *highly available* set-up, featuring many options to maximise these qualities, such as parallel execution, redundancy and multi-node operations with error correction.

Experimental validation ensures compatibility between the two, as well as successful data transit into the open file formats used by Hudi, as well as the reverse path of re-building from these formats into easily processed JSON files through the Hudi query system, fulfilling specified requirements of data *integrity*.

The ability to perform lineage and time-travel queries also fulfils non-functional requirements of *auditability* and *security*, which aids in adoption of compliant strategies.

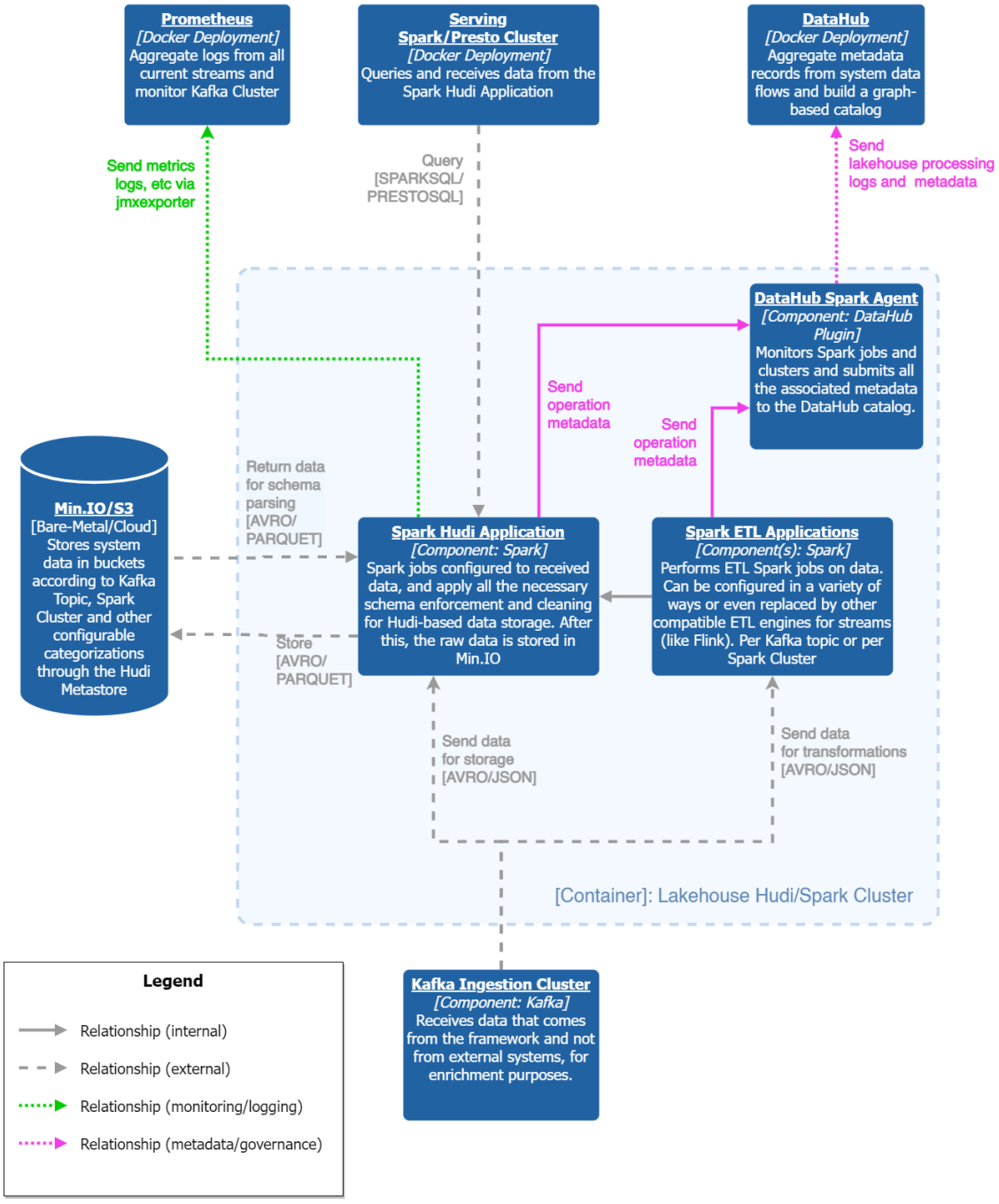

Figure 3.4: Component View of the Lakehouse Hudi/Spark Cluster component. Grey arrows indicate data flows, green arrows indicate log flows and pink arrows indicate metadata flows. Additionally, dashed grey lines indicate data flows from/to outside the component.

3.4.3 Serving Spark/Presto Cluster

To interact with the stored data, the **Serving Spark/Presto Cluster** (Fig. 3.5) container leverages the Spark execution environment to create a performant and flexible querying layer, featuring open-source and highly documented technologies fit for on-prem and cloud-based deployment.

Functionalities

The container relies on the use of the SparkSQL and Presto to query the Hudi service, allowing for a variety of query types (ad-hoc, preset, lineage, time-travel) and for a considerable degree of control over the authorization and access of the end-user. This is performed through the use of interfacing APIs which may connect to an external *Identity and Access Management* (IAM) software, providing access control to tenant queries.

To send data into the framework, a serving-side API enables data to be routed back through the Kafka ingestion cluster, and go through the categorization processes of the Hudi lakehouse.

The use of an Airflow scheduling component enables end-users to automate queries to their external services, and it ships with access control associated to its central dashboard. This feature enables the automation of data presentation to external AI/ML models, and also the snapshotting and saving of models/model checkpoints to long-term storage.

Much like the previous containers, the DataHub catalog tracks all uses of data into the metadata catalog, while Prometheus manages the health and execution state of the container's components.

Qualities

The use of Presto and Spark as querying engines allows the system's queries to be very fast, executing in a highly *performant* way, even in ad-hoc scenarios.

By routing the accesses to data through APIs it is possible to ensure a large degree of control, interfacing with external IAM software and enabling domain lock-out. Spark, through combination with Hudi, also enables credentials to be propagated through the querying process, returning only the data that the user has access to, omitting fields that they are not authorized to see.

This, along with the recording of all data movements in DataHub fulfils the qualities of *security*, *traceability* and *privacy* which are required for a safe data handling and sharing environment, potentiating the evolution to a Data Mesh scenario.

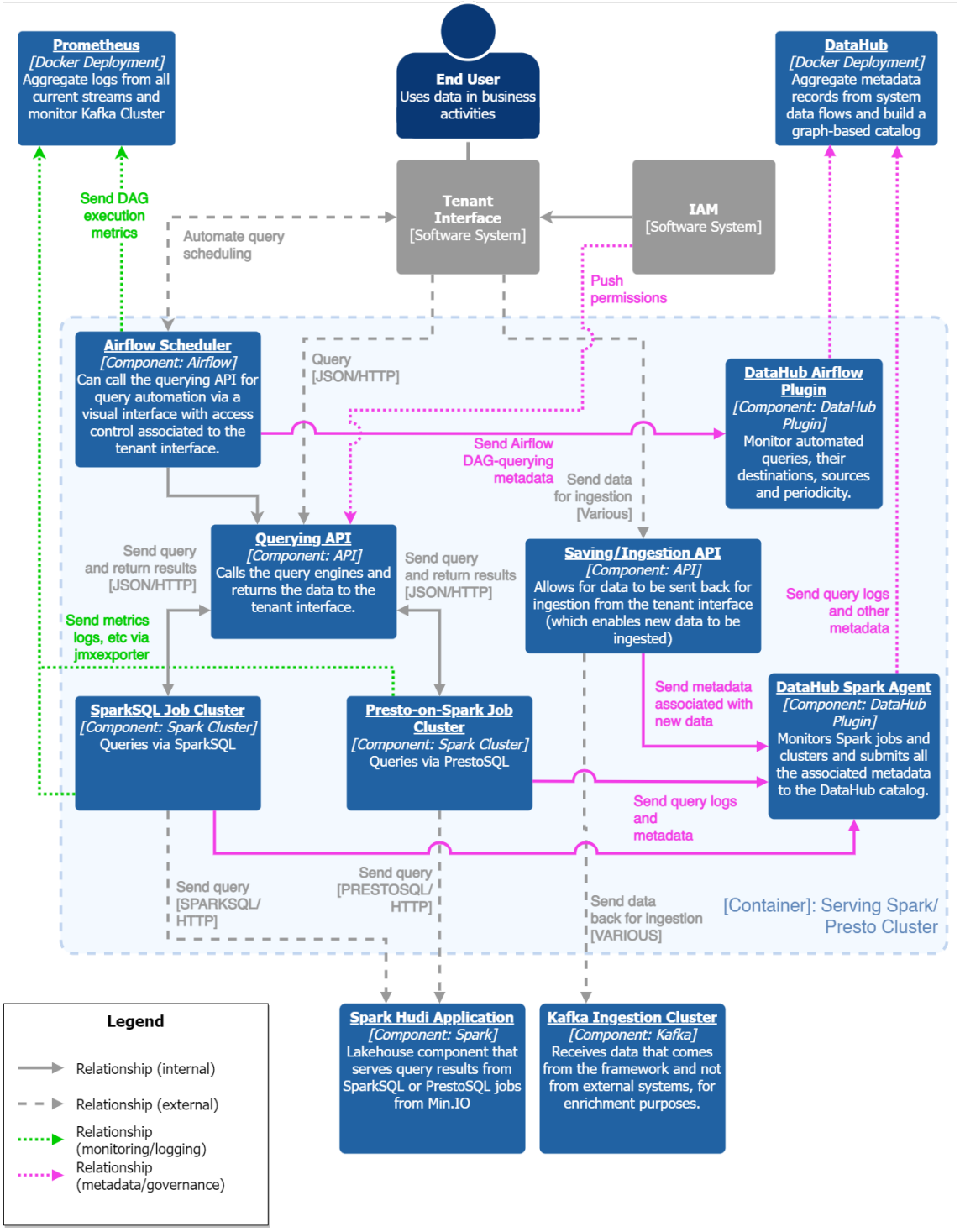

Figure 3.5: Component View of the Serving Spark/Presto Cluster component. Grey arrows indicate data flows, green arrows indicate log flows and pink arrows indicate metadata flows. Additionally, dashed grey lines indicate data flows from/to outside the component.

3.5 Alternatives

Along with the selected technologies for the final architecture, some of the researched components are, based on the selection criteria described in the previous chapter, possible viable alternatives for the functionalities of the main data pipeline.

As research could not cover or experiment on the identified metadata catalog alternatives, this section will not present any alternatives for the data governance side of the framework, as DataHub is both the most featured and most well documented choice in this area.

Additionally, some considerations can be made on the management and monitoring components, although mostly these rely on the use of the built-in management tools provided by cloud-native service providers.

The alternatives presented in this section provide the same feature-set as the selected components, but were not selected as the exclusion process undertaken in the refinement stages favoured a more suitable alternative. Nonetheless, they will be presented as they may present a better fit if new conditions arise in the future of the Data Framework Architecture project.

3.5.1 Ingestion

For Ingestion, as a replacement of Kafka, few components present as much flexibility and performance. However, there are options, namely the ones offered by the **AWS Kinesis Suite**, which present very good performance and scalability features and an unparalleled level of availability. They can be described as follows:

- **AWS Kinesis Data Streams** - Real-time streaming service which serves as a drop-in replacement for Kafka, featuring a lot of the same principles (records, re-transmission and "topic"-based streams). Like Kafka, the scalability is manually performed by pre-sizing and adding more "topics" or, as Kinesis documentation refers, *shards*. It specializes in real-time streaming featuring low latency by default.
- **AWS Kinesis Data Firehose** - Batch, un-managed data streaming solution that focuses on high throughput, while sacrificing some the low latency provided by a more structured service. Latency with Firehose is often above 60ms for a given data record's transmission, making it less suitable for real-time workloads, and more useful for massive transfers of data under less strict requirements.

It should be noted that, while these services provide similar functionalities, they are more advantageously used when the rest of the chosen technology stack is also within the AWS Suite, as it ends up providing native, "one-click" integration, as well as a connected and unified management dashboard system.

Alternatively, it is possible to offload the management of the Kafka cluster to an external service provider, maintaining the exact same functionality while gaining the benefit of high availability and improved latency.

Both the AWS and externally managed Kafka services will, however, incur in costs that can become exacerbated by the multitude of streams demanded by a multi-tenant architecture.

3.5.2 Storage and Serving

The storage functionalities and qualities, provided in the architecture by S3/Min.IO with Hudi as the Lakehouse component, can be achieved using other platforms, albeit with significant changes being necessary to the serving stack. As such, these alternatives are grouped together.

AWS S3-centric

By using S3 and choosing to use the AWS managed cloud-native solution, it is possible to also use **AWS LakeFormation** and **AWS Glue** to achieve the Lakehouse functionality. However, the compatibility with the currently chosen Serving stack is not clear, and instead, it is recommended that this use is paired with other AWS services, as these integrate seamlessly without the need for extensive modification and untested pairings.

As per the research conducted and presented in Chapter IV, the use of **AWS Athena** and **Redshift** can, in combination, result in a powerful interactive query engine which can satisfy the requirements of the Serving layer.

HDFS-centric

These solutions typically present some challenges, as the Lakehouse components typically rely on tailor made connectors fit for more recent object storage services (such as S3, Azure, Google Cloud).

However, in this case, Apache Hudi presents an HDFS connector which may enable it to resolve this problem, however, experimentation would be necessary to ensure compatibility of the stack. If possible, this would make the current serving stack compatible, and the requirements would likely still be met under the HDFS-based platform.

3.6 Future Work

With the work taken as far as the dissertation project's constraints would allow, the avenues for future work were identified. Three main areas were identified:

- **Conclude the experimental validation** - While the main data pipeline was validated experimentally, the conditions for the tests were simply meant for rapid prototyping and used significantly reduced hardware specifications in their execution. Additionally, the hardware limitations made it impossible to test the usability of the metadata catalog correctly, and thus this could also be worth exploring. The two main tasks here are:
  - Test the framework with more resources, using distributed configurations, virtualization and create synthetic scenarios closer to the estimated production workload.
  - Finish testing the governance aspects (catalog metadata visualization) with a suite of tests designed around usability.
- **Refine the monitoring solution** - As the monitoring was indicated as an area which should not be focused on, as internally, Altice Labs S.A. already had significant experience in the area and had proposed existing tooling (the Prometheus toolset), it may be valuable to explore the monitoring and management drivers more and see how Prometheus can be potentially used in the PaaS/IaaS scenario.
- **Design the interfaces for the users** - This was outside of the scope of the project, but, nonetheless, remains a very important part for the production architecture, as some of the concepts associated with managing a multi-tenant architecture can not be explicitly solved through architectural decisions, but rather through interface design processes.

And, while there are still some areas where improvement is possible, implementation could feasibly begin on the data pipeline side, as the technologies which were presented, as well as their data flows and component relationships will allow for the identified functional and non-functional requirements to be met.