FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Framework for Dataflow Orchestration in Lambda Architectures

Rui Botto Figueira

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Hugo Sereno Ferreira, PhD

March 18, 2018

A Framework for Dataflow Orchestration in Lambda Architectures

Rui Botto Figueira

Mestrado Integrado em Engenharia Informática e Computação

March 18, 2018

Abstract

Nowadays, each user interaction in an online platform originates data that is relevant to derive knowledge and provide business information to it's stakeholders, but to do so, data needs to be transformed or combined before being ready to be analyzed. As a company grows, the expanding customer base increases the volume of data that needs to be integrated into it's big data cluster and with it, the complexity of it's various processing pipelines.

At Farfetch, the lambda architecture implementation is supported by an everchanging ecosystem that needs to support multiple technologies and paradigms in order to be interoperable. Each pipeline can be seen as defining a dataflow that specifies how data traverses between the several layers from start to finish of a process. This flow is not an explicit property of the pipeline but an implicit one defined by it's underlaying technological implementation which inherently promotes tight coupling between the components definition and the specification of how data should flow. This leads to difficulties in creating a common model to be responsible for component reuse, rising the engineering costs in the pipeline construction and creating a lack of flexibility in the orchestration of the dataflow.

In order to tackle this problem, this dissertation proposes the creation of a framework that enables the setup and configuration of the flow of data in the processing pipelines that are part of the lambda architecture by weighting on abstraction as it's main feature. The proposed solution will have it's focus on a flexible and modular architecture that will allow big data engineers to establish the flow from the moment that data is ingested until it is written to a final persistent storage, while leveraging the power of already used technologies in the cluster.

By creating a robust actor system empowered with reactive programming principles, the architecture should be responsible for managing the communication between different components while taking advantage of it's own data abstraction, allowing an easy setup of complex data flows and facilitating the creation of data pipelines by specifying explicitly the flow of data that it should encompass.

Keywords : *Software Engineering, Software Architecture, Big Data, Data Orchestration*

Classification :

- *Computer systems organization → Architectures → Other architectures → Data flow architectures.*
- *Information systems → Data management systems → Information integration → Mediators and data integration.*

Resumo

Hoje em dia, cada interação de um utilizador numa plataforma *online* origina dados que são relevantes para extrair conhecimento e providenciar informação do seu negócio aos seus *stakeholders*, mas para o fazer, os dados precisam primeiro de ser transformados ou combinados de forma a estarem preparados para ser analisados. Com o crescer de uma empresa, a expansão do número de utilizadores aumenta o volume de dados que têm de ser integrados no seu *cluster* de *big data* e devido a isso, a respectiva complexidade das suas várias *pipelines* de processamento.

Na Farfetch, a implementação da sua *lambda architecture* é suportada por um ecossistema em constante mudança que necessita de suportar múltiplas tecnologias e paradigmas de modo a ser interoperável. Cada *pipeline* pode ser vista como definindo um *flow* de dados que especifica como é que estes percorrem as diferentes camadas de ínicio ao fim de um processo. Este *flow* não é uma característica explícita da *pipeline*, mas uma característica implicitamente definida pela sua implementação tecnológica, o que inerentemente promove um forte acoplamento entre a definição dos componentes e a especificação de como é que os dados devem fluir. Isto, por sua vez, leva à criação de problemas na criação de um modelo comum responsável pela reutilização de componentes, criando um crescimento nos custos de engenharia envolvidos na construção das *pipelines* e uma falha de flexibilidade na orquestração do modo como os dados fluem.

De modo a resolver este problema, esta dissertação propõe a criação de uma *framework* que permita a configuração do *flow* de dados nas *pipelines* de processamento que fazem parte de uma *lambda architecture*, usando a sua abstração como principal característica. A solução proposta irá ter como foco uma arquitectura modular e flexível que irá permitir aos engenheiros de *big data* estabelecer um *flow* desde o momento em que os dados são ingeridos até eles serem escritos para uma camada final de persistência, utilizando para isso as capacidades das tecnologias que já se encontram em uso no *cluster*.

Através do desenvolvimento de um sistema de actores robusto, fortalecido pelos princípios de programação reativa, a arquitectura deve ser responsável pela gestão da comunicação entre diferentes componentes, tirando partido da sua própria camada de abstração de dados e permitindo desta forma a fácil configuração do *flow* de dados e permitindo a criação de *pipelines* através da especificação explícita do modo como os dados devem ser orquestrados.

Palavras Chave : *Software Engineering, Software Architecture, Big Data, Data Orchestration*

Classificação :

- *Computer systems organization → Architectures → Other architectures → Data flow architectures.*
- *Information systems → Data management systems → Information integration → Mediators and data integration.*

"In character, in manner, in style, in all things, the supreme excellence is simplicity."

Henry Wadsworth Longfellow

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

E-Commerce	Electronic Commerce
DFS	Distributed File System
DSL	Domain Specific Language
ETL	Extract Transform and Load
ERP	Enterprise Resource Planning
HDFS	Hadoop Distributed File System
IP	Internet Protocol
IoT	Internet of Things
I/O	Input and Output
JAR	Java Archive
JVM	Java Virtual Machine
GUID	Globally Unique Identifier
KPI	Key Performance Indicators
MVP	Minimum Viable Product
ODS	Operational Data Store
ORM	Object Relational Mapping
RDBMS	Relational Database Management System
RDD	Resilient Distributed Dataset
REST	Representational State Transfer
SSL	Secure Sockets Layer
SCM	Supply Chain Management
TB	Terabyte
UI	User Interface

Chapter 1

Introduction

Nowadays, electronic commerce (e-commerce) platforms heavily depend on insights generated by the data produced by their users. This data allows companies to build better recommendation systems, understand user profiles, pinpoint points of failure and understand trends, which ultimately leads to an indispensable feedback engine that can be used to solve problems and make improvements to increase sales.

Typically, data is generated by different user activities on a platform, examples of it being clickstreams, user transactions and user viewing history. Almost every single action a user does online, is able to be measured and quantifiable. But for most of this data to be relevant, it has to be processed in order to feed the services used to extract information from it and for it to provide some source of feedback. To do this, most companies build a batch-processing platform together with a streaming-processing that is service oriented. This allows the generation and processing of information in real-time to multiple services. Architectures that have both systems are usually named lambda architectures.

There are several frameworks that allow companies to process data, being it in real time or not, but ultimately is up to the company to decide how to structure their platform in order to support their needs. What is common between all of them is the fact that for every new source that gets integrated into the system, they all have to orchestrate the flow of the data that comes in, in order for it to be put to used correctly.

1.1 Motivation

As a company grows, so does the number and diversity of data sources that have to be integrated into the platform. Each one, representing a new pipeline with different needs when it comes to ingestion, transformations and integration. Every new source, requires the specification of how to ingest data from the source, how to transform it and to which data sink to direct it.

Data can pass trough many different layers of processing after being ingested and it can be cumbersome to have to manually create and define the entire dataflow each time a new data source gets added. A processing job can be made of chains of different sources, processors and sinks until a final point is reached. This creates a challenge when it comes to the operational complexity of multiple pipelines, leading to problems in data quality due to the possible difference in quality standards between teams and to the rise of engineering costs created by the differences on the pipeline construction.

When it comes to the engineering structure of each dataflow layer, most of them share a common construct as they all read from sources and output to other processes or sinks, which means that for every new pipeline constructed there's a big part of the system and process that could be re-used between them to provide a common interface to process the data independently of the technologies used.

1.2 Aim and Goals

The aim of this dissertation is to define and implement a modular architecture capable of facilitating and managing the orchestration of new data pipelines that are to integrate into the big data cluster. The objective is not only to be able to support the setup of the dataflows that compose the pipelines, leveraging current used technologies, but also to reduce the time needed to create them, while allowing them to have a set of standards to promote uniformization of processes, together with data lineage support.

The expected final result of the work proposed in this dissertation will be a implemented prototype of the architecture defined, that should be validated against a set of technologies used at Farfetch through a proof of concept test. The final goal of the encapsulated framework is to allow developers to specify how data should flow from source do destiny, encompassing all the necessary layers of abstraction for it's processing and middle communication steps. With it, teams can concentrate their work in creating modular implementations of the needed technologies while taking advantage of the framework to leverage the flow of data.

1.3 Context

This problem was proposed as a MsC Dissertation by Farfetch, a Portuguese e-commerce platform that specializes in selling products from luxury fashion brands. It currently processes 10 million site visits per month and has around 120.000 curated products from 1500 brands.

The company is currently improving their development practices when it comes to unifying the pipeline creation methodologies. To do this, they were faced with the problem of setting dataflows for new pipelines while assuring data quality across teams together with the need to decouple the dataflow setup from it's implementation. Currently for every new data ingestion job there's a lot of unnecessary work that gets repeated across tasks, so their objective is to find a way to solve the

problem of ingesting and orchestrating the data flow in a more structured way that also takes in consideration their future development and progress.

The prototype of the framework presented above will be developed and applied in the context of the big data team of the company that is included in the business intelligence cluster.

1.4 Document Structure

In chapter 2, it's provided a contextualization on the area of big data together with an overview of the current system architecture of Farfetch's big data platform and an overview of technologies used in that context. The problem addressed in this dissertation together with it's major challenges is analyzed on chapter 3. Chapter 4 presents an overview of the main features and decisions that were taken in account to define the architecture behind the framework and chapter 5 presents it's implementation. In chapter 6 we test the framework by creating a simple pipeline with technological extensions and in the final chapter 7 we reflect upon the main contributions of this dissertation, together with the proposal of some future work to extend the work developed.

Chapter 2

State of the Art

This chapter provides an overview on the area of big data and it's importance for e-commerce platforms, together with an analysis of typical architectures and strategies used to process data at that scale. In section 2.1 it's presented how big data is important to derive relevant insights in e-commerce platforms. In section 2.2 different technologies across the data processing pipeline are analyzed together with an overview of the typical architectures found in a big data cluster and in the section 2.3 it's presented the current technological architecture of the system used in Farfetch.

2.1 Big Data Overview

Big data was a term that started to be broadly used in the technological high-tech community in the nineties popularized by John Mashey [Die12]. It was used as a simple expression to describe how the boundaries of computing were advancing and how we needed a term to describe the new wave of data computation . Nowadays, Big Data has evolved into an area of Software Engineering that categorizes systems of big dimension that process data at a very large scale. In 2012, Gartner [1], an American research and advisory firm in the field of information technology, defined Big Data as:

> "Big data is high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization.'

This area started to get a lot of attraction in the last years due to the sheer amount of data created by the new wave of technology we live in [Hil11]. In 2015, it was estimated that everyday, 2.5 Quintillion bytes of data were generated [Wal]. The major goal of big data as an area is to provide the means to process large amounts of data so that insights can be created to generate knowledge. Knowledge in this sense, is data with context associated. [AH15]

[1] https://www.gartner.com/technology/home.jsp

Big data can be characterized by the it's 5V's [Ish15]:

- Volume: the main characteristic that makes data "big", is it's sheer volume;
- Variety: it usually encompasses information from multiple sources, which leads to a big variety of formats and structures;
- Veracity: the data has to be trustworthy for it's stakeholders to believe the insights generated;
- Velocity: data has to be generated at a high frequency;
- Value: it has to be important in the context where it is created.

2.1.1 Big Data in a E-Commerce Platform

E-Commerce is one of the biggest areas to take advantage from the advent of big data. The main reason is the diversity of metrics generated by the interactions with the system. Any system where the user interaction can be traced to a profile and requires some kind of transaction system related to a persistent state storage is a good target for the application of data analytics. [AW16]

With the collection of data, companies can leverage their business by better understanding relations hidden in data. Trough the analysis of metrics such as user buying history, it's possible to predict which other items users would be interested in, allowing for a personalized recommendation system that offers the user an unique experience and helps it thrive. But theres also information that can be derived from the products or even from shopping trends. Companies can understand which items are bought more during a special season of the year, or they can understand if certain payments seem out of sync with the profile of a certain customer. [Edo14]

In general, big data can help a business thrive by providing a detailed look into it's operations and logistics. It provides a way for the generation of several Key performance Indicators (KPI) related not only to their platform but to the system as a whole. Considering that in this market space it's indispensable to keep innovation a constant factor, big data support has become a necessity.

2.1.2 Big Data at Farfetch

As stated in 1.3 Farfetch main focus is selling high-end fashion products to clients over the world. In order to derive essential insights about it's operations it is paramount that the system supports the collection of all the data metrics produced by their platform. Currently Farfetch's operations uses this data to fuel various end uses such as:

- Their recommendation engine, to give users a custom experience while shopping;
- To collect metrics on their sales and products in order to better understand trends in fashion according to user demographics;
- Understand the platform usage trough click streams analysis.

This allows the company to have information advantage that can be turned into an economic profit. By better knowing their business, Farfetch can reduce costs and increases profits, backing those decisions with data.

2.1.3 How are Insights Derived from Big Data?

For data to be turned into knowledge we first need to input it into our system, but we will look how to do it in detail in section 2.2.2. For now we will focus on the two main components of big data - storage and data crunching. For insights to be created we first need to store data in an appropriate way taking into consideration it's characteristics.

Usually when we talk about the realm of big data, we are not only talking about the size of the data that is stored, but also of the complexity intrinsic to it and the various formats that might be used to describe it. Typical systems store data in relational databases, where each entity of the system is represented by a class and the relations between them are classified in the taxonomy of one-to-one, one-to-many or many-to-many. This types of databases are extremely common in the realm of the Internet, because of their ability to store data relations and allow for simple queries. When used by simple websites they fit the necessary needs, but when the goal of the company is to obtain and extract useful information from that data, it's beneficial to transform it into a more appropriate format for aggregations and complex queries.

The dimensional model proposed by Ralph Kimball [KR13], proposes that we divide our data into facts and dimensions. Facts in this model are tables that represent the numeric values that we wish to aggregate or analyze and the dimensions are the entry points for getting the data to the facts. There are several benefits when compared to the relational model such as [Sah]:

- Understandability: information is grouped into coherent business categories, making it easier to interpret it;
- Query Performance: dimensional models are denormalized and optimized for data querying instead of data storage;
- Extensibility: dimension and facts can easily accommodate new changes.

A database system that supports this design is called a data warehouse and normally, is where the data that enters the system eventually gets stored for further usage. The star schema is the simplest data warehouse schema as it can be seen in figure 2.1. The data warehouse acts as a central hub for integrating data from one or more disparate sources and their main purpose is to create analytical reports to provide knowledge.

Typically, a company will have both the database systems described above, as they serve two different purposes, but most of the time the information stored on the data warehouse will have it's origin from operational databases used by the system. To transform it from one model to the other a process called Extract Transform and Load (ETL) is used. As the name shows, it's a three-step sequential process that encompasses:

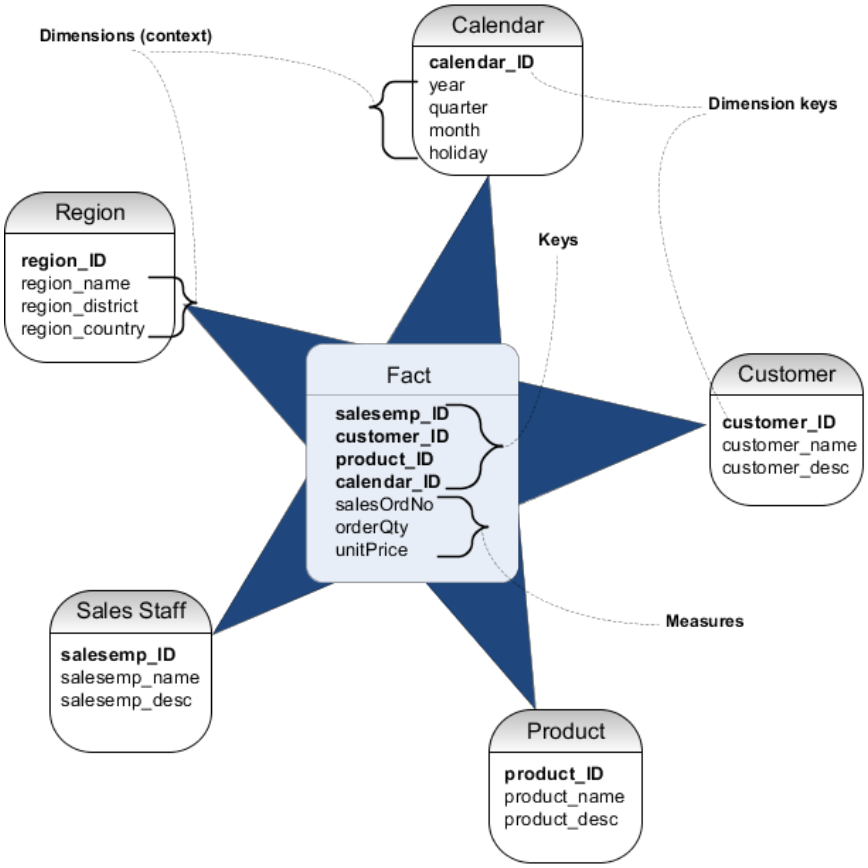

Figure 2.1: An example of a star schema model for a system with orders [Inf]

- Extracting: Data gets extracted from the operational systems, they can be operational databases or other services;
- Transforming: Data gets transformed into the appropriate loading format. New parameters are calculated and data validation and cleansing happens to ensure data quality;
- Loading: Loading the transformed data into the data warehouse.

This process usually occurs in between the data warehouse and the operational systems in a so called intermediate layer as showed in figure 2.2. This layer, called integration is where the disparate data gets integrated before moving to the data warehouse. A usual data warehouse is divided into multiple data marts. A data mart represents a subset of the warehouse oriented to a specific business line, which means that normally the information stored in a data mart pertains to a single department. This isolates the usage, manipulation and development of data per department.

As stated before, having data in the proper format for data warehouses is just a specific way of storing and allowing us to query our data in an easier way to extract relevant statistics. This is especially important and relevant for the business users, since it allows them a proper structure to try and derivate insights, but if we are generating a great amount of data on a short time scale that needs to be analyzed and processed automatically, it's also useful to have the data in a format that

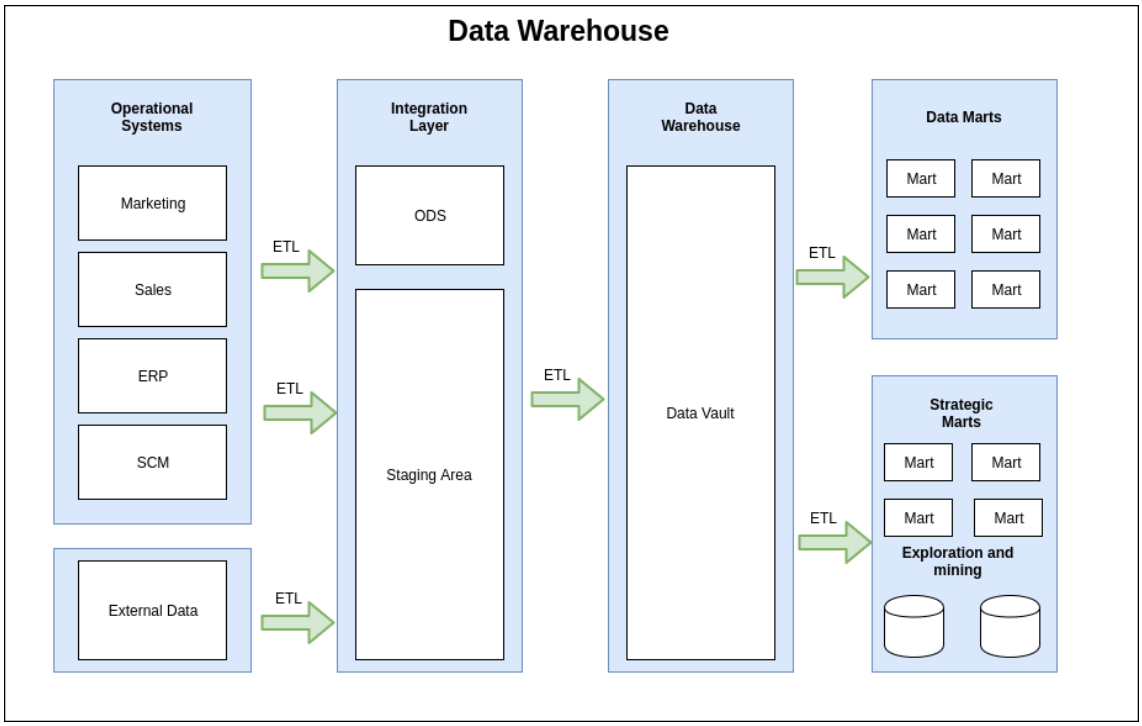

Figure 2.2: Data warehouse overview

facilitates processing. To do this, it's also advantageous to have a system where raw, unstructured data gets paired with structured data, this way we can expand the possibilities of computing by refining the data as we please and increase it's flexibility. In a data warehouse, once data gets transformed and loaded into the system, the initial information that was filtered is gone, so in case there's some additional requirements to be applied to the data, or if we want to use extra fields for a computation, it can only be done to new data that arrives to the system, not old one.

Systems where we store unstructured data with structured one are often called data lakes. The term was coined by James Dixon, Chief Technology Officer at Pentaho [2], as a proposal for a new architecture revolution in big data. He proposed that companies should rethink their systems as data lakes, not only storing the structured data created by the ETL processes and aggregation services but actually also store information in it's raw format [Fan15]. His perspective was that there was a lot of potential knowledge being wasted that could be put to use for business intelligence.

Even if compared to one and other, a data lake can be used as a complement to the data warehouse, since it can offload some data processing work and host new analytics applications as filtering data sets or summarize results that can then be sent to the data warehouse for further analysis by business professionals. As data lakes don't enforce a specific format or schema for data, they usually apply flows of Extracting Loading and Transforming (ELT) when compared to the ETL process used in data warehouses.

Now that we saw the typical possibilities of data storage for big data, we will see how we can

[2] http://www.pentaho.com/

DATA WAREHOUSE	vs.	DATA LAKE
structured, processed	DATA	structured / semi-structured / unstructured, raw
schema-on-write	PROCESSING	schema-on-read
expensive for large data volumes	STORAGE	designed for low-cost storage
less agile, fixed configuration	AGILITY	highly agile, configure and reconfigure as needed
mature	SECURITY	maturing
business professionals	USERS	data scientists et. al.

Figure 2.3: Data warehouse compared to data lake [Dul]

process it to extract knowledge. Due to the scale of the data, one of the strategies used to allow for faster computations and more efficient storage is to split the data across a distributed file system (DFS). A DFS is a cluster of computers that together creates a uniformed file system. Not only this allows for distributed computing but it also for fault tolerance trough replication on multiple machines.

The most known DFS in the big data industry is architected under Hadoop. Hadoop is an open source project from Apache that encompasses multiple projects for scalable, reliable distributed computing [KSC10]. One of the most used of it's projects is the Hadoop Distributed File System (HDFS). Hadoop works like a data lake storage system together with a processing engine called MapReduce. MapReduce [DG04] is a programming model and implementation for processing big data sets with parallel, distributed algorithms on a cluster. Is composed of a Map() procedure that performs filtering and sorting and a Reduce() method that performs a summary operation. Through those operations, metrics can be calculated on the data stored in the HDFS. An example of it might for example be to calculate the average sales value per item type everyday. To do so, what the MapReduce model does is to divide and map the operations to a cluster of computers in the network and then in the end reduce all the values calculated from each computer into one machine with the final result. [Kai]

The processing paradigm into which the MapReduce model falls is called Batch processing. It can be defined as a series of bulk operations that are to be applied on a large dataset without manual intervention. A batch is a grouping of items that are processed together. It's main purpose is to update information typically at the end of the day, generating reports that must complete reliably within certain business deadlines. In batch processing each operation is atomic and must run to completion before the next one starts, being that the batch of data is transmitted as a whole between steps of processing. Batch processing is used in a variety of areas such as transaction, reporting, research and billing. Even though batch processing has it's advantages when considering the

calculation of daily or even hourly insights, it wasn't designed to scale well when dealing with iterative and online processes that need real time processing like stream analytics.

Stream processing is another data processing paradigm that is designed to analyze and act on real-time streaming data. It is designed to handle high volume of data in real time while maintaining a scalable, highly available and fault tolerant architecture. In contrast to batching where data is first stored and only then processed, stream processing processes the data as it streams through the server. Even though it's main characteristic is to allow for the real-time processing of information, it also supports connections to external data sources, in order to enable updates to external databases with processed information for later usage by business intelligence. [Kre]

Most companies today, incorporate the two processing systems together in what is called a lambda architecture, as there is important data that should be stream processed and other on which a daily batch job is suffice.

Batch	**Stream**
Computes a set of aggregated data all together at the time	Computes a function of one data element, or a small window of recent data at the time
Efficient at processing high-volume data	Efficient at continuous input and output of data
Time is not a constraint	Information is needed in real time

Table 2.1: Batch processing compared to stream processing

Considering the 5V's presented in 2.1, when the main concern of a system is the volume, batch processing is a more suitable model, if we also have the constraint of volume together with velocity where results have to be created in a small time frame then stream processing is better.

2.2 Big Data Technologies and Architecture

The big data technologies landscape has evolved extremely quickly in the last years. One of the main reasons has been the need to process information in real time. During many years batch processing was suffice to fulfill the big data needs of most companies, but as technology and hardware evolved, information started to be able to be generated at a higher frequency and areas like the Internet of Things (IoT) appeared. These new areas were dependent of sensor data and needed to react in real time to changes in it. This led to a rise of streaming processing and with it, a big revolution on the architecture of big data.

In this section we will discuss the major technologies being used today in big data and how they fit in the global picture of a big data processing platform.

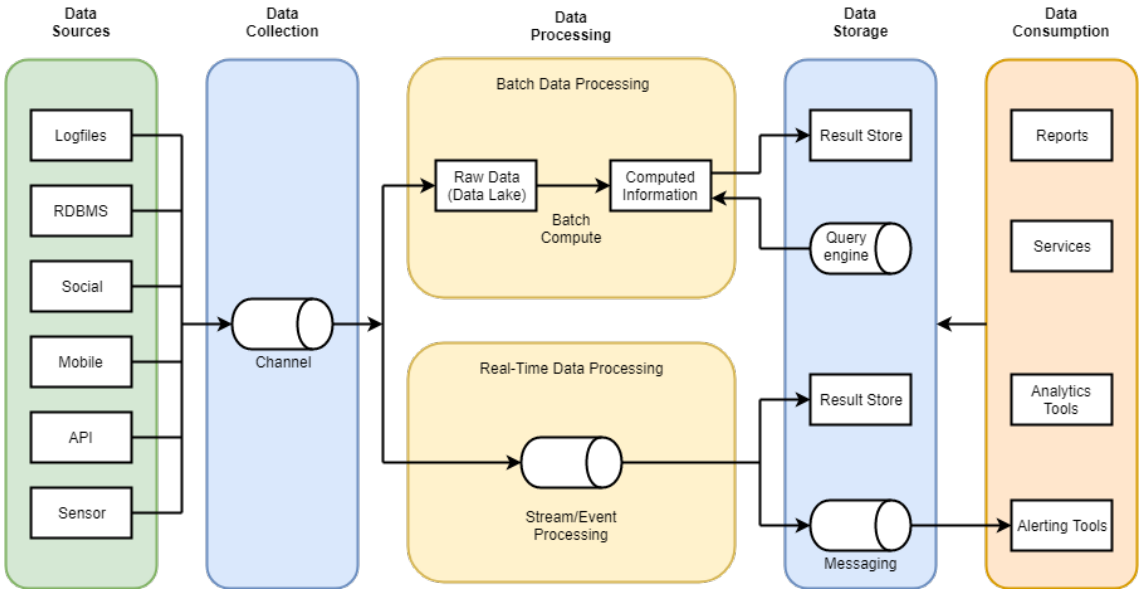

Figure 2.4: Big data lambda architecture

2.2.1 Platform Architecture

Figure 2.4 represents a typical architecture for a big data platform that supports data processing in batching and streaming in parallel. Most of the current systems in production employ a lambda architecture in order to fulfill the different needs of it's operations. [MW15, Sch]

As seen on the first layer, there's several sources types from where information might get collected. Examples of it are logfiles from different machines, mobile clickstreams, social data or even information stored in relational database management systems (RDBMS). This data gets aggregated and ingested into a data collection layer where information might get integrated with other sources before being ready to use. In lambda architectures, this layer is usually abstracted by a messaging queue system taking advantage of the publish-subscribe pattern to be the intermediate component between ingestion and processing.

Once information is ready to be used by the system, it can be ingested into the processing layer. Processing in batch computing, happens periodically. First data is ingested into the data lake, where it stays until an hourly or daily bucket of data gets taken in order to calculate new information. On the other hand, on the streaming platform, data gets processed as it is ingested, on an event-base or using a very small time window between the processing jobs. One of the big differences between the two systems, besides the time frame is the fact that the second one doesn't use an intermediate storage step.

Once data is finished processing, it can be stored in systems like data warehouses or it can be used to feed certain services. Information processed in the data lake can be queried directly and it can also be stored for service usage. Information created by the streaming platform, might use message queues to feed real time services like alert or monitoring ones.

To construct such a system, there is not a single technology that is used but a mix of technologies applied to the different layers of it. In the next subsections we will see an overview of these different layers and the technologies that might be used in each. [Rag]

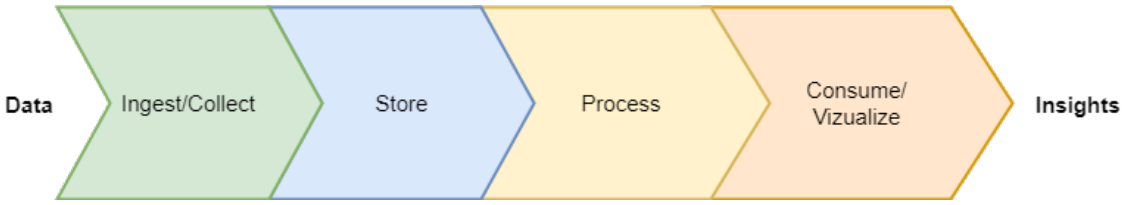

Figure 2.5: Big data processing layers

2.2.2 Data Ingestion/Collection

Data Ingestion/collection is usually the first layer in the architecture. It's responsible for the aggregation of data from various sources to enter the processing system. In this layer, data usually goes under some transformations and cleansing in order to be able to be stored. Data can be stored directly from sources such as applications or sensors, but usually its advantageous to have a structured framework monitoring and controlling data ingestion. There's several technologies that might be used to achieve this, here we present three of the most used.

2.2.2.1 Apache Kafka

Kafka is much more than just an ingestion mechanism, according to it's authors, it's a streaming platform on itself. For the context of this section we will focus on the characteristics that make Kafka a reliable data ingestion/collection framework. At it's core, Kafka supports the publish-subscribe pattern applied to streams of data and in this sense we can assume Kafka to be considered a message queue system. Kafka allows developers to create message queues separated by topic and acts as a broker between the corresponding publishers and subscribers of the system. In an ingestion process, publishers generate and send data to the corresponding message queue. This communication is done using a language-agnostic Transmission Control Protocol (TCP).

The core abstraction provided by Kafka it's the topic, a category to which records are published. Topics are always multi-subscriber as they can have zero, one or many subscribers and for each one, the Kafka cluster will maintain a partitioned structured commit log in which each partition is an ordered, immutable sequence of records to which information is continually written to. The records in this partitions are assigned a sequential id number called offset that serves as an unique identifier within the partition.

One of the big characteristics of Kafka is that it will retain all published records, whether or not they have been consumed, thanks to a configurable retention period. This allows it to have a strong fault tolerance mechanism. Consumers access data in the topic by advancing it's offset

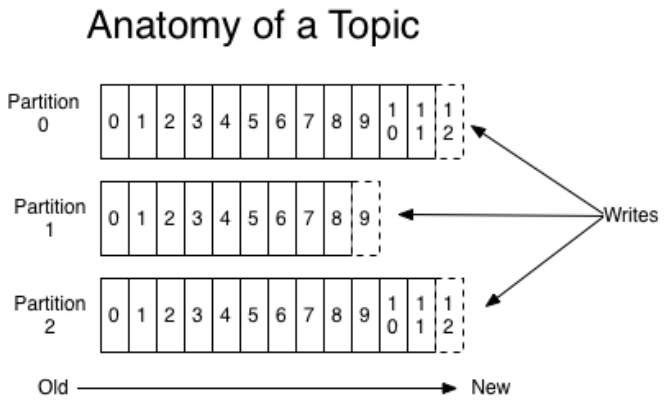

Figure 2.6: Kafka topic's partitions

linearly as it reads the records. Each consumer, has full control of the order it wants to read data, making it possible to reset to an older offset to reprocess data.

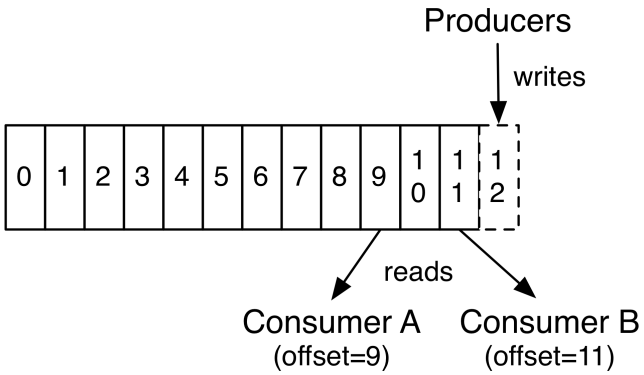

Figure 2.7: Kafka log consumption

The partitions in the log allow the system to scale to a distributed cluster. Each individual partition has to fit on the server where is hosted, but a topic can have several partitions so it is able to handle an arbitrary amount of data. When it comes to it's topology, each partition has a "leader" and zero or more servers which act as "followers". The responsibility of the leader is to handle the read and write requests for the partition while the followers on the other hand, passively replicate the data orchestrated by it. This is done, so that in the case of a failure of the leader, one of the followers can take the place as the new leader. As a way to load balance the cluster, each server might act as a leader for some of its partitions and as a follower for the others.

Messages that are sent from one producer to a topic partition will be appended in the order they are sent. This means that if a first record (R1) was sent by the same producer as a second one (R2), then R1 will have a lower offset than R2. For a topic with replication factor N, Kafka is able to tolerate up to N-1 server failures without losing any records committed to the log.

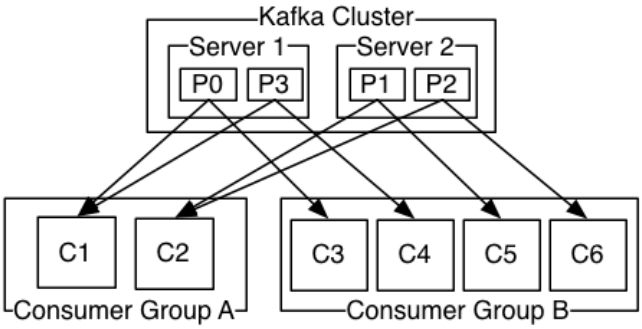

Figure 2.8: Kafka cluster with two server groups

Kafka encompasses two messaging models : queuing and publish-subscribe. In the first one, consumers read from a server and each record goes to one of them, in the second, the record is published to all the subscribers. Queuing allows us to divide the processing of data over multiple consumer instances, allowing for processing scalability, but they don't allow multi-subscriber, which means that once data is read, it's gone from the queue. Publish-subscribe on the other hand allows us to broadcast data to multiple processes, but has no way of scaling since every message goes to every subscriber. Kafka generalizes both these concepts, making possible to every topic to have these two properties.

Besides that, Kafka provides both ordering guarantees and load balancing. This is possible due to assigning partitions in the topic to consumers, so that each partition in a cluster is consumed by only one in the group. With this, we can ensure the data is consumed in order. [Fouf, GSP17]

2.2.2.2 Apache Flume

Flume is a distributed service to collect, aggregate and move large amounts of log data to a centralized storage system, like the HDFS. The basic unit of it's topology is a flume event, which is defined as a unit of dataflow. A typical flume topology is called an agent and it can use different data sources together with event chaining, i.e, the output of a flume agent can be the input of another one, creating a multi-hop flow. A flume agent can be seen as an independent daemon process in a Java Virtual Machine (JVM). It receives events from clients or other agents and forwards it to the next destination. Each agent has the same architecture and it is made up of a source, a channel and a sink.

- Source: The component that receives the data from data generators (there's different type of sources for different events from a specified data generator);
- Channel: A transient store that receives events from the source and buffers them until they are consumed by a sink;
- Sink: Consumes the data (events) from the channels and delivers it to destinations like the HDFS.

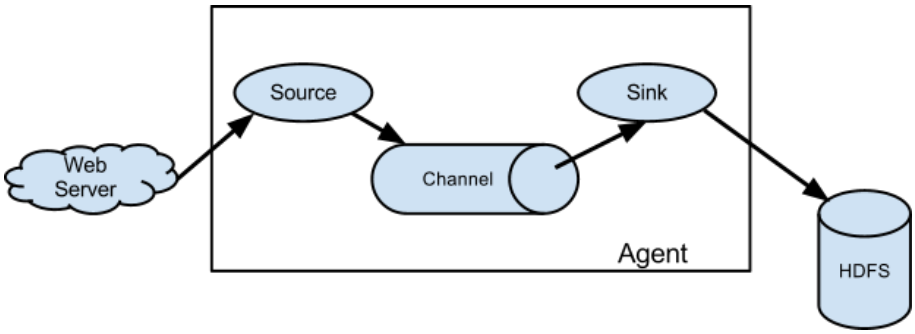

Figure 2.9: Structure of a flume agent

Everytime a source component receives an event, that event is stored into one or more channels of the topology. In this system, the channel acts as a non-persistent storage that keeps the event until it's consumed by a sink and once that happens, the event is removed from the channel. Both the source and the sink work asynchronously with the events staged in the channel and in order to guarantee reliable delivery of the events, Flume uses a transactional approach where both the sources and the sinks encapsulate the storage/retrieval, respectively, of the events placed in it.

Flume also supports fault recovery by having a persistent file channel that is backed by the local file system. Together with it, there's also a memory channel which simply stores the events in an in-memory queue in case of need. If an agent process dies, the events that are left in the memory channel can't be recovered. [Fouc]

2.2.2.3 Logstash

Logstash is a data collection engine with real-time pipeline capabilities. It allows the unification of different data sources and normalization of data into centralized storage systems. It works on top of data pipelines that have three stages:

- Inputs: responsible for generating events;
- Filters: apply modifications and transformations to the event data allowing them to be chained together;
- Outputs: Ship the data to a persistent storage.

There's several configurable input plugins that allow ingestion through file tailing, raw socket/packet communication and others. It lacks a persistent internal message queue, which makes it rely on an external queue like Redis [Red] to assure persistence across restarts. [Foug]

Kafka is many times used in conjunction with either flume or logstash, delegating the log aggregation to them and being mainly used as the transport system. Other than, Kafka is better suited for real-time systems based on events, where information needs to arrive to the subscribers

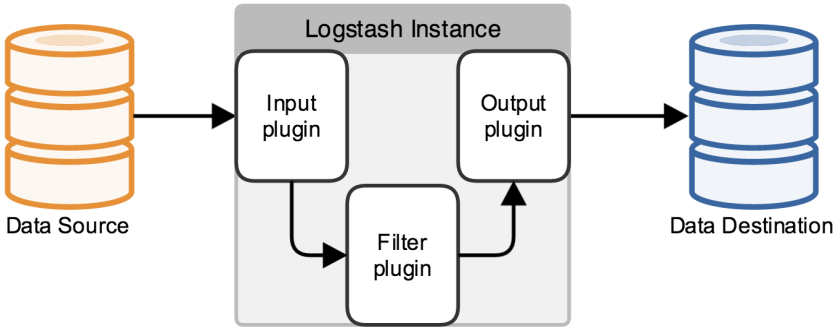

Figure 2.10: Logstash pipeline

as soon as they are published. Flume and logstash are better targeted for batch systems as information might be aggregated for some time before being processed. Due to being part of the Elastic ecosystem, logstash is used more in platforms where other Elastic products are also used.

2.2.3 Data Storage

After data is ingested into the system, if it's batch based, data is first stored before being processed. When it comes to streaming systems, data might be stored in a non-persistent message queue in order for it to be processed as it is ingested. In this layer, some technologies that are usually used are:

2.2.3.1 Apache Kafka

As referred in subsection 2.2.2.1, Kafka can be used in a lot of different layers of action in a big data architecture. As it supports message queues where publishing is decoupled from consuming, it can be used as a non-persistent storage system for data that gets ingested into the system before being processed on the data processing layer. Data that is written to Kafka, is written to disk and replicated for fault-tolerance. It allows producers to wait on acknowledgment on writing to guarantee that information is persisted.

2.2.3.2 Hadoop Distributed File System (HDFS)

The HDFS is a distributed file system inspired by the Google File System [SGL03], used to store large amounts of unstructured and structured data together on commodity hardware, providing very high aggregate bandwidth across a cluster of machines. It is highly fault-tolerant and designed to be deployed on low-cost hardware. An HDFS instance may consist of hundreds or thousands of server machines , each one storing part of the filesystem's data. Typically, a file on the system is gigabytes to terabytes in size. [Foul]

2.2.3.3 Amazon S3

S3 is Amazon response to the need of reliable cloud data storage. An s3 instance works as a bulk repository where data gets stored as in a data lake. It offers a web interface to interact with the data, fault tolerance and scalability mechanisms and can be easily integrated with other Amazon products. It supports data transfers over Secure Sockets Layer (SSL) and automatic data encryption once it is uploaded.

S3 stores data as objects within resources called "buckets". It is possible to have as many objects as wanted within a bucket. Each bucket can be up to 5 TB in size. It includes geographic redundancy and the option to replicate data across regions. [Ama]

2.2.3.4 Apache HBase

HBase is a non-relational, distributed database modeled after Google's Bigtable [FCG06]. It runs on top of HDFS, and it provides a fault-tolerant way of storing large quantities of sparse data, i.e, data where a the majority of values are empty. It is a column-oriented key-value data store. Tables in HBase are typically used as input and output for MapReduce jobs in a Hadoop cluster and in order from them to be accessed, a Java API or another API using representational state transfer (REST) is used. Unlike the relational and most traditional databases, HBase does not support SQL queries, instead the equivalent is written in Java, creating a system close to an object relational mapping (ORM) that allows data to be treated as objects. [Foud]

2.2.3.5 Apache Hive

Hive is a data warehouse software that provides data summarization, query and analysis capacities to Hadoop. It enables access to data stored in various databases and file systems integrated into Hadoop using SQL that can be extended with user code. It's major function is to act as a data warehouse. [Foue]

When comparing these technologies for data storage, there's several advantages from one and other in certain application contexts. Amazon S3 for example, is a better for architectures that extensively rely on Amazon cloud services for their stack. Kafka is better used as a temporary storage system for non-persistent data in an ingestion layer. HDFS is very useful as a data lake to store all information that we might want to process or use in the future, and it's very easily integrated with software such as HBase and Hive to provide additional analysis features.

2.2.4 Data Processing/Analysis

This is the step where data finally gets to be used to generate new information. In data processing, frameworks take the information that was stored into persistent systems and apply transformations to it in order to generate new data.

2.2.4.1 Apache Spark

Spark is a general-purpose cluster computing system mainly used for large-scale data processing. It can process data in a big variety of storage formats such as HDFS and HBase and it's mainly designed to perform both batch and streaming processing. Through extensions of the core Spark framework it also allows for interactive queries and machine learning applications on data. When it comes to it's processing, Spark uses a micro-batch execution model where each batch can be as short as 0.5 seconds while still enabling exactly-once semantics and consistency.

Spark allows for big data processing with less expensive shuffles in the process which is why performance can be several time faster than other big data technologies. To do it, Spark holds intermediate results in memory instead of writing them to disk which is extremely useful when working with multi-pass computations that require work on the same dataset multiple times. It's written in Scala and, as Java, runs on the JVM. Other than the Spark Core API, the are additional libraries that can be integrated and that provide additional capabilities. This libraries include, Spark Streaming[3], Spark SQL[4], Spark MLlib[5] and Spark Graphx[6]. Spark has three main components:

- Data storage: Spark uses HDFS for data storage purposes. It works with any Hadoop compatible data source including HDFS, HBase and Cassandra[7];
- API: Provides developers the ability to create Spark based applications using a standard API interface. It supports Scala, Java and Python;
- Management Framework: Spark can be deployed as a Stand-alone server or used on a distributed computing framework like Mesos [8] or YARN[9].

The core concept in Spark is the Resilient Distributed Dataset (RDD). It acts as a table in a database but it can hold any type of data. Data in Spark is stored in RDDs on different partitions. They are fault tolerant, because they are able to recreate and recompute the datasets in case of a failure and they are also immutable, which means they can't be modified. In order to derive new data from them, transformations need to be applied. Each transformation takes the original RDD, copies it, applies the changes and then returns a new RDD while maintaining the original RDD the same.

From the various libraries that Spark has, the one, besides the core, that interests us in the context of this dissertation is Spark Streaming, an extension that makes it easy to build fault-tolerant processing of real-time data events. The way it works is by dividing a live stream of data into micro-batches of a pre-defined time interval and then treating each batch of data as a RDD. These RDDs are then transformed using operations and it's results are returned in batches that

[3] https://spark.apache.org/streaming/
[4] https://spark.apache.org/sql/
[5] https://spark.apache.org/mllib/
[6] https://spark.apache.org/graphx/
[7] http://cassandra.apache.org/
[8] http://mesos.apache.org/
[9] https://hadoop.apache.org/

are usually stored into a data store for further analysis and to generate reports or to feed real-time services. [Foui, MZW15]

2.2.4.2 Apache Storm

Storm is a distributed stream processing framework written in Clojure used for real-time event processing. A typical storm application is enclosed in a topology that can be compared to a MapReduce job with the difference that the latest finishes once data is processed, whereas a storm topology runs forever as a daemon or until it is killed. Streams are the core abstraction in Storm and they are seen as an unbounded sequence of tuples that are processed and created in parallel in a distributed paradigm. A typical topology consists of:

- Spouts: Source of streams. They read tuples from an external source and emit them into the topology. They can be reliable or unreliable. In the first case, a tuple is replayed if it failed to be processed by Storm, in the second, a tuple is forgotten as soon as it it emitted. They can emit more than one stream.

- Bolts: Processing components. They are responsible for applying filtering, aggregation, joins, databases connections and much more. Typically a bolt will apply a simple transformation to the data. Complex operations are normally the result of a chain of several bolts.

Typically a topology is designed as a Directed Acyclic Graph (DAG) with the spouts and bolts acting as the graph vertices and streams as the edges. [Fouj]

2.2.4.3 Apache Flink

Flink is a streaming processing framework written in Java and Scala, focused on the execution of programs in a data-parallel and pipelined manner. It supports two kinds of datasets : unbounded and bounded. Unbounded represent infinite datasets that are appended to continuously and bounded represent finite, unchanging datasets. The first ones are typically associated with batch processing, while the second with streaming. Flink relies on a streaming execution model, which is an intuitive fit for processing unbounded datasets. It provides accurate results even in the case of out-of-order or late-arriving data and it guarantees exactly-once semantics by check pointing the summary of data that has been processed over time.

It's dataflow programming model provides event-at-a-time processing for both unbounded and bounded datasets. At it's core it consists of streams and transformations. A transformation operation in Flink can take more than one stream as input and produce one or more output streams as a result of it. It already offers built-in source and sink connectors to systems like Apache Kafka and HDFS. Each program runs as a distributed system within a cluster and can be deployed in a standalone mode as well as in YARN or Mesos. [Foub]

2.2.4.4 Hadoop's MapReduce

MapReduce is a processing technique and programming model that allows for the processing of large amounts of data in a distributed system using the batch paradigm. A MapReduce job usually splits the input dataset into independent chunks which are then processed by the map tasks in a completely parallel way. The responsibility of the framework is to sort the outputs of the maps, which are then inputs to the reduce tasks. Generally the input data is in the form of file or directory and stored in the HDFS, being passed to the mapper function line by line and decomposing it into small chunks. The small chunks are then associated with machines in the cluster and once the processing is done, data is reduced into a new set of data that is stored in the HDFS. [Fouk, Whi15]

There's many more processing frameworks currently being used in production than the ones explored in this section, but this ones represent some of the most widely used across the industry. Hadoop's MapReduce continues to be one of the most used batch processing methods. Systems like Spark can leverage the power of the HDFS and provide batch processing together with streaming and for that reason are starting to be widely adopted as the main processing system together with HDFS. Storm can be considered pure streaming, as it processes data per event which makes it very low-latency and well-suited to data that must be ingested as a single entity. Flink, on the other hand, acts as a system like Spark, as it also offers batch and streaming capabilities in a system that has the low latency of Storm and the data fault tolerance of Spark.

2.2.5 Workflow Management

When the entire process is defined and implemented, it's useful to rely on a management tool to define exactly how the worflow should behave in the cluster. These tools are used to schedule jobs and to complement pipelines with other useful tasks that are not bound to the scope of the pipeline but that can be used for facilitating certain operations and to support monitoring and troubleshooting. All of the projects defined in this section deal with orchestrating jobs in the batch paradigm.

2.2.5.1 Oozie

Oozie was one of the first workflow scheduler systems to be widely used in Big Data due to it's ties to the Hadoop ecosystem. It's mainly used to manage Hadoop Jobs and as most of the projects in this area, it relies on DAG's to specify it's control flow. It allows for the combination, fork and parallelization of multiple jobs, allowing users to create complex jobs out of simpler ones. It's main feature is being used to perform already existing ETL operations on data in Hadoop and storing it's result in a certain format. It has visual editors to be able to construct the DAG's, being that the most used one is the one bundled with Hue[10]. [Fouh]

[10] http://gethue.com/

2.2.5.2 Luigi

Luigi defines itself as being a plumber to tackle long-running batch processes, it was created at Spotify to handle an evergrowing necessity of task requirements that weren't built for Hadoop. It can still be used to run Hadoop jobs but also a variety of other tasks like Spark jobs, Python snippets of code, Hive queries and others. It already comes with a toolbox of common tasks that can be used out of the box and has it's own file system abstraction for HDFS and the filesystem. It comes with it's own web interface, which allows for a central visualization of all tasks and for features like filter and searching. [Spo]

2.2.5.3 Airflow

Airflow was created at Airbnb to deal with the increase of complexity in inter-dependencies between data pipelines. It's main target was how to deal with a growing complexity of graph of computations of batch jobs. It has a very rich set of operators bundled with a very clean web application to explore DAG's definition together with their dependencies, progress, metadata and logs. The base modules are very easy to be extended so it promotes the community development of extensions. [Foua]

Even though oozie is still widely used due to most projects dependencies being tied to Hadoop and lot's of ETL projects still being bind to it, the fact that it only allows for Hadoop jobs management has created a need for new frameworks like Luigi and Airflow that allows for the management of workflows in other technologies like Spark. When it comes to the decision between using Luigi or Airflow, it's a question of personal preference and feature compatibility with the company needs as they are very similar at the macro-level.

2.2.6 Data Visualization

After data is finished being processed, it's usually used in different systems that serve data visualization purposes in order to see metrics and healthiness of the cluster and of it's individual processes.

2.2.6.1 Grafana

Grafana is a metrics dashboard that allows to display information from various sources such as Graphite[11], Elasticsearch[12], OpenTSDB[13], Prometheus[14] and InfluxDB[15]. It's essentially used for monitoring ends as it allows for information to be displayed in time series on visual graphs such as histograms, heatmaps, geomaps and others. It allows for the integration with alert services in

[11] https://graphiteapp.org/
[12] https://www.elastic.co/products/elasticsearch
[13] http://opentsdb.net/
[14] https://prometheus.io/
[15] https://www.influxdata.com/

order to keep track of certain thresholds and for the easy extension of the platform with the usage of plugins. [Gra]

2.2.6.2 Kibana

Kibana is a data visualization plugin for the Elasticsearch stack. It's main objective is to allow for the visualization of aggregated information from various sources trough histograms, line graphs and line charts. It also allows for time series analysis and machine learning application to data through the usage of extensions. Is mainly used for monitoring and to extract correlations between data. [Ela]

Both Kibana and Grafana, serve similar purposes when it comes to monitoring and visualizing data, the main difference between them is that Kibana is more suitable for working with Elasticsearch products while Grafana is more widely used for other technology stacks.

2.2.7 Data Monitoring

Monitoring is an area that is very tightly coupled with visualization, as it's the engine that enables information to be fed into the graphical tools. Typically, visual tools read their information from a database, using polls to update their graphs every x seconds. This databases or other persistence systems are controlled by tools that give the user power to define rules, thresholds and alert triggers for the data systems. Monitoring is an essential component of the big data stack as it allows for systems to be kept in check and healthy in order to reduce downtime or failure.

2.2.7.1 Prometheus

Originally created at SoundCloud, Prometheus serves it's purpose as a monitoring and alerting system. At it's core, its constituted by this main components:

- The server: responsible for scraping and storing time series data in a time-based database;
- A push gateway: that supports short-lived jobs like ETL's and that allows data to be pushed instead of polled from HTTP endpoints;
- An alert manager: that allows the users to setup rules and alert triggers that can be chained into different situations.

It works very well for recording numeric time series and it's very suitable for usage with highly-dynamic service-oriented architectures that are constantly producing data which makes it very suitable for big data architectures with multiple and fast throughput sources. [Pro]

2.3 Farfetch's Architecture

As a growing e-commerce platform, Farfetch needs a system that can handle the high volume and high velocity of the data generated by it's platform. In this section we will look at the architecture designed to handle this problem, together with a detailed overview of it's main supporting components.

2.3.1 Architecture Overview

Farfetch's current system implements a lambda architecture, with a batch processing system in parallel with a streaming one. For this, the team is divided into two functional parts:

- Messaging team: is responsible for consuming data in real time from a Kafka cluster that then feeds the HDFS and Hbase tables. It's main focus is in creating streaming jobs with spark streaming to read data constantly and make it available either to HDFS for later processing by the big data team or by directly processing it and making it available in Hbase tables or SQL server to be used by the data science and business intelligence teams;
- Big data team: is responsible for managing the big data infrastructure which encompasses the data lake and all the services that feed on it. It's also responsible for creating most of the ETL pipelines that process data in the HDFS to make it available in other operational systems.

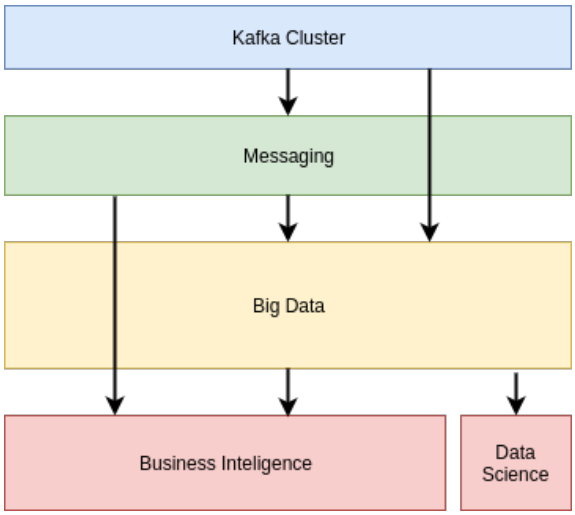

Figure 2.11: Data teams interaction in the BI cluster

In picture 2.12 we can see an overview of the lambda architecture employed by Farfetch. At the top we can see that data gets aggregated and collected by Logstash as it gets produced by different services such as clickstreams and external API's and concurrently, information like transactions and viewing history gets stored in a operational Cassandra database. Both this systems

feed Farfetch's Kafka cluster that is a global service that is used to serve different teams. In the big data cluster, the messaging team has a local Kafka cluster that reads from the global Kafka and creates the processes that are responsible for the validation and assurance of the data quality before it gets written to the HDFS or other systems. This way, all data that gets processed by the big data team, is already confirmed to be valid. Currently due to the existence of various legacy platforms and systems still in use, the processing of data is still divided by the two teams, but the intention of the company in the future is to transfer all the responsibility of data processing to the messaging team and make the big data team only responsible for infrastructure.

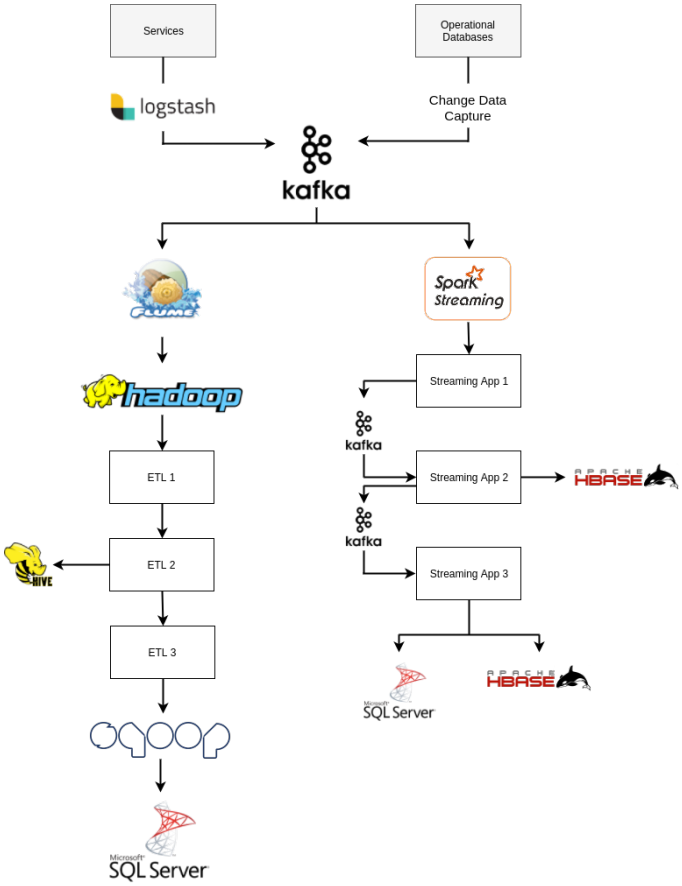

Figure 2.12: Farfetch lambda architecture

Analyzing the picture into more detail, in the left side we have the batch processing system where data is gradually stored in the HDFS from multiple source systems in production using flume. In the HDFS, hourly or daily batches of data are collected into buckets to be processed by several ETL processes using Spark and Hadoop's MapReduce. These processes are orchestrated using oozie and airflow. It is possible that in between these processes, intermediate representations of the data gets stored in systems such as Hive, as the team might want to use the intermediate data for other purposes or make it available to other teams. Once the batch process is finished, Sqoop[16]

[16]http://sqoop.apache.org/

moves the data from HDFS to the data warehouse for the business intelligence team to be able to use the newly created information for insight generation.

Parallel and aiding the batch system in dealing with the data complexity, data that needs to be processed in real time, gets trough the streaming system on the right side of the picture. In it, data gets processed using Spark Streaming and follows a sequence of transformations steps before being written to a persistent layer. This system is currently mainly used as a buffering system of the big data team, however, is also used to do some direct processing on some data that needs to be available in real time to other teams. Because in a streaming job, pipelines are always constant and are always running, they don't need any kind of orchestration. To be run they are usually submitted to the spark cluster using spark-submit.

2.3.2 Dataflow

Dataflow represents the path that data follows once it is ingested into the system until it arrives to a final persistent layer. In the case of the architecture presented above, if data is batch processed then the dataflow is sequential and data gets processed in blocks from start to finish in the same way everytime until it is finished. On the other hand, on a streaming system because we are dealing with data in real time, dataflow can be more complex as information from multiple sources might have to be merged before being processed, promoting a parallel processing of the dataflow as opposed to the sequential model followed in batching. Even though there are differences, pipelines in both the systems still present a a very similar structure, having the same macro-components but requiring different structuring.

2.3.2.1 Pipelines

Every time a new sources is added to the cluster or every time a new business requirement arises, a new pipeline is created or modified. In this sense, a pipeline is the basic unit of work done on the big data cluster and it's behavior relies on the underlaying dataflow specification. For every new pipeline, a developer needs to write a script that specifies the dataflow that the pipeline should be responsible for. A typical pipeline independently of the paradigm in which is inserted consists of:

- One or more sources: data is always ingested from at least one source. It can be from HDFS, from a Kafka message queue or any other storage system;
- One or more processing steps: after being ingested, data suffers transformations in order to generate new information which is done by using Spark, MapReduce or Spark Streaming;
- One or more sinks: after data is finished being processed it is written to a persistence storage, according to it's nature and future usage.

2.3.2.2 Workflow Orchestration

As the number of pipelines increases, it's important to have a system that allows for the visualization and control of most of them in a central manner. This guarantees that all processes are

started from the same place and in the same way. This is not only helpful to see what processes are active and working correctly in the cluster, but also to setup certain essential steps as creating warning triggers or error strategies for pipelines. At Farfetch, Airflow and oozie are used to manage the pipeline orchestration of already existing batch pipelines i.e after an engineer writes a new dataflow pipeline using some set of technologies, they use these platforms to setup a flow of how this process should behave in the cluster. Due to their nature, and even though it would be possible to also do orchestration on streaming jobs, this setup is only used for batch jobs. The reason is that streaming jobs are by nature constant processes that are responsible for bringing data in real time to other systems, if we want to calculate some metrics on the data we can do it on a batch approach every hour or so. Some of the properties that are setup in airflow and oozie are:

- Schedule intervals (hourly/daily) of the execution of batch jobs;
- Metrics endpoints for process and cluster monitoring using Prometheus;
- Slack warnings and email alerts triggers in case of certain scenarios (p.ex deployment success/ process failure).

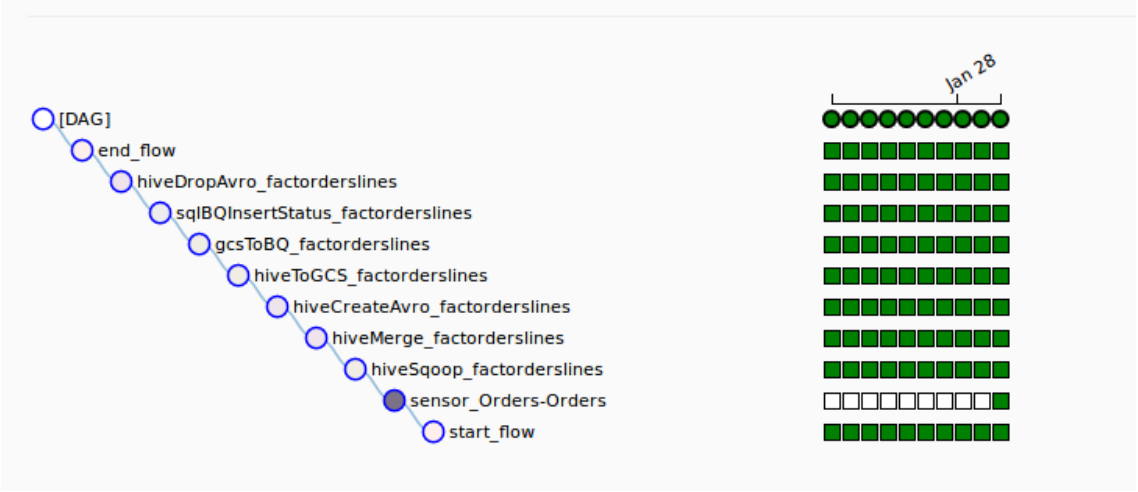

Figure 2.13: A oozie worflow DAG used at Farfetch

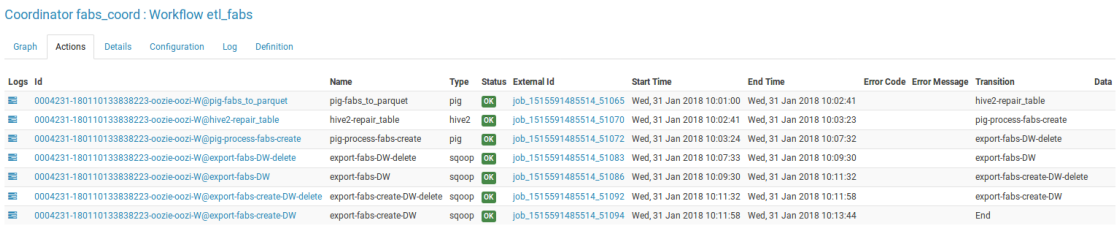

Figure 2.14: Oozie dashboard with several jobs

2.3.2.3 Monitoring

Currently, process monitoring at Farfetch is done using Prometheus and Grafana for the processes that are orchestrated and using the Spark streaming user interface (UI) for streaming jobs. Each process that is setup with airflow creates an endpoint that is consumed by Prometheus. Prometheus then sends this metrics to it's integrated time series database that is used to poll information in intervals to Grafana. Prometheus PushGetaway is also used to receive information and general metrics from ETL jobs in a Push manner. The AlertManager is then used to setup certain thresholds and rules that if reached, should be used to alert the big data team through messages and visually on a dashboard that is currently displayed in a screen. Grafana then feeds on this timeline database to construct graphs of how many jobs are currently up on the cluster and what is the their current state. On the Spark job UI, its possible to see information of how many messages are being processed by second and the current state of the jobs that are up.

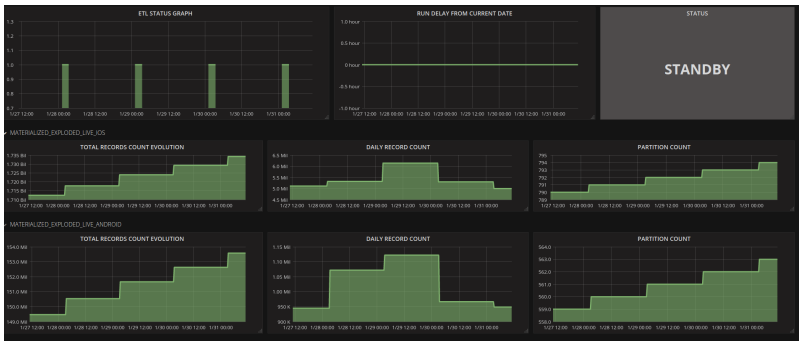

Figure 2.15: The grafana dashboard displayed in the big screen at Farfetch

2.4 Conclusions

Big data is experiencing a high-speed evolution. The need for processing information in real time has been shaping new technologies and has been creating the need to improve the current system architecture of many companies that have established themselves online. In this chapter we saw some of the ways companies like Farfetch are leveraging the power of new frameworks to allow data to be processed and the way they are using that information to fuel essential services for their operations. We analyzed how data is stored in data systems in order to be evaluated and used by business analysts and we also discussed in detail the architecture layers of a big data cluster, with special emphasis on the one currently used at Farfetch.

The lambda architecture currently used at Farfetch, has been slowly adapting to the needs of the company, but theres still room for improvement when it comes to the way dataflows are defined in the pipelines. As seen by the current pipeline setup system, there's already a set of good practices when it comes to the workflow orchestration of the jobs that are run, but when it comes the way that the dataflow is setup for each pipeline, there's still a lot of common layers

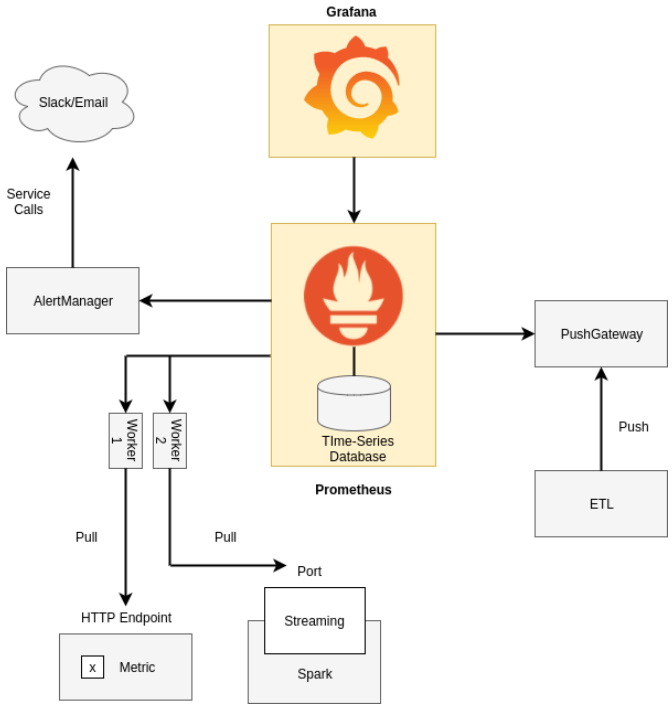

Figure 2.16: Monitoring architecture used at Farfetch

that get repeated by different processes and that get developed by different people or even teams. It can become unpractical and extremely cumbersome to have to deal with all the new pipelines from scratch without a concise common model to support it. Not only that, but in a fast growing area like Big Data, it's important to experiment and pivot with new technologies that can be more suitable for certain scenarios, so it's important to have a flexible model that thinks how current processes might be affected in the future by other technologies. By analyzing the current state of the art of technologies, together with the current system in use at Farfetch we believe that is possible to improve this process to be easier to setup by the developers.

Chapter 3

Problem Statement

This chapter presents the current problems and difficulties faced by the big data cluster in their current approach to dataflow orchestration, together with an overview of improvements that can be applied to it and a list of functional requirements to implement into the solution. Section 3.1 introduces an overview of what data orchestration encompasses and some reasoning about it's necessity. Section 3.2 reviews how the current teams working in big data at Farfetch define and handle the data complexity involved in specifying the flow of data. In sections 3.3 and 3.4 we analyze in detail the problems that arise from the current process and discuss improvements to it. To conclude the chapter, section 3.5 presents a list of functional requirements that we defined for the solution.

3.1 Dataflow Orchestration

As stated in 2.3.2, dataflow defines how data moves from one component to the other inside a processing pipeline. A typical big data system usually presents a large number of processing jobs, each one representing their own pipeline that depends on several components from the cluster. These pipelines might be fundamentally different, but they are all based on the same ingest-process-write model. Their main difference is on the complexity of the flow. A simple process might simply be to read from one source, apply a set of transformation to it's data and then write it to HDFS - this is the typical scenario when looking at a dataflow of a batch processing job. An example of it might be calculating the total sale value of a given day, at the end of it. On the other hand, a more complex process might read data from various sources that produce data at different rates and have to merge this data before processing it and that then write to several sinks - this is more of a scenario related to a streaming pipeline. An example of it might be to update the recommendations of a user, after the visualization of an item. Even though the complexity of the pipeline changes between these two cases, the underlaying model stays the same.

Orchestration, in this context, refers to managing this flow in a structured and defined manner i.e, how do we define flows like the ones specified above to be run. The same way that a typical workflow is managed by a tool like oozie or airflow, dataflow should also be managed in an abstract way that doesn't depend on technology specifics. The worflow management platforms allow us to define a set of steps to be run in a certain order, but they don't take in consideration the construction of the pipeline as it doesn't deal with the flow of data between components, they are only worried in executing processes in a black-box manner. Due to this, it's not possible to define dataflow on these tools, but they show us how dataflow could and probably should be managed. Because every pipeline has different needs when it comes to it's flow, controlling and managing the dataflow of different jobs becomes extremely cumbersome and time-consuming as a platform scales. As the complexity of data integrations and transformations grows, it becomes very hard to manually manage the flow of data that integrates one process as well as making sure that the current job execution is being correctly processed. Due to this, dataflow orchestration presents itself as a big necessity in big data.

3.2 Current System

Farfetch's current infrastructure doesn't support dataflow orchestration as a separate process from it's implementation. What is done currently is that once theres a need for a new processing job, the flow is defined by the team responsible for implementing it and then that job is delegated to one of the big data engineers. This person is then responsible for creating a processing job that in the code script encompasses all the necessary transformations done to the data, together with the steps necessary to read the input data and write the output data. So in this sense, there's no orchestration of the flow of data outside of the code script, each process is responsible for defining and managing itself from start to finish.

3.3 Problems With the Current System

Delegating the responsibility of the dataflow orchestration to each process, creates several problems:

- Dataflow coupling: dataflow is always bound to it's own technological implementation, being dependent on the specifics of the technology used;
- Difficulty managing complexity: since every process is different theres no common model we can use to allow standardization;
- Difference in quality standards: because processing jobs are often written by different people, it leads to differences in the pipeline construction with differences in code quality;
- Rise of engineering costs: due to the differences in the pipeline construction, it's much harder to pinpoint the source of a pipeline failure, making it difficult to solve problems;

- Resistance to change: difficulty to prototype new technologies with old processes due to strong dependencies;

3.4 How Can it be Improved?

As stated in subsection 2.3.2.1, pipelines always present the same macro-level structure. They might have more or less layers of components and the number of components might vary, but each pipeline is always composed of sources, processors and sinks. Most of these components that read and write data are common between many pipelines, what varies from one to the other it's the configuration of which topic to read from or what table to write to, but the underlaying implementation it's always the same. There's several pipelines, for example, using Kafka or HDFS components as their source and even though we might read from different topics or different locations, the code that is responsible for reading the data and ingesting it into the pipeline is mostly the same between processes. The same goes for components responsible by writing, like Hbase. When it comes to the transformations applied to the data by the processors like Spark, most of the transformations are the same and don't depend on data schema, which means we can also generalize them independently of their structure across processes. Based on this, there's some improvements that could be applied to the current system:

- Decouple dataflow from it's implementation: by providing an abstraction layer to the way dataflow is specified in pipelines, we can guarantee that independently of the concrete technological implementation of it, we can manage all of them in the same way;
- Abstract components to a higher level: this way, some of the repeating code that is shared across pipelines could be leveraged and reused across processes;
- Provide a unifying system to create and manage dataflow: by having high level components, every developer could rely upon a unique system to define the dataflow of new pipelines into the cluster, providing standardization of processes;
- Allow developers to only focus on the component specifics definition, leveraging the components communication and the flow of data: by creating a system responsible for the way components communicate, developers could concentrate in only specifying the characteristics of a component, and relying on a system that deals with the rest of the complexity of the dataflow;

3.5 Solution Requirements

In order to establish concrete objectives for the implementation of the solution and to establish a defined scope to narrow the range of possibilities, a requirements collection was done to identify the crucial features that should be accomplished based on the problems stated in 3.3 and the

overview of possible improvements described in 3.4. The following is a list of functional requirements to be implemented by the solution :

- **FR01 - Source Configuration**: The user should be able to configure components that act as sources of data, together with it's properties, in order for it to be used by the system;
- **FR02 - Processor Configuration**: The user should be able to configure components that act as processors of data, together with it's properties, in order for it to be used by the system;
- **FR03 - Sink Configuration**: The user should be able to configure components that act as sinks of data, together with it's properties, in order for it to be used by the system;
- **FR04 - Pipeline Configuration**: The user should be able to configure pipelines by specifying the components, previously defined, that compose it in order for the system to know the respective dataflow of a pipeline;
- **FR05 - Read Data From Sources**: The system should be able to read data from the configured source components in order for it to be used in the specified pipelines;
- **FR06 - Apply Transformations to Read Data** : The system should be able to apply transformations to data previously read from sources using the processor components configured previously in order to enable data's processing;
- **FR07 - Write Processed Data**: The system should be able to write the processed data to sink components previously configured in order to allow for the pipeline's persistence;
- **FR08 - Component Communication**: Components should be able to pass data and communicate between each other independently of using different concepts in order to allow component interoperability;
- **FR09 - Component Concurrency**: The system should be able to support concurrent access to the same components in order to allow different pipelines to re-use them;
- **FR10 - Pipeline Parallelism**: The system should be able to support parallel executions of different pipelines in order to allow multiple pipelines to be run at the same time;
- **FR11 - Component Parallelism**: The system should be able to use two or more components of the same type parallel to each other in order to support more complex pipelines;
- **FR12 - Record Data Lineage Between Components**: The system should be able to support data lineage through the usage of metadata information between the components of the system in order to provide a record of the path of data;
- **FR13 - Support Multiple Technologies**: The system should be prepared to work with different technologies, independently of their implementations, in order to guarantee flexibility;

- **FR14 - Application Logging**: The system should support logging of the actions executed in order to help debugging error scenarios;
- **FR15 - Pipeline Fault Tolerance**: The system should be able to recover from errors that might arise from the pipeline execution in order to allow for the system not to fail;
- **FR16 - Pipeline Monitoring**: The system should support the monitoring of the correctness of it's pipelines in order to allow it's user to see if everything is working as intended;

3.6 Conclusions

Analyzing how dataflow orchestration is currently handled at Farfetch, it was possible to identify several points that can and should be improved in order to allow the big data cluster to correctly handle it. The current system creates too many drawbacks when considering the scalability and flexibility need that comes with exponential growth of the number of processes to support the cluster. There's the clear lack of a standardized and controlled way to define and manage the flow of data and for that reason, it becomes crucial to create a solution to tackle this problem and allow developers to have a proper system that leverages a big portion of the common work used between processes, mainly the component definition and communication in order to support data orchestration. For that reason, we believe that the requirements identified in section 3.5 will allow us to implement a feasible solution to this problem.

Chapter 4

High Level Overview

In this chapter we present an overview of the underlaying architecture created to solve the problems stated in 3.3 taking in consideration the needs of the functional requirements in 3.5. The work presented here is agnostic of specific technologies and provides an abstract specification of the main features that compose the architecture in order to make it's implementation possible. In each section, with the exception of 4.1, we present a feature and the reasons why it was taken in consideration for the architecture design.

4.1 Framework Overview

In order to solve the problems we stated in 3.3, it was necessary to devise a global solution that could be used by every big data engineer to define and orchestrate the dataflow of a pipeline. Taking in consideration the improvements we defined in 3.4 and the functional requirements in 3.5 the envisioned system is a framework that abstracts the creation of new pipelines into it's own components, leveraging it's own communication system and data abstraction to support interoperability and flexibility between technologies. This way, we can decouple technology from the dataflow specification, while allowing for the configuration of new data pipelines in an agnostic manner. By also providing an extensible architecture, we also allow engineers to extend the technologies supported to it's own needs, promoting common modules and reducing duplicated codebases. With this framework, end users only have to worry about the specification of the pipelines they want to create, leveraging the technological part to the framework. With this, we can achieve the promotion of a common system to orchestrate dataflow and create new pipelines, eliminating most of the drawbacks of having different people writing different pipelines, while also having the control to extend and support new features. As we can see in picture 4.1, the usual flow assuming the use of the framework is the following:

- New requirements for a new pipeline arrive and with them a new pipeline configuration is created;

- This configuration file is then passed to the framework where it's information is parsed;
- Internal representations of the components to be used are created, and a new pipeline is created with them;
- A pipeline is then run using already existing elements on the big data cluster;
- The monitoring service feeds on an exposed endpoint.

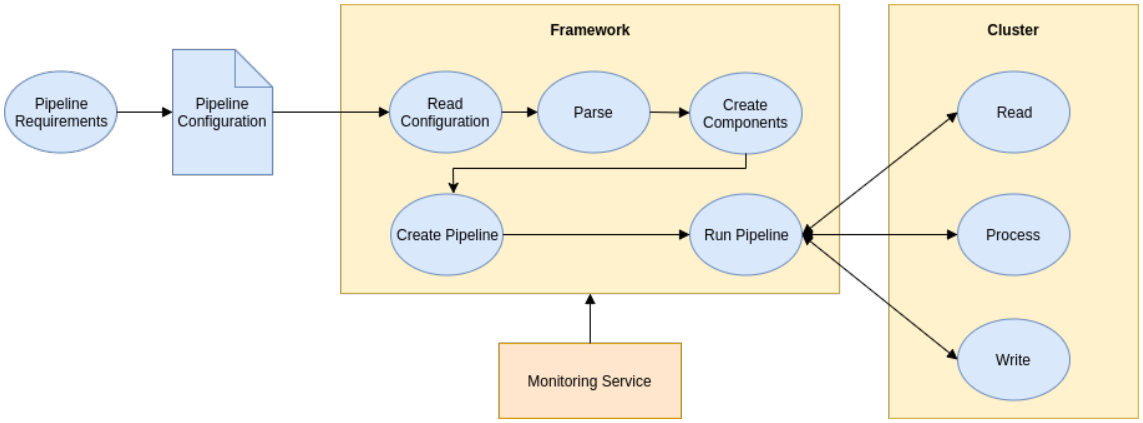

Figure 4.1: High-Level overview of the framework behavior

In the same fashion as the workflow orchestration platforms, the first draft of the solution to be implemented was a flow-based programming approach that would allow for the visual setup and visualization of new processes using a model-driven definition to interconnect them, but due to complexity of that solution and considering the time constraints imposed by the dissertation period, it was not feasible to achieve such a system in such a short period of time. It was then decided that the system envisioned could be divided into two big components - the underlying framework described above and the flow-based setup that would allow it's control. This thesis is focused on the definition and implementation of the first part, but took into consideration the second one in it's architecture decisions in order to allow future work to be done on top of it.

4.2 Dataflow Components Abstraction

To be able to support multiple technologies independently, we have to be able to treat them all in the same manner. For that reason, it was necessary to abstract them to a higher level representation. Instead of having for example a Kafka source or a RabbitMQ source, both of them are just considered as sources for our framework. The same applies to processors and sinks. To do this, we enforce a common API to be implemented by each component that gets included in our architecture by forcing it to extend a base interface. This means that each component, no matter it's technological implementation can be used in the same agnostic way in our framework. The classes that extend that API can then be used as proxies to the real implementation allowing us to write

all the necessary code we need to make them compliant to the specified API while maintaining the proxy as clean as possible to be used by the rest of the architecture. The rest of the framework can then interchangeably call a defined method on a Kafka proxy or a RabbitMQ proxy, without needing to know what's the underlaying technology.

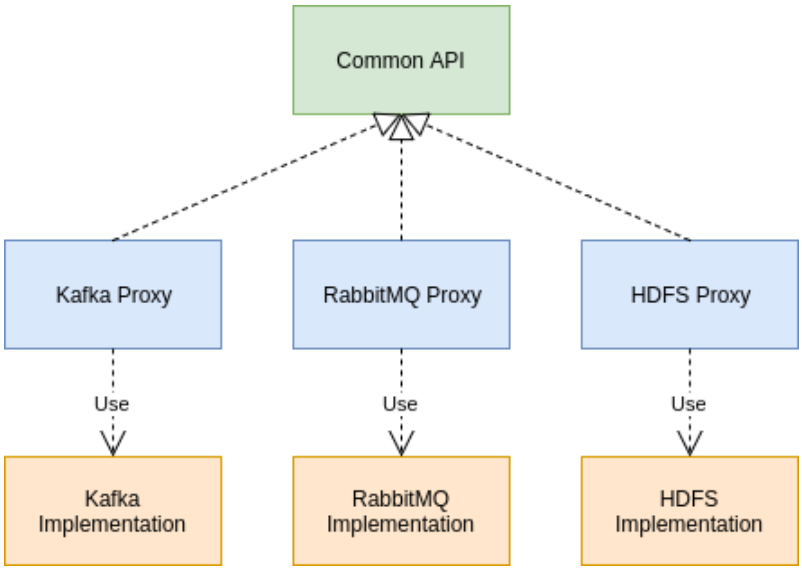

Figure 4.2: Example of the source component abstraction

Everytime an engineer wants to extend the set of supported technologies of the framework, it just needs to implement it once and set the corresponding proxy to call the methods to work with the common API specification.

4.3 Dataflow Configuration

When it comes to the configuration of the dataflow, it was necessary to support a system that was able to not only be used in a bash oriented way but also to support the possibility of extension to a visual flow-based component system. Since in a visual representation, there would always have to be a persistent format to be able to read/save created graphs, the decision was to use a file configuration. In this file, the user will be able to specify the characteristics of each component that he wants to use - characteristics like the Internet Protocol (IP) address of the machine to connect to and it's port, the topic to read, location to write in between others. Besides that, it will also allow to specify how the pipeline is constructed from those components. To do so, we decided to follow a configuration format very similar to Flume that has the following structure:

```
1 app.<component-type>.<component-name>.type = <component-technology>
2 app.<component-type>.<component-name>.<property-name> = <component-property-value>
3
4 app.pipelines.<pipeline-name>.<component-type> = <component-name>
```

Listing 4.1: Component and pipeline configuration definition

```
<word> := ^[A-Za-z]*
<number> := [1-9]

<component-type> := "sources" | "processors" | "sinks"
<component-name>, <pipeline-name> := <word><number>
<component-technology> := <word>
<property-name> := ^[A-Za-z\_]*
<component-property-value> := ^[A-Za-z0-9]*
```

Listing 4.2: Grammar definition

The two listings above represent an overview of the definition of the dataflow together with the grammar that supports it. What we want the user to do is to basically specify which sources, processors and sinks he wants to use in a certain application together with it's respective properties and then specify the construction of the respective pipelines, assembling together the components. The framework will then be able to create an internal representation of the components specified and know how each pipeline should be run. This way, we separate the pipeline definition from it's components definition.

4.4 Data Abstraction

In order to achieve complete interoperability between high order components, we also need to create a system that allows for all data to be treated equally. Different components might read and process data in different formats. To solve this, we created a common data abstraction, that not only allows components to be compatible between each other, but that also allows us to create a data model that we can control and reshape in order to our needs. This means that we can establish a set of characteristics standards when it comes to properties like compression while also enriching our initial data with metadata between components. To do this, we will abstract data into a class that will contain as a property the real data into a generic stream of bytes, together with a data lineage tracking system that we will explain more in detail in section 4.6. This way, everytime we make data pass from a component to another, we can send and receive data under the same abstraction.

4.5 Components Communication

With the components sharing a common data abstraction and being able to be interoperable, we can now define how the communication between them should be defined. However, there's several problems we need to take in consideration when designing this part of the system:

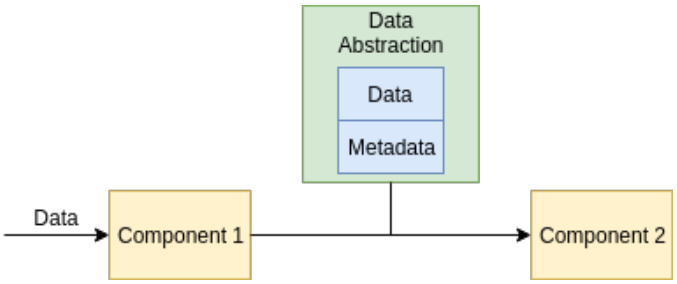

Figure 4.3: Data abstraction layer

- Asynchronism versus Synchronism: in a lambda architecture, both batch and streaming jobs are supported, so our system needs to take in consideration that components can be of these two natures when defining communication;
- Component Concurrency: components might be re-used by different pipelines in one application. This leads to the necessity of supporting concurrent access to the same components by different processes;
- Pipeline Parallelism: multiple pipelines can be defined in a same application that need to be run parallel to each other;
- Resource Management: since we will be dealing with big amounts of data and concurrent work in components it's important to do a smart management of the available resources in order to avoid deadlocks and component starvation;
- Communication Protocol: in order to promote standardization it's important to have a defined API to communicate;
- Error Handling: errors might arise during the communication of two components, so the frameworks needs to be able to manage and solve those situations in a reliable manner.

Assuming the asynchronous and concurrent nature of the communication between components in the framework, we decided that a feasible solution to this problem would be using an Actor model to define our architecture. In an actor model, every component is considered an actor, a self-contained process that has a set API that it uses to communicate with other actors. Each actor has a defined behavior based on what messages it expects to receive. They are usually arranged in hierarchies, having one or more actors that are responsible for overseeing the program and creating child actors that represent smaller and more manageable tasks of the system. The parents can then act as supervisors, allowing them to define failure strategies and other general definitions of a subsystem. Due to this organization, the system can support multiple subsystems that can be aggregated according to their functional purpose.

Using this approach, each component of the system can be seen as an actor where we define specific messages that act as the communication API. By separating the components into each one

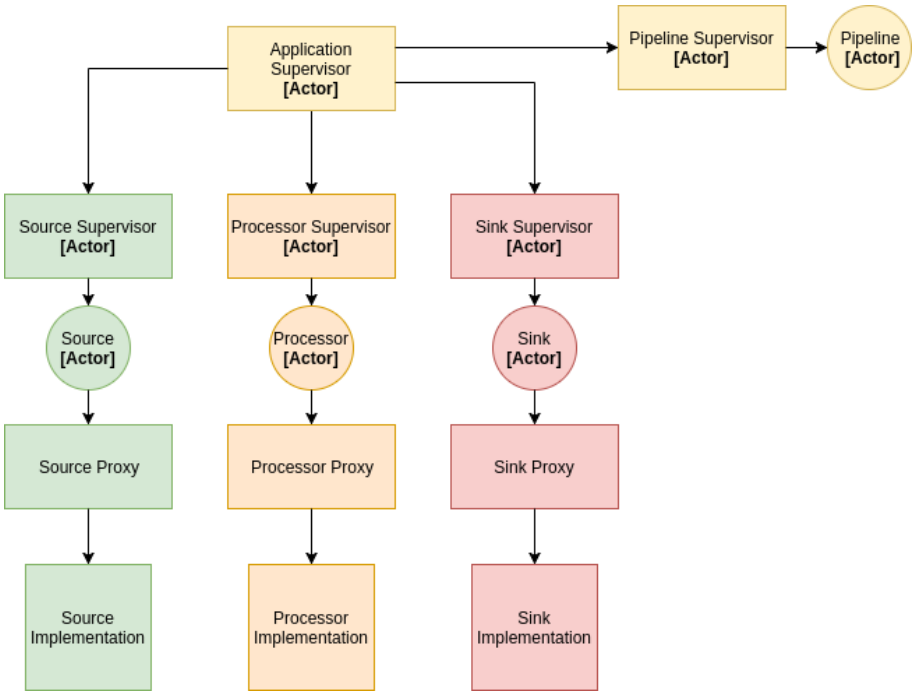

Figure 4.4: Actor organization overview

of the functional areas - sources, processors, sinks - we can create subsystems where we can have supervisors for each one them. These supervisors will have as responsibilities:

- Creating child actors for each one of the components specified in the configuration, based on it's component type;
- Acting as a broker to leverage communication between actors of different actor subsystems;
- Leveraging and manage the error handling scenarios of it's children;

Since each actor is self-contained, and assuming it's implementation in a system that supports multi-threaded components, the actor can self-manage it's own thread pool, making sure to delegate it's work efficiently with the resources available. By coupling this with our own data and component abstraction, each actor can be responsible for:

- Receiving messages from others actors and executing functions on the proxy implementation of it's component type;
- Return the data location of the data encapsulated into the data abstraction layer together with the respective component stamps that gets persisted using the system explained in 4.7.

This, together with queuing on the actor, allows us to solve the problems of concurrency, resource management, communication protocol and error handling. To solve the problem of multiple pipelines, we decided to create an actor responsible for the pipelines, that sets up and runs

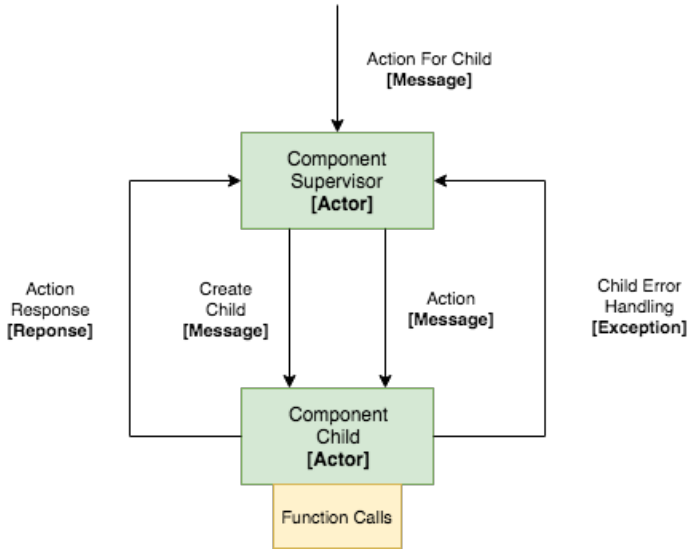

Figure 4.5: Supervisor and child actors interaction

child actors for each one of them. The reason behind this is that we want to re-use the already existing components that make up for the pipeline and at the same time we want to have individual control over each one. Each one of these child actors will run the necessary steps for achieving the wanted flow of data in a sequential manner. Sequential only in the aspect of the overall flow of the process, but parallel for each component type i.e if one flow we have the necessity to read from multiple sources, each source is read parallel to each other, but only passes to the processing step once the reading is finished, hence the sequential definition.

With this, we are left with the last problem of how to read from asynchronous and synchronous components simultaneously. To do this we will use a Futures/Promises approach coupled with the messaging system of the actors in the pipeline child process. Futures are just a vessel for values that are initially unknown and that usually need to be waited on for their computation to finish. This means that in the sequential part of the code, we can handle the parallel call to the proxy implementations by using Futures, collecting them in the end and managing asynchronous and synchronous processes seemingless.

4.6 Data Lineage

Data lineage can be defined as the data life cycle history, allowing us to understand where it came from, where it went and what happened to it's content along the path. It's an essential part to support traceability and analytics of data. Having control over the component definition inside of the framework and having our own data abstraction coupled with a persistence layer, we can very easily support data lineage across components. In order to do this, we created a Stamp System that "stamps" data across the different steps of a pipeline. A stamp is characterized by:

- A Timestamp: to indicate the moment the stamp was created;

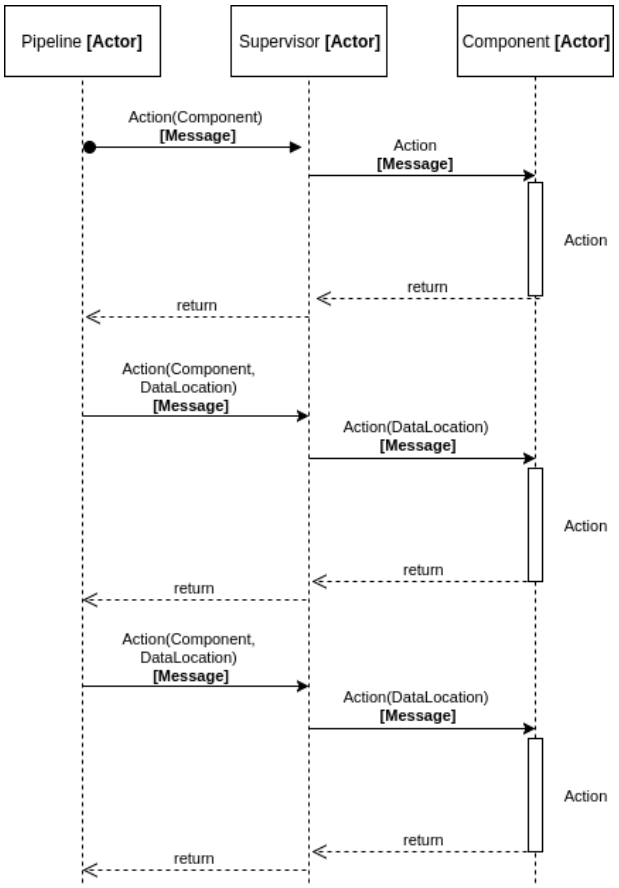

Figure 4.6: Sequence diagram of a simple pipeline

- A Stamp Type: to differentiate the processing steps that compose a pipeline;
- A Signature: to know the component responsible for originating that stamp;
- Volume: number of bytes of the data processed in that step.

Everytime data gets in or leaves one of the system components, a new stamp is added to indicate the beginning/end of that step. That data together with the stamps is then persisted, being that this stamps are a property of the data abstraction layer. With this we can have full traceability of the data that gets processed in the system. Even though we didn't implement an instrumentation tool, the data lineage supported could easily be complemented with one that would allow for the visualization of the state of the current processes, probably coupled with the visual tool proposed as an extension of this work.

4.7 Persistence Layer

In the first draft of the actor system defined in the previous sections, the return messages sent from one actor to another used to incorporate the entire data abstraction and send it throughout

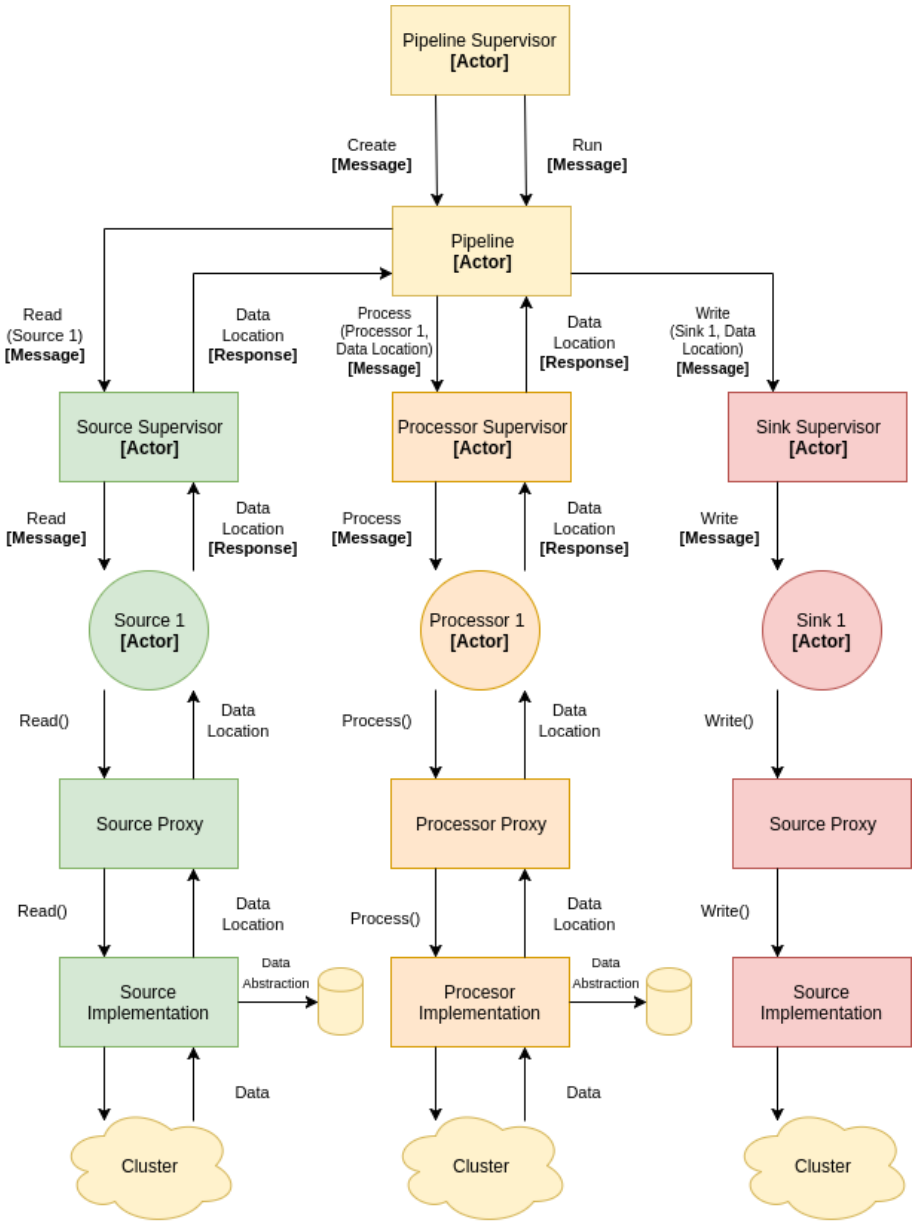

Figure 4.7: Pipeline actor flow overview

the system until the flow ended. This meant that we actually got a lot of memory usage on parts of the system that didn't use the data due to the layered architecture of the components. Not only that, but in case of process malfunction that might require reprocessing of the data, having the data in memory would be a point of failure. For those reasons, we added a persistent layer after data is read/processed and encapsulated into the data abstraction layer. This way, we serialize data by storing it in a persistence layer that could range from a kafka topic, to HDFS or filesystem. After data is written, we return to the above layers the data location instead of the data in itself. This way we minimize data transfer costs, but increase the possible I/O costs in case we have to read it from disk. In the end, in order to minimize storage costs of temporary storage of this data, we use

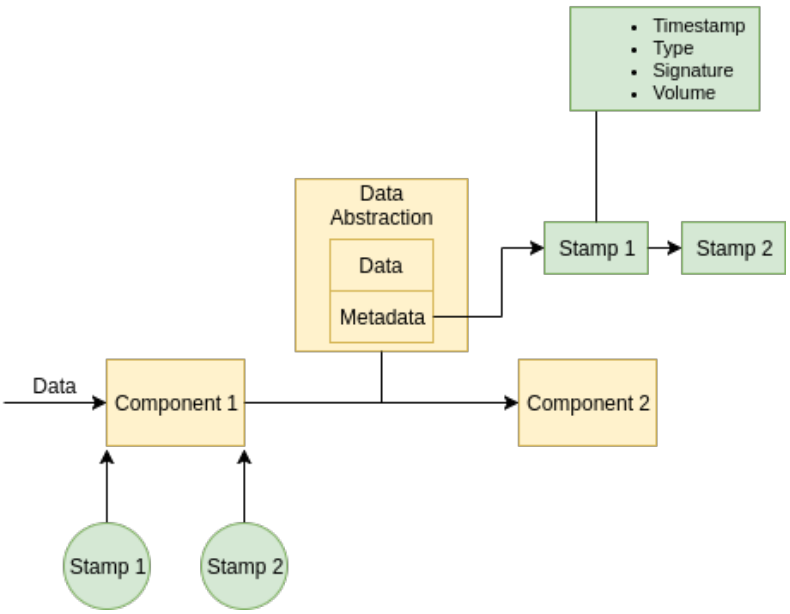

Figure 4.8: Data lineage system

a scheduled cleaner to erase data at every defined time interval.

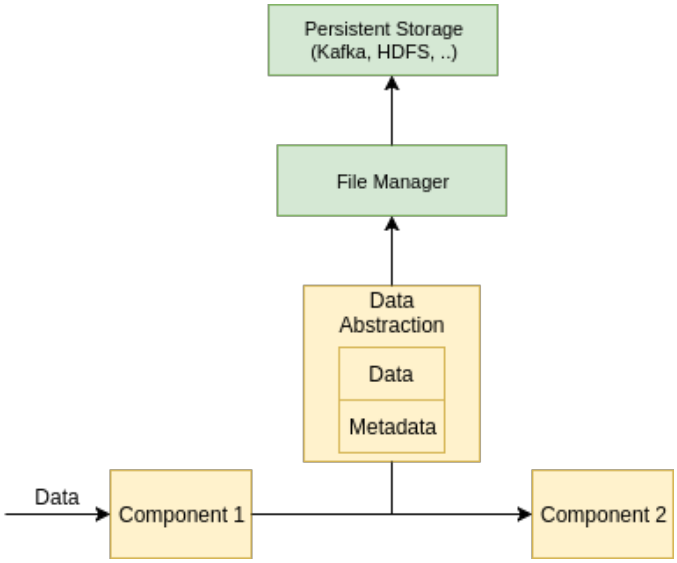

Figure 4.9: Persistence layer system

4.8 Logging

In order to promote a transparent and reliable way to collect the behavior of the system, our framework supports a logging mechanism for every action of the framework, either for real-time usage or to serve later analysis in error scenarios. To do so, in the same manner we use the stamp

system for data lineage, at every component of the system, we store information of the current flow in the process. Everytime a new component is started, accessed for computation or finished, we store that information under the form of a message with the respective information in a log file of the filesystem. A new file is created by each new application run and old files are deleted after a certain period of time.

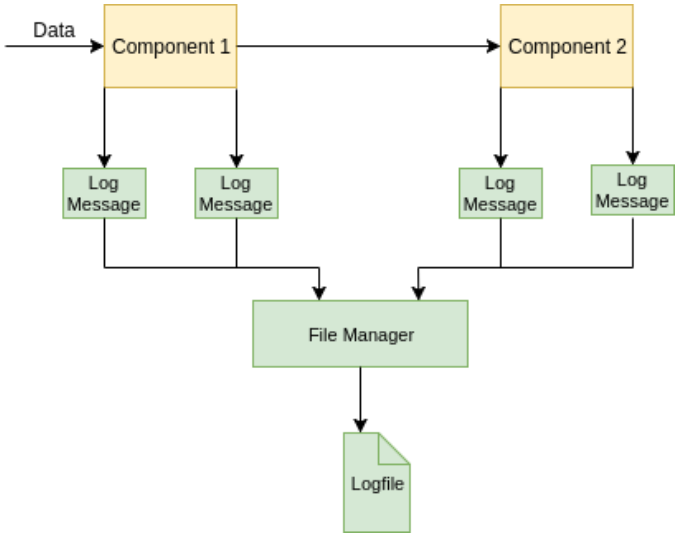

Figure 4.10: Logging system

4.9 Service Monitoring

Since the framework doesn't encompass a visual module, one of the ways to still allow for the verification of the correctness of the pipelines is to integrate it with an already in use monitoring service. Services like Prometheus coupled with Grafana allows us to consume an exposed HTTP endpoint that can be used for metrics evaluation. By having a logging system, we allow the system to be transparent about information such as the number of reads per minute, volume of data processed in a pipeline or the number of errors encountered. By coupling it with module that allows us to serve a GET endpoint we can serve the results that are constantly being logged to it and use it to feed the service. This way, if we also want to change the usage of the endpoint to another purpose, we can easily integrate it with our own solution, or extend it for the already supported ones.

4.10 Process Scheduling

Since most of the defined dataflows need to be re-run continuously in a certain time interval, our framework needs to support the concept of process scheduling. In order to do so, we decided to couple a dispatcher to each pipeline actor in the system. What this means is that the user should

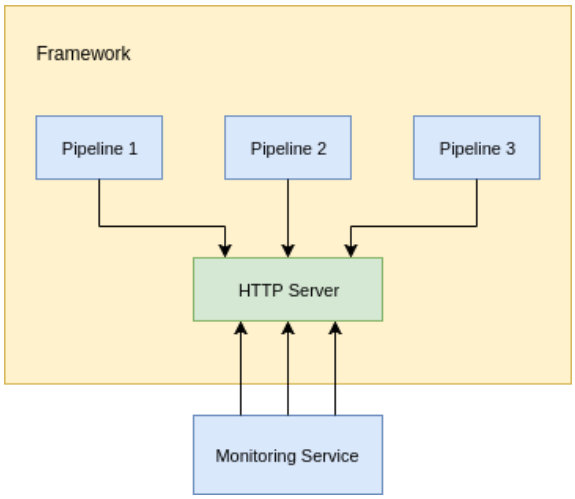

Figure 4.11: Monitoring system

be able to set as a property of each pipeline the periodicity at which he wants the pipeline to be run, with this value, the framework will then be responsible for re-launching the process.

Figure 4.12: Dispatcher system

4.11 Limitations

Even though we took in consideration some of the major decisions when it comes to the architecture definition of the framework there's still some points that were not covered:

- Priority between same type components: the current architecture reads or processes all the components of the same layer in a parallel and concurrent way, there's no possibility of specifying that a certain component should be run first than another;
- Schema validation: data is all processed in the same manner in the framework, there's no system to allow for the verification and validation of the schema on read;
- Operator connection configuration: the connections are only set when the definition of the pipeline is done, there's no support for the individual configuration of the properties of a source to a processor, for example.

4.12 Conclusions

In this chapter, we got to see an abstract overview of the high-level architecture of the framework created to solve the problems currently faced by Farfetch when it comes to dataflow orchestration. The reason to present it in such a way first, it to allow for the decision making of the architectural decisions to not be bound to any kind of technology or implementation bias, letting it be taken in consideration for future extension or different implementation than the one explored in the next section. As stated in section 4.11 there's obviously other concerns that could have been taken in account when designing the framework, but due to time constraints couldn't all be satisfied. However, the decisions explored, discuss the main points that the architecture should support and allows us to focus on the implementation of a proof of concept with the presented functional requirements.

Chapter 5

Implementation Details

In this chapter we will present into detail one of the possible implementations approaches for the proof of concept of the architecture defined in the previous chapter. We will look at the technology decisions and the reasoning behind them in 5.1 and 5.2 together with the component breakdown and definition of several parts of the framework in 5.3. In the last section, we will present the details of the implementation of each one of the functional requirements specified in 3.5.

5.1 Scala and the Akka Framework

Scala is a statically typed, general purpose language that is a superset of Java. It offers a mix between the object-oriented and the functional programming paradigm while promoting immutability at it's core. It runs on the JVM and due to it's collection API and functional nature is widely used in the big data community, examples of it being the Spark project, that is mainly written in Scala. Akka[1] is the implementation of the actor model on the JVM. It provides a framework to build resilient and distributed applications based on reactive concepts that are built upon an actor architecture. It supports up to 50 million messages/second on a single machine and is used in production by companies like Amazon and Paypal. When considering technologies for the implementation of the underlaying architecture of the framework we tried to focus on ones that had strong support from the big data community and had a proven record of being used to solve problems in this area. Not only that, but also technologies that were familiar and used at the big data cluster at Farfetch. For those reasons we chose Scala as the language of implementation of the framework since most of the engineers write their processing pipelines in it. Akka, on the other hand, even tough not used in the cluster, presented itself as the most resilient implementation of the actor system that we envisioned for our architecture, allowing us program on top of it and very easily leverage it's robust communication infrastructure to our needs.

[1] https://akka.io/

5.2 Configuration Parser with Typeconfig

For the definition of the configuration file we opted for a human friendly JSON superset that took in consideration the grammar defined in section 4.3. The reason for the choice of this format is the flexibility provided by it, allowing us to easily shape the configuration between formats like YAML and JSON seamingless which in turn allows us to support different parsers. To support it, we decided to use Typeconfig[2], a type-safe configuration library that supports several JVM languages and allows us to easily parse a nested configuration taking in consideration data types. Besides that, it also supports schema validation of the configuration.

5.3 Component Breakdown

Figure 5.2 presents an overview of the main component interactions in the proof of concept implementation. The implemented code is divided into three packages:

- Core: the main components that make up for the actor system and actor interaction together with it's main abstractions;
- Utils: helpful components that leverage work in the framework but are not directly bound to it's implementation and could be easily replaced by alternatives, as for example the file manager for data persistence;
- Extensions: the specific technological implementations of the sources, processors and sinks used in the framework together with it's proxies.

As stated in section 5.1 we are using Akka for the actor model abstraction. In Akka, all actors exist inside the **user guardian** that is created when the program is initialized. This acts as the father of all the sub-systems of the program. Together with it two others parent actors are created as it's children : the **/user** for all the future created actors by the program and **/system** for internal necessities of the actor system. To create new top-level actors that are direct children of **/user** we can call **system.actorOf()** specifying as an argument a class that extends the Akka Actor Class and a name to reference the actor. The system actors can then create children with **context.actorOf()** with the same arguments as the previous function. In our framework, all managers are created as direct children of system and it's children as context actors of the managers.

Each actor can then be referenced by it's name in it's subsystem. In each Actor extended class we implement a **Receive()** method that is overridden and in it we define the behavior for the messages that the actor expects to receive. Messages in Akka can be seen as the definition of a communication protocol in a request-respond manner. Using the Scala API with Akka, we can define messages as case classes and use it's pattern matching feature to match with the expected cases in the **Receive()** method. The Akka actor API exposes lifecycle hooks that can also be overwritten by the actor implementation allowing us to specify certain behaviors on different steps

[2]https://github.com/lightbend/config

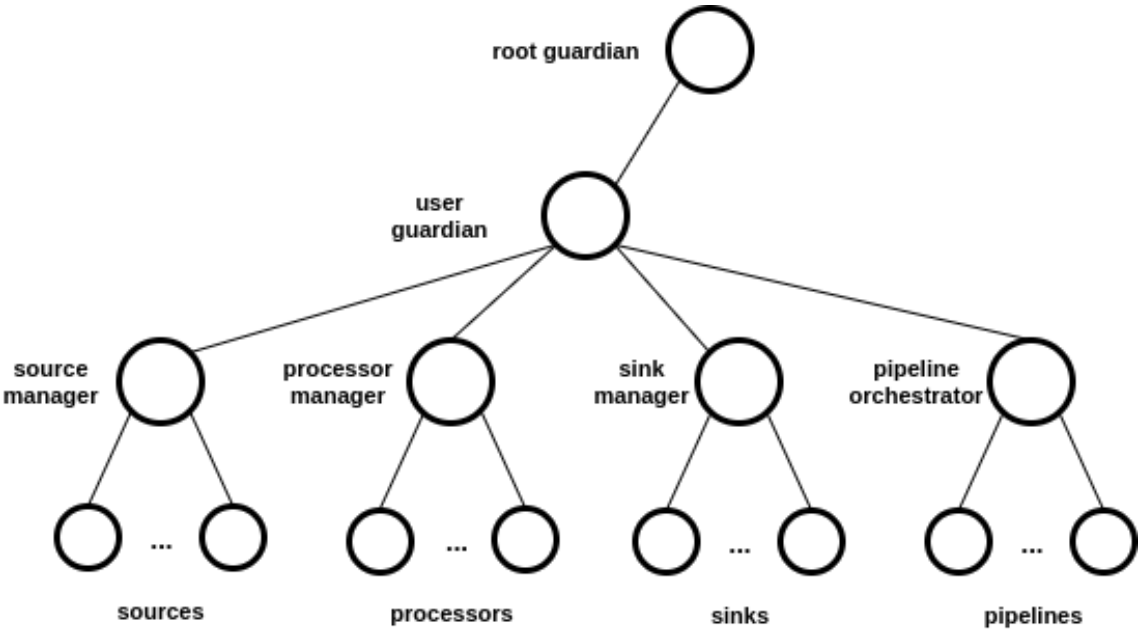

Figure 5.1: The framework akka tree of actors

of the actor lifecycle as for example **PreStart()** and **PostStop()**. We can also define supervisor strategies in the children actors that specify the behavior in case of error or exceptions allowing for fault tolerance scenarios where the it's father can be responsible for the error handling. Message-ordering between two actors is also guaranteed by the Akka system i.e if two consecutive messages are sent by one actor and in the middle another actors sends another message, we still get the guarantees that the first message sent from the first actor will get received first than the second one from that same actor. This, in combination with the built-in actor logging system allows us to build an application log. Akka also has built-in support for futures using the Futures library from Scala together with it's own operator **Ask - ?** that allows actors to call services that are Future based. At the end of the actor lifecycle, either when it is stopped by error or by using **context.stop()**, all of it's children are also recursively stopped, promoting easy resource cleanup.

When it comes to the object oriented architecture of the framework, we defined traits to enforce the common API between component type. **PipeSource**, **PipeProcessor** and **PipeSink** are all traits that get extended by the technological implementations in the extensions package. By extending the trait, we can abstract the creation of the elements to a factory.

Each component in figure 5.2 is numbered in order to allow for it's caption and explanation:

- **(1) core.Main**: The entry point for the framework. It receives the location of the application.conf trough the command line and sends it to the **utils.ConfParser** for parsing. It creates the actor managers and the **core.PipelineOrchestrator** in the akka actor system using **system.actorOf()**. After having them created, it then sends the **Create** messages to the managers with the respective created component and component name for the creation of the actor responsible for it. It's also responsible for sending to the **pipelinesConf** to the

PipelineOrchestrator to setup the pipelines that are to be run and to send the **Run** message to start the processes;

- **(2) utils.ConfigParser**: A singleton object responsible for reading the .conf file and using typeconfig to extract it's values to variables. It calls the factory methods to create the components specified trough it's type property. It also parses the pipeline configuration and creates a Map of **PipelineConf** that will be used as the definition for the pipelines flow;

- **(3) core.SourceFactory, core.ProcessorFactory, core.SinkFactory**: The component factories. Their responsibility relies upon creating the components that implement the components traits with the specifications in the .conf file and returning the component objects proxies to be used in the Actor system.

- **(4) core.PipelineOrchestrator**: The actor responsible for creating the pipeline children actors using the **pipelineConf** passed from the **core.Main** and for sending the starting **Run** message to initiate the pipeline flow;

- **(5) core.PipelineActor**: The actors that are responsible for the execution of the flow of each defined pipeline. They are responsible for sending a **Read** message for each source in a pipeline to the **SourceManager** and then use it's answer to send the **dataLocation** through a **Process** message for each processor to the **ProcessorManager** (the same for the **SinkManager**);

- **(6) core.SourceManager, core.ProcessorManager, core.SinkManager**: The actors that create the component actors that will be responsible for handling the components created by the factories. They are also responsible for routing the requests to them, acting as brokers and by handling their runtime errors.

- **(7) core.SourceActor, core.ProcessorActor, core.SinkActor**: The actors that manage the proxies and allow them to leverage the actor system to handle concurrent communications;

- **(8) Source, Processor, Sink**: The proxies of the technological implementations that are called by using the defined API methods of the shared traits;

- **(9) core.DataAbstraction**: The serializable class that encapsulates the data in order for it to be sent across different component types together with the stamps used for data lineage;

- **(10) extensions.FileManager**: A singleton object that is responsible for saving to disk the data abstraction objects and for reading them from disk to memory. It creates a Globally Unique Identifier(GUID) for every write. It also cleans data older than two days from disk everytime the program is initialized.

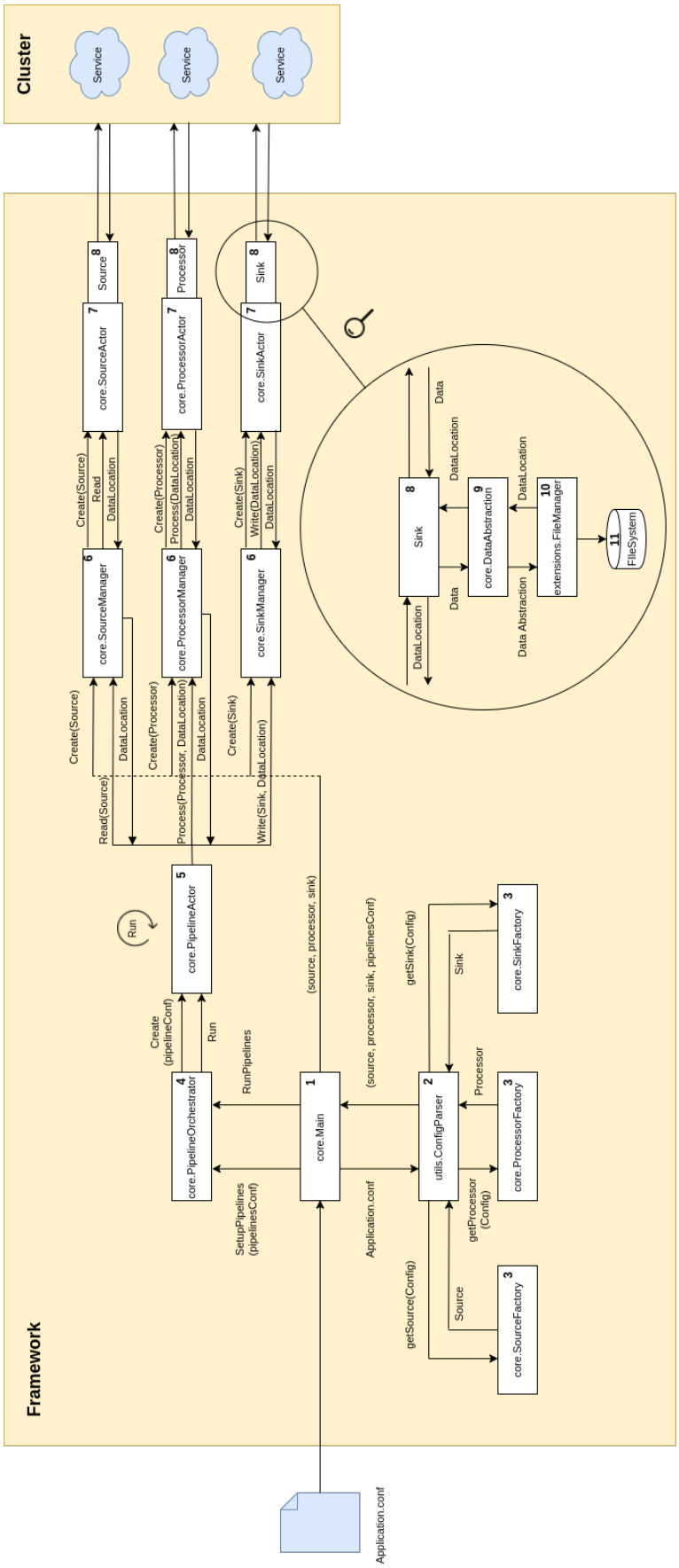

Figure 5.2: Framework components interaction

5.4 Requirements Implementation

In this section, we present the details of the implementations of each one the requirements specified in section 3.5 taking in consideration the components implemented and defined in the previous section.

5.4.1 FR01, FR02, FR03 - Source, Processor and Sink Configuration

All the configuration requirements are made possible by leveraging the typeconfing framework used by **utils.configParser** coupled with object-oriented programming principles. The parser allows us to read the property fields of each component as a tree object that we can traverse looking for already specified keys such as *sources*, *processors*, and *sinks*. The *sources* object, for example, can then be traversed in order to individualize it's children, each one representing a different source specified in the configuration file. Each one of this children is then passed to the respective factory that checks it's **type** property and creates the appropriate subclass of the respective father trait. Each one of the subclasses knows the respective properties of that type of component. After the parser finishes, it returns a **Map[String, <fatherTrait>]** of each type to the main program where the string represents the name of the component and <fatherTrait> represents the object that was created by the respective factory.

5.4.2 FR04 - Pipeline Configuration

The main differences between the pipeline and the components configuration is the fact that we don't have different types of pipelines and that a pipeline is not an object that gets instantiated. We still parse the respective information for each pipeline in order to know the components that compose it, but with that information we create a **pipelineConf** object that is just a blueprint of how the data should flow in that pipeline and that is then used later for the pipeline construction. We then return from the parser a **Map[String, pipelineConf]** with all the parsed pipelines configuration.

5.4.3 FR05 - Read Data From Sources

In order to read from the sources, not only do we need to implement the technological details that effectively are responsible for it as extensions of the framework, but in the core of the application we also need to create an abstraction layer that supports it, independently of the technology used. To do it, we implement the **PipeSource** trait that forces the method **read()** to be implemented by all the source subclasses. These subclasses then act as the proxy to the real implementation and allows us to always call the same method on different implementations of the source trait. The **core.SourceActor**, direct child of the **core.SourceManager** is the one responsible for handling the reading and it does so by receiving and acting upon a **Read** message that is sent from the manager. To send the message, the manager uses the **ask()** function from Akka that expects a **Future** as response.

5.4.4 FR06 - Apply Transformations to Read Data

In order to support any processing engine and still make use of the data component abstraction that allows us to treat all the processors in the same manner, we decided that we should enforce beforehand what transformations can be applied to the data. This means that the **PipeProcessor** trait should enforce, through inheritance, the possible transformations functions that can be applied to the data and force all the implemented processors to implement those in conjunction with the function **process()**. That function creates an entry point into the processor in the same manner that read() did for the sources. This might reduce the flexibility of the possible transformations we might wish to apply to our data, but it also allows us to setup a set of common transformations that can be applied by just referencing a string in the configuration file. The **core.ProcessorActor** reacts on **Process** messages that are sent by the **core.ProcessorManager** and is responsible for calling the **process()** function on the proxies which will use the other defined functions by the processor class. This part of the system also uses the **ask()** pattern to send messages to the child actors and receives a **Future** as an answer.

5.4.5 FR07 - Write Processed Data

To write data to different persistent layers, we followed the same strategy as for the sources, but making the **write()** method be implemented by any subclass of the **PipeSink** trait. The **core.SinkManager** then sends **Write** messages using **ask()** to **core.SinkActor** that then call the proxy implementation and return a **Future**.

5.4.6 FR08 - Component Communication

Components are able to communicate by using the actor system coupled with the data layer abstraction. In order to enforce the flow of the pipeline, the **core.PipelineActor** uses the previously defined **core.PipelineConf** to setup the sequential calls to the respective components that are part of it. After routing the reading of data to one or more **core.SourceActor** using the **core.SourceManager** it's responsibility is to pass the **DataLocation** of the stored information to the respective **core.ProcessorActor** that using the common abstraction **core.DataAbstraction** knows how to deserialize the class and get the data to be processed. The same system is used to then pass the processed data to the **core.SinkActor**.

5.4.7 FR09 - Component Concurrency

Concurrent access to the components is one of the inherent characteristics of the actor system leveraged by the usage of the Akka framework, allowing each actor extended from the framework to have a message queue to deal with concurrent requests in order of arrival.

5.4.8 FR10 - Pipeline Parallelism

In order to allow the framework to support multiple pipelines simultaneously we create a **core.PipelineActor** for each one of the **PipelineConf** parsed from the configuration file. To do so, we use the **core.PipelineOrchestrator** that is responsible for receiving a **SetupPipeline** message to create each one of it's actor children and then the message **RunPipelines** to initiate all of their execution.

5.4.9 FR11 - Component Parallelism

To support a multi-component layer in a pipeline, for each component of that type in the **PipelineConf** we send an **Ask()** request and wait for the response as a **Future**. In order to aggregate the futures result, we use **Future.sequence()** which gives us as a result a **ListBuffer[String]** with the multiple data locations that were produced as a result of the calls. In order to then allow the next layer in the pipeline to correctly handle the data, we do a merge of the data of the several data locations of the previous layer i.e if we read from different sources, before processing it, we merge the data from all those sources into one single **core.DataAbstraction** component by using the **core.FileManager**.

5.4.10 FR12 - Record Data Lineage Between Components

To record the flow of data inside of the framework we use the stamp system proposed in section 4.6. Every proxy component is responsible for calling the **utils.StampMarker** object and passing the respective data to be stamped together with the wanted metadata to be recorded. This metadata is then serialized together with the respective data using the **core.DataAbstraction** and **utils.FileManager**.

5.4.11 FR13 - Support Multiple Technologies

By using the object oriented approach with the implementation of the **PipeSource**, **PipeProcessor** and **PipeSink** traits by the components subclasses, we allow the system to be agnostic of the technological details by providing a specific API extension for each and treat all the components in the same manner.

5.4.12 FR14 - Application Logging

To allow for the application logging we use the Akka inbuilt logging system coupled with the Apache Log4j [3] **BasicConfigurator**. The first one allows us to establish logging messages on the actor behavior and functionality by overriding functions on certain actor lifecycle points such as **preStart()** and **postStop()**, and the second one, allows us to automatically have several logging settings about the application usage of the technological resources implemented as extensions.

[3] https://logging.apache.org/log4j/2.0/

5.4.13 FR15 - Pipeline Fault Tolerance

To implement a system to deal with exceptions, we needed to support error handling in the framework. To do so, we once again leveraged the power of the Akka framework and for each actor, we implemented a supervisor strategy that specifies how that actor should behave in the case of an exception. Each manager is responsible for handling the error situations of it's children. To do it, each manager implements a **OneForOneStrategy**, which means that the decision of how to handle the exception only applies to the child actor that failed. We implemented handling for **TimeoutException**, **IllegalArgumentException** and for the general **Exception**. When a child actor raises a **TimeoutException**, the father retries the connection up to ten times in a timerange of one minute by restarting the child. If it is not capable of connecting, it kills the actor.

5.4.14 FR16 - Pipeline Monitoring

This requirement was not implemented in the framework due to time constraints.

5.5 Conclusions

The proof of concept implemented and presented in this chapter presents a good starting point for extension and improvement. It presents one of many possible implementations that could be done on the architecture presented on 4 and it allows us to validate the actor system that we proposed. With it we can now focus on implementing technology specifics and test it.

Chapter 6

Testing

In this chapter we will implement technological extensions of the framework, one for each component type, and create a simple pipeline in order to test and gather results on the actor architecture implemented in chapter 5.

6.1 Kafka as a Source Component

In order for our framework to support Kafka as a source extension, we need to read information from a topic that is in a certain machine or cluster. This machine can be identified by an IP and a port. By looking at the Kafka consumer API [1] we can understand the properties that are necessary to be set when reading from a Kafka queue. The current API requires that at least three properties are set when setting a consumer:

- bootstrap.servers: list of brokers to contact under the format <ip>:<port>;
- key.deserializer, value.deserializer: specifies how to turn bytes into objects. We could for example specify string deserializers, and our record's key and value will just be simple strings;
- group.id: group of consumers that the consumer belongs to.

Many more properties can be set using the same API. Since the current application only supports typeconfig specification for the parsing, and since we are in control of the properties we want to add to a certain specification, we can just add that property as a required field of a Kafka type source and use it on the configuration of the consumer. This way, every specific implementation of a type can have it's own properties. By implementing the underlaying specifics of Kafka under it's own class file, we can then implement the respective proxy that will allow it to connect to the

[1] https://kafka.apache.org/0100/javadoc/index.html?org/apache/kafka/clients/consumer/KafkaConsumer.html

rest of the system. The proxy only has to extend on the **PipeSource** trait and implement it's **read()** method.

6.2 Spark as a Processor Component

To use Spark in our framework we need to send spark jobs to the spark cluster. A spark job consists in the loading of data and a set of transformations done to it. To do this, spark uses spark-submit [2] that allows us to submit a compiled Java Archive (JAR) with the respective processing transformations to the data in it. It's tipically used through the bash interface, but supports an API to allow us to do it programatically. This is the only way for spark to be used. Because we never know the transformations that the user wants to apply until runtime and because we don't want to do compilation of jars in runtime to have a jar with the requested transformations, one of the possible solutions to this problem was to create an uber jar [3] that packs all the possible transformations in it and then receives as arguments an array of strings of the transformations to be applied to the data, together with the data location. This means that we still leverage our common trait and data abstraction, but we have to bundle all the transformations in a separate jar of our application to be used independently for spark-submit. To do so, in our application.conf we need to specify the appname of the spark application that will be run, together with the master broker of Spark or Yarn and a set of strings separated by a dot that represent a set of ordered transformations that we want to apply to the data and that our application supports. For this simple case, that means only two simple operations:

- toLowercase: maps all the characters of the data to it's lowercase pair;
- removeSpecialCharacters: removes [,.!?:;] from the data and replaces it by an empty space;

By using the same context provided by the application of the transformation on the same JAR, we can chain the multiple transformations that we want to apply, where the output of the previous operation can serve as the input of the next one. This is done by using the RDD abstraction of Spark.

6.3 Filesystem as a Sink Component

Due to the fact that Scala already natively supports the filesystem abstraction, there should not be the need to create our own abstraction to connect it. However, we will still need to create it as a sink, implementing the **PipeSink** trait in order for our system to support it in an agnostic manner. To do so, as with the components before, we need to create a proxy that implements the necessary methods, which in the case of the sink is just **write()**, and then create it's technological implementation. For the filesytem, all we need is the path of where the information we want to

[2] https://spark.apache.org/docs/latest/submitting-applications.html
[3] https://stackoverflow.com/questions/11947037/what-is-an-uber-jar

read and write is, since we are already using the filesystem as the persistence layer. To do so we can just reuse the **utils.fileManager**.

6.4 Test Pipeline

With the technological implementations of the three types of components of our framework, we are now able to create a pipeline as an example to test it. In order to setup a simple pipeline with one source, one processor and one sink, we configure the application.conf file in the following manner:

```
# Source Configuration
flow.sources.kafka1.type = "kafka"
flow.sources.kafka1.bootstrap_servers = "quickstart.cloudera:9092"
flow.sources.kafka1.topic = "validation-test"
flow.sources.kafka1.group_id = "test-consumer-group"

# Processor Configuration
flow.processors.spark1.type = "spark"
flow.processors.spark1.appname = "spark-app"
flow.processors.spark1.master = "spark://quickstart.cloudera:7337"
flow.processors.spark1.transformations = "toLowerCase.removeSpecialCharacters"

# Sink Configuration
flow.sinks.filesystem1.type = "filesystem"
flow.sinks.filesystem1.location = "~/tmp/results/"

# Pipeline Configuration
flow.pipelines.pipeline1.sources = "kafka1"
flow.pipelines.pipeline1.processors = "spark1"
flow.pipelines.pipeline1.sinks = "filesystem1"
```

Listing 6.1: Test Pipeline Configuration File

To do so, we need to have available a kafka topic and spark in standalone mode. To do this, we used cloudera manager quickstart [4], a Virtual machine that already comes bundles with a graphical tool for configuration and management of several big data technologies infrastructures. With it, we setup a kafka console producer that was responsible by producing data to a created topic and we setup spark in standalone mode in order to use the filesystem instead of a distributed filesystem like HDFS. To access it from our localhost, we just need to expose the virtual machine ports and then add the respective host to our hosts file. To run the kafka producer we create ten million messages with an individual record size of a hundred bytes and a throughput of hundred thousand messages per second based on the standard properties specified by kafka for a producer.

[4] https://www.cloudera.com/downloads/quickstart_vms/5-12.html

```
> kafka-producer-perf-test.sh  \
--topic validation-test \
--num-records 10000000 \
--record-size 100 \
--throughput 100000 \
--producer.config config/producer.properties
```

Listing 6.2: Kafka producer command

For the spark configuration, we will be using the default specification of cloudera manager and simply submit the job to the corresponding port by using the standalone mode. The filesystem sink also doesn't require any extra setup. To run the application, we pass the path of the configuration file to the compiled jar of the application.

6.5 Results

The framework was able to correctly run the configured pipeline. It correctly read the messages from the kafka topic and store them by serializing the data abstraction layer to the filesystem. The processor layer was also able to correctly read the data from disk and put it back into the data abstraction with the new respective stamps. This data was then successfully processed by the spark processor and because we were using spark in standalone mode, we were able to just use the fileManager to read the data and then write it to the specified location in the configuration file. This allowed us to confirm that our proof-of-concept implementation is able to correctly process simple pipelines in the way we expected.

6.6 Additional Testing

Bellow are two other pipeline.conf scenarios that were tested in order to validate the framework with the same technologies. One where two pipelines are run simultaneously and another one where one of the component layers has more than one component to be run concurrently. Both of the tests were also successful.

6.6.1 Pipeline Paralelism

```
# One Extra Kafka Source
flow.sources.kafka2.type = "kafka"
flow.sources.kafka2.bootstrap_servers = "quickstart.cloudera:9092"
flow.sources.kafka2.topic = "validation-test-2"
flow.sources.kafka2.group_id = "test-consumer-group"

# Another Pipeline to be Run Together With pipeline1
flow.pipelines.pipeline2.sources = "kafka2"
```

```
flow.pipelines.pipeline2.processors = "spark1"
flow.pipelines.pipeline2.sinks = "filesystem1"
```

Listing 6.3: Pipeline Paralelism Configuration File

6.6.2 Component Paralelism

```
# Two Sources to be Read Simultaneously
flow.pipelines.pipeline3.sources = "kafka1 kafka2"
flow.pipelines.pipeline3.processors = "spark1"
flow.pipelines.pipeline3.sinks = "filesystem1"
```

Listing 6.4: Component Paralelism Configuration File

6.7 Limitations

The testing applied to the framework was relative small in terms of scope, restraining the range of possible scenarios and their results. To increase it, these are some scenarios that would be beneficial:

- Use the framework on a real case scenario with more complex technological implementations and needs;
- Force different errors upon the pipeline and see how the system handles them based on the supervisor strategy imposed by akka.

Chapter 7

Conclusions and Future Work

This chapter concludes the work realized for the dissertation by presenting the main contributions and the proposal of future work to continue the development of the framework.

7.1 Overview and Main Contributions

The following points summarize the main contributions of the work achieved by this dissertation:

- The study and definition of a flexible and modular architecture that serves as a solution for the problem of handling data orchestration in big data pipelines;
- The implementation of a proof-of-concept framework taking in consideration the architecture definition, using Scala and Akka, that can now be used by the team at Farfetch to be expanded to it's own technological needs and used to implement processing pipelines.

Overall, considering the extent of the work done we think we achieved the main goal of this dissertation. We managed to achieve a solution to the problem currently faced by the big data team and gave them a proof-of-concept that can now be used to implement their own extensions and start being tested on already existing pipelines used in the cluster. Unfortunately, we didn't manage to implement the pipeline monitoring feature as intended due to time constraints, but we constructed a resilient system that can easily be extended for that by leveraging the already existing data lineage system.

7.2 Future Work

Both the architecture and the proof-of-concept implemented present a good foundation for further development but still present certain limitations. In order to improve the usability of the project in the big data team, there are some points that could be taken in account for the future:

- **Flow-Based Visual Module**: As stated since section 4.1, this project would improve substantially if it had it's own flow-oriented visual approach for control and monitoring of the

	Proposed Solution	Implemented	Tested
FR01 - Source Configuration	●	●	●
FR02 - Processor Configuration	●	●	●
FR03 - Sink Configuration	●	●	●
FR04 - Pipeline Configuration	●	●	●
FR05 - Read Data From Sources	●	●	●
FR06 - Apply Transformations to Read Data	●	●	●
FR07 - Write Processed Data	●	●	●
FR08 - Component Communication	●	●	●
FR09 - Component Concurrency	●	●	●
FR10 - Pipeline Parallelism	●	●	●
FR11 - Component Parallelism	●	●	●
FR12 - Record Data Lineage Between Components	●	●	●
FR13 - Support Multiple Technologies	●	●	●
FR14 - Application Logging	●	●	●
FR15 - Pipeline Fault Tolerance	●	●	●
FR16 - Pipeline Monitoring	●	—	—

Figure 7.1: Overview of functional requirements completeness

created applications and pipelines. Most of the decisions related to the configuration of the framework were done so it could be easier for this kind of integration in the future.

- **Continuous Deployment Integration**: Another aspect that would improve the usage flow of the framework is the support of continuous integration and deployment tools like Jenkins[1]. By implementing support in the configuration, or by implementing it's own module, the framework could be responsible for allowing the processes to be integrated and deployed to the current cluster automatically;

- **Schema Validation**: One of the most frequent validations that is done on data that arrives to be processed is on it's schema in order to guarantee that the data is in the correct format. Even though we didn't approach this topic, we think it's important to extend the architecture to take this in account when reading data.

[1] https://jenkins.io/

- **Extend Framework Testing**: Improve the tests done on the current proof-of-concept in order to improve the current capacities when it comes to error handling and complex pipeline scenarios;

7.3 Lessons Learned

The work developed in this dissertation allowed us to gather some important lessons relative to the way the stated problem was tackled and to the way the solution was devised. The following list encompasses challenges and pitfalls that were identified in it's development:

- Overwhelming technological space: big data has an extensive set of technologies, frameworks, architectures and paradigms that can easily overwhelm someone new to the area. When starting it's easy to fall into the trap of thinking one should use all of them in order to be compliant with the needs of a lambda architecture implementation. However, as more research is put into understanding the underlaying model of each one of the different technologies, it's possible to identify that most of them, even though used for different purposes, follow the same underlaying concepts. In the case of the solution developed in this dissertation this realization helped us define a common abstraction model that was able to represent different components in an agnostic manner and still leverage the features of those technologies;

- Focus on the general overview before specifics: when we started the design of the framework, we were too focused on the concerns related to the specific technologies we wanted to implement as our testing set and that made the initial architectural decisions be very dependent on them. Eventually we realized that the architecture needed to be extensible and the decisions were no longer viable so we had to discard the previous assumptions before realizing that we should have defined first how to treat the framework in a more general and abstract approach;

- Dataflow specification needs to be explicit: new paradigms arise, new necessities surge and requirements evolve. For this reasons most of the technologies that are used today will eventually be surpassed by other ones in the future and any solution that tackles dataflow orchestration should have this as a main concern in the definition of it's architecture. If dataflow keeps itself bound to it's implicit technological definition it creates problems in scalability and extensibility. By providing the abstraction needed to decouple the two, we allow dataflow orchestration to be agnostic and thats exactly what we tried to achieve in our solution;

- A framework is never finished: we tried to present and identify the main characteristics that should be taken in account in the definition of the architecture of the framework proposed for the problem, but as stated before, it would be impossible to create a "finished" one. A framework is in constant development and all that is possible is to lay the foundations for a

solid future and allow it to be easily extended and modular which we tried to achieve with the decisions taken in the architecture and with the proof of concept implemented.

References

[AH15] AmirGandomi and Murtaza Haider. Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management, Volume 35, Issue 2*, pages 137–144, 2015.

[Ama] Amazon. Amazon simple storage service (s3) details. Available in https://aws.amazon.com/s3/details/, accessed last time in June 2017.

[AW16] Shahriar Akter and Samuel Fosso Wamba. Big data analytics in e-commerce: a systematic review and agenda for future research. *Electronic Markets Volume 26 Issue 2*, pages 173–194, 2016.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.

[Die12] Francis X. Diebold. A personal perspective on the origin(s) and development of "big data": The phenomenon, the term, and the discipline. Technical report, University of Pennsylvania, November 2012.

[Dul] Tamara Dull. Data warehouse vs data lake. Available in http://www.kdnuggets.com/2015/09/data-lake-vs-data-warehouse-key-differences.html, accessed last time in June 2017.

[Edo14] Uyoyo Zino Edosio. Big data analytics and its application in e-commerce. In *E-Commerce Technologies*. University of Bradford, 2014.

[Ela] Elastic. Kibana user guide. Available in https://www.elastic.co/guide/en/kibana/current/index.html, accessed last time in June 2017.

[Fan15] H. Fang. Managing data lakes in the big data era: What's a data lake and why has it became popular in the data management ecosystem. *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*, pages 820–824, 2015.

[FCG06] Sanjay Ghemawat Wilson C. Hsieh Deborah A. Wallach Mike Burrows Tushar Chandra Andrew Fikes Fay Chang, Jeffrey Dean and Robert E. Gruber. Bigtable: A distributed storage system for structured data. 2006.

[Foua] The Apache Software Foundation. Apache airflow documentation. Available in https://airflow.apache.org/, accessed last time in June 2017.

[Foub] The Apache Software Foundation. Apache flink introduction. Available in https://flink.apache.org/introduction.html, accessed last time in June 2017.

[Fouc] The Apache Software Foundation. Apache flume user guide. Available in https://flume.apache.org/FlumeUserGuide.html, accessed last time in June 2017.

[Foud] The Apache Software Foundation. Apache hbase reference guide. Available in http://hbase.apache.org/book.html, accessed last time in June 2017.

[Foue] The Apache Software Foundation. Apache hive documentation. Available in https://cwiki.apache.org/confluence/display/Hive/Home, accessed last time in June 2017.

[Fouf] The Apache Software Foundation. Apache kafka introduction. Available at https://kafka.apache.org/intro, accessed last time in June 2017.

[Foug] The Apache Software Foundation. Apache logstash reference. Available in https://www.elastic.co/guide/en/logstash/current/index.html, accessed last time in June 2017.

[Fouh] The Apache Software Foundation. Apache oozie documentation. Available in http://oozie.apache.org/docs/4.3.1/index.html, accessed last time in June 2017.

[Foui] The Apache Software Foundation. Apache spark overview. Available in https://spark.apache.org/docs/latest/, accessed last time in June 2017.

[Fouj] The Apache Software Foundation. Apache storm documentation. Available in http://storm.apache.org/releases/2.0.0-SNAPSHOT/index.html, accessed last time in June 2017.

[Fouk] The Apache Software Foundation. Haddop mapreduce tutorial. Available in https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html#Overview, accessed last time in June 2017.

[Foul] The Apache Software Foundation. Hdfs architecture guide. Available in https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, accessed last time in June 2017.

[Gra] Grafana. Grafana features. Available in https://grafana.com/grafana/, accessed last time in June 2017.

[GSP17] Neha Narkhede Gwen Shapira and Todd Palino. *Kafka: The Definitive Guide*. O'Reilly, 2017.

[Hil11] Martin Hilbert. The world's technological capacity to store, communicate, and compute information. *Science 332. 60*, pages 60–65, 2011.

[Inf] InfoR. Star schema. Available in https://docs.infor.com/help_lawson_cloudsuite_10.0/index.jsp?topic=%2Fcom.lawson.help.reporting%2Fcom.lawson.help.bpwag-w_10.4.0%2FL55461185818015.html , accessed last time in June 2017.

[Ish15] J.Anuradhab Ishwarappa. A brief introduction on big data 5vs characteristics and hadoop technology. *Procedia Computer Science Volume 48*, pages 319–324, 2015.

REFERENCES

[Kai] William El Kaim. Big data architecture (re)invented. Available in https://www.slideshare.net/welkaim/big-data-architecture-part-1, accessed last time in June 2017.

[KR13] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. Wiley, Third edition, 2013.

[Kre] Jay Kreps. Putting apache kafka to use: A practical guide to building a streaming platform (part 1). Available in https://www.confluent.io/blog/stream-data-platform-1/, accessed last time in June 2017.

[KSC10] Sanjay Radia Konstantin Shvachko, Hairong Kuang and Robert Chansler. The hadoop distributed file system. 2010.

[MW15] Nathan Marz and James Warren. *Big Data - Principles and best practices of scalable realtime data systems*. Manning Publications, 2015.

[MZW15] Andy Konwinski Matei Zaharia, Holden Karau and Patrick Wendell. *Learning Spark - Lightning-Fast Big Data Analysis*. O'Reilly, 2015.

[Pro] Prometheus. Prometheus documentation. Available in https://prometheus.io/docs/introduction/overview/, accessed last time in June 2017.

[Rag] Siva Raghupathy. Big data architectural patterns and best practices on aws. Available in https://www.slideshare.net/AmazonWebServices/big-data-architectural-patterns-and-best-practices, accessed last time in June 2017.

[Red] RedisLabs. Redis documentation. Available in https://redis.io/, accessed last time in June 2017.

[Sah] Sunita Sahu. Dimensional modeling advantages. Available in https://www.slideshare.net/sunitasahu101/dimensional-modeling-53600268, accessed last time in June 2017.

[Sch] Guido Schmutz. Big data architectures. Available in https://www.slideshare.net/gschmutz/big-data-architecture-53231252, accessed last time in June 2017.

[SGL03] Howard Gobioff Sanjay Ghemawat and Shun-Tak Leung. The google file system. 2003.

[Spo] Spotify. Luigi documentation. Available in https://github.com/spotify/luigi, accessed last time in June 2017.

[Wal] Ben Walker. Everyday big data statistics. Available at http://www.vcloudnews.com/every-day-big-data\-statistics-2-5-quintillion-bytes-of-data-created-daily/, accessed last time in June 2017.

[Whi15] Tom White. *Hadoop: The Definitive Guide*. O'Reilly, Fourth edition, 2015.