POLITECNICO DI MILANO

Dipartimento di Elettronica, Informazione e Bioingegneria
Master of Science in Computer Science and Engineering

DESIGN AND IMPLEMENTATION OF A SOFTWARE PIPELINE FOR MACHINE LEARNING ON STREAMING DATA

Supervisor:
Prof. Francesco Musumeci
Co-Supervisor:
Prof. Massimo Tornatore

Candidate:
Matteo Formentin
Matricola 928164

Academic Year 2020-2021

Abstract

The application of machine learning (ML) methodologies in various fields of everyday life has become prominent in recent years. ML models are used in operational environments to perform many difficult tasks, such as, traffic classification, automatic failure detection, image recognition, that traditionally require specialized human expertise and effort to be accomplished. ML models are becoming more and more accurate, less prone to errors and able to highlight patterns that are hidden inside data and not even an expert eye can find. One of the most challenging objectives when deploying ML algorithms, besides developing the ML models, is how to make such ML models efficiently work in an operational environment, e.g., where the scale of the data to handle grows with the number of users and no service downtime can be tolerated. In such cases, the hardware/software infrastructure that supports the ML models should be able to scale and be resilient to software or hardware failures. In this work, we propose a distributed, scalable and fault-tolerant software pipeline that supports data ingestion and application of ML models to streaming data. The proposed pipeline can support thousands of data sources in a real environment and is built using state of the art and open sources software that are chained to form a software pipeline that achieves near real-time application of the machine learning model. As an application environment of the developed data ingestion pipeline, we consider the case of failure management in microwave networks, where the objective of ML models is to detect and to classify failure in microwave equipment by only looking at the working logs of the hardware that reports power measurement over the link. Finally, we present the experimental results on this use case, concentrating on the latency introduced by the pipeline components on a different number of active data streams.

Abstract (Italiano)

Le applicazioni delle tecniche di machine Learning (ML) nei diversi settori è diventata prominente negli ultimi anni. I modelli ML sono usati in ambienti operazionali per eseguire compiti difficili non assolvibili da algoritmi tradizionali e che spesso richiedono personale specializzato, come la classificazione di traffico criptato, il riconoscimento automatico dei guasti, o il riconoscimento delle immagini. I modelli ML sono sempre più accurati e sono capaci di evidenziare pattern nascosti nei dati che spesso neanche un occhio esperto riesce a trovare. Una delle sfide più difficili nella progettazione di algoritmi ML, oltre allo sviluppo del modello stesso, è come rendere il modello ML efficiente in un ambiente operazionale, dove la dimensione dei dati da gestire cresce all'aumentare degli utenti e dove non possono essere tollerate interruzioni del servizio. In questi casi, l'infrastruttura che supporta l'esecuzione del modello ML deve essere scalabile e resistente a guasti hardware e software. In questo lavoro, proponiamo una pipeline software distribuita, scalabile e resistente a guasti per supportare l'acquisizione e l'applicazione di modelli ML a dati streaming. L'architettura proposta è in grado di supportare migliaia di sorgenti dati e l'applicazione di modelli ML in near real-time in ambienti reali ed è costruita usando software open source concatenati in una pipeline software. Come caso di studio per la pipeline sviluppata, consideriamo il caso della gestione dei guasti nelle reti costituite da link a microonde, il cui obiettivo è di riconoscere e classificare errori negli apparati di rete usando un modello ML che si basa sui report di funzionamento e sulle misurazioni radio eseguite dai link. Infine, presentiamo i risultati sperimentali in questo caso d'uso, focalizzandoci sulla latenza tra la creazione e la classificazione dei report di rete introdotta dai componenti della pipeline considerando una quantità crescente di sorgenti dati attive.

Contents

List of Figures

Chapter 1

Introduction

The application of machine learning (ML) methodologies, thanks to the increasing number of available frameworks, often open-source, is becoming pervasive in many fields, from business analytics and finance to the gallery of our phones. Also, the huge availability of training data, driven by the high storage capacity of datacenters and modern data collection systems, makes it possible to develop ML models that are more accurate and less prone to errors. The machine learning diffusion corresponds to an increasing amount of users and systems that access the predictions provided by the ML model. This poses new challenges, as the developed ML model has to be placed in the so-called operational environment. Operational environments are typically not under the control of the developers, in contrast to the development ones, which are fully under control. The main difference is that it is difficult to predict the scale of the data that the system could be required to handle and that these values can change anytime. Also, hardware and software systems can become faulty at any moment, but no downtime can be tolerated in operational environments, as real users and systems rely on the services provided by the ML infrastructure. To address these two problems, the infrastructure that powers the ML model should be scaled and replicated accordingly [10].

Another challenge of operational environments is that we are subject to strict time requirements, often real-time, to avoid bottlenecks in the system supported by the Ml model. Most of the time, the application of ML models is only one element of a larger processing chain, called a software pipeline. Pipelines are used to handle streaming data flows,

where the system receives the data, also called events or messages, to be processed in real-time [11]. Events weights just a few kilobytes, and they can be seen as a start-less, endless flow of data, making them very difficult to handle. Streaming data are becoming the most common type of data to handle, since they could be associated with web interactions, like for example the clicks of a user on an e-commerce website, or working logs, like the one produced by IoT devices. The applications that are chained in the pipeline perform one transformation step on the inputs events and then forward the output to the next component of the chain. If the machine learning application step requires too much time, it could lead to blocking the full pipeline. While scaling the infrastructure partially solves these issues, the problem should also be addressed by running ML models on specialized software that enable distributed low latency prediction on large amounts of data. Monitoring the performance and the status of the software in the pipeline becomes fundamental to address the strict requirements imposed by the nature of the operational environment: administrators must be able to detect faulty applications and fast replace them, early recognize possible bottlenecks and solve them as soon as possible. Monitoring requires a specialized application and dedicated storage to support the collection of the metrics, as well as a dedicated interface to analyze the collected data.

Another important aspect of operational environments is how to handle security, a problem that is often neglected in development phases. Hacker attacks on operational systems are becoming more frequent, making security one of the most important concerns for operational environments where sensitive information is processed. We should be able to ensure the so-called CIA triad [15]: *Confidentiality*, meaning that all the communication between the processing steps must not be read or made available to non-authorized entities, *Integrity*, the data should be consisted and not altered by malicious users and *Availability*, the ability of the system to be always working. Ensuring this property in an operational environment is critical to avoid potential data leaks or outage of the system by an attacker, so the securing of the stack of applications should be performed from the beginning of the development process by applying security best practices like encryption and authentication between parties. The software that is chosen should be well

known to have been developed with security as the first concern, what is called *security by design*. Open-source applications have a greater advantage over closed source ones when concerning security: being free they are widely adopted and tested against vulnerabilities and bugs, and since their code is publicly available it is usually peer-reviewed by security experts.

To incorporate the aforementioned requirements in our framework, we propose a software pipeline architecture supporting the application of ML models in an operational environment in a scalable and fault-tolerant way, and per- forming near real-time processing of streaming data. The proposed design can support up to ten thousand data sources in a real, unpredictable environment and is built using state of the art and open sources software modules that are chained to form a software pipeline. All the components support distributed deployment and are compliant with security best practices: all the applications are designed to run in a cluster to perform work distribution and parallel processing, and they include encryption of exchanged data and authentication between parties. The proposed pipeline architecture is composed of four parts:

- Data Ingestion: Applications that support the ingestion of the data inside the system. It has four phases: buffering, preprocessing and storage. This is the entry point for the streaming data sources.

- Machine learning model application: stored data are batch-processed with the machine learning model and the results are stored back.

- Data Visualization: allows users to look at the input data and outcomes of data processing in a web-based interface.

- Health monitoring: monitors the status of the pipeline applications to detect bottlenecks or software failures and fast react to them.

To make a reference implementation and to have a real use case to test the developed pipeline, we consider the application of ML for failure management in microwave networks and use a pre-developed ML model that performs classification of failure causes in a real microwave

communication network. Nonetheless, the proposed design is general and can be adapted to any kind of use case that involves applications of ML on huge amounts of streaming data sources.

1.1 Related work

Software pipelines are becoming the standard to handle streaming data sources in many different fields. In the networking field, one notable example is [20], where the authors propose a method to collect network telemetry using the Apache Kafka publish-subscribe pattern, an approach that is also developed in this work, with the possible extension with machine learning consumers for further data analysis. Also in [21], Kafka is used as the core application to implement the data collection function of the Zero Touch Network and Service Management (ZSM), an emerging architecture for automating network management. In [18], a ML pipeline orchestrator is proposed to automatically manage and deploy the applications that compose the pipeline. In [19], the authors present a general-purpose machine learning pipeline based on microservices for log classification that is able to perform automatic model training. In [14], an Apache Spark-based pipeline is used to handle machine learning classification of streamed ECG (electrocardiogram) data to detect anomalies in patients.

1.2 Thesis outline

The thesis is organized as follow:

In chapter 2, the software tools used to build the pipeline are introduced.

In chapter 3, we describe the design of the pipeline in detail. First, we introduce the data ingestion part, then we explain how to handle the machine learning model applied to the data streams. Finally, we describe how to monitor the pipeline performance and how to visualize the machine learning application results.

In chapter 4, we introduce the failure management in microwave networks problem, and then we explain how the proposed pipeline architecture can be configured to suit this particular use case.

In chapter 5 we present the numerical results about the performance of the pipeline, using as a test case the failure management in microwave networks problem.

In chapter 6 we present the conclusions of our work and explore the possible future research directions.

The documentation of the implementation of the pipeline for failure management in microwave networks use-case is provided as an appendix to this thesis.

Chapter 2

Software used

This chapter describes the main applications that are used to build the pipeline. We highlight the main purpose and design characteristics of each software that is chosen to be part of the pipeline architecture. We also introduce Docker, the tool used for powering the infrastructure that runs the pipeline.

2.1 Docker

Managing and maintain a cluster made of many applications is a big challenge in deploying distributed systems. Some years ago the only option to run multiple instances in parallel on limited hardware was virtualization. Virtual Machines have the great advantage of enabling hardware sharing with a fine-grained assignment of available resources, but that comes at some cost. The main drawback of VMs is that they are difficult to maintain since it is necessary to take care of the operating system other than the software that they are intended to run. Running a complete OS requires a huge amount of resources, and in case we need another instance of an existing component to perform horizontal scaling, another VM should be installed and configured to handle the new requirements. All these limitations make a VM-based cluster unsuitable for a development environment, where specifications can change at any time and things are quite prone to break, but also for an operational environment, where fast scaling is needed to accommodate peaks of works or react to the fault of a component.

Docker comes as a solution for all these limitations and adds some

extra features that are not available in traditional virtual machines, such as image-based deployment, image registries and automatic stack deployment, while at the same time it guarantees the complete isolation between the different applications as provided by virtual machines. for all these reasons Docker is the chosen engine to run our pipeline. In the next sections, the main architecture and features of Docker are presented, using as reference the official documentation [24].

2.1.1 Containers and Virtual Machines

The solution adopted by Docker to overcome the limitation of virtual machines is to add another level of abstraction: while a traditional hypervisor abstracts the underlain physical hardware, Docker abstraction is placed at the operating system level. This is achieved by introducing *containers*, a way to package an application with its required dependencies such as additional binaries or libraries: for example, a Java application could be packaged in a container that contains also the JVM to run it. Each container is provided with an interface to a virtual Linux operating system, making the containerized software believing it is running on top of a reals OS: this will ensure that any existing application could be run inside a container without requiring any modification to the source code. Containers provide the same isolation allowed by VMs, since each one runs in a different virtual environment, completely disjoint from the others. The Docker engine is in charge of managing the OS calls of the containerized application and of forwarding them to the host operating system. Figure 2.1 highlights the main difference between Docker and classical virtualization [2]: the most evident advantage is that despite the number of running applications is the same, Docker requires just one operating system, which means only one system to manage and keep updated and functional and lots of disk space saved: since they do not include the OS, containers are lightweights than virtual machines. This makes it possible to package every single component of a cluster in a different container, something which will be too costly for a VMs environment, allowing the so-called microservices architecture [9]: this higher degree of isolation makes the system much easier to manage and ensure no conflicts arises between software installed on the same environment.

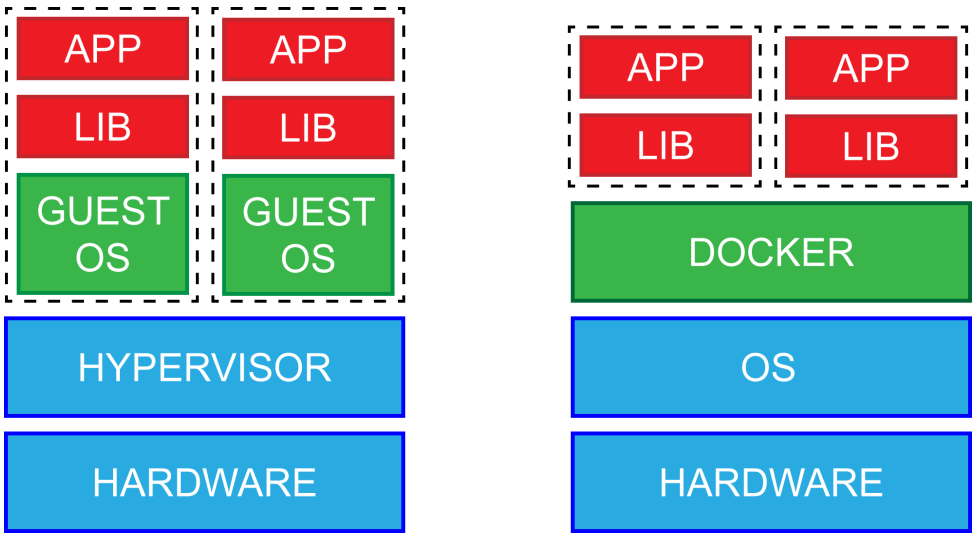

Figure 2.1: Architecture of Docker in comparison with classical virtualization.

2.1.2 Images

The containers mechanism has another big advantage over virtual machines: being the application and its dependencies completely abstracted by the OS, to define a container the only thing to do is to specify the application to run and its dependencies. Docker implements a simple solution to define how new containers should be created by using containers templates, called images. Images can be build by defining a dockerfile, which declares how to install the application and the dependencies that should be containerized with a bash style syntax. But the real power of images is the public registry: there the images for almost any available application can be found and easily instantiated as containers in just a few seconds. Thanks to the imaging system, scaling any application by adding more instances requires no effort: it is sufficient to create a new container starting from the required image.

2.1.3 Containers Lifecycle

To run a container, an image of it should be available. Images can be created by running the *docker build* command with a dockerfile as an argument or they can be downloaded from a registry if already available

using *docker pull*. Once the image is available, it can be instantiated as a container with the *docker run* command. Figure 2.2 shows the life cycle of containers. Multiple containers can be created from a single image, allowing fast replication and horizontal scaling of the application. Every time a container is created it starts in a "clean" state that is the one defined by the image. Soon after the boot, the container will hold a state that reflects the computations done by the application and eventual file system modifications, like files creation. Once a container is deleted, everything different from the clean state is lost, and new containers are created in the clean state. However, docker allows the persistence of the file system by defining volumes that are mounted inside the container in a particular directory. For example, a common operation is mounting a configuration directory in all the containers created as instances of a particular image: in this way, all of them can read and run the provided configuration.

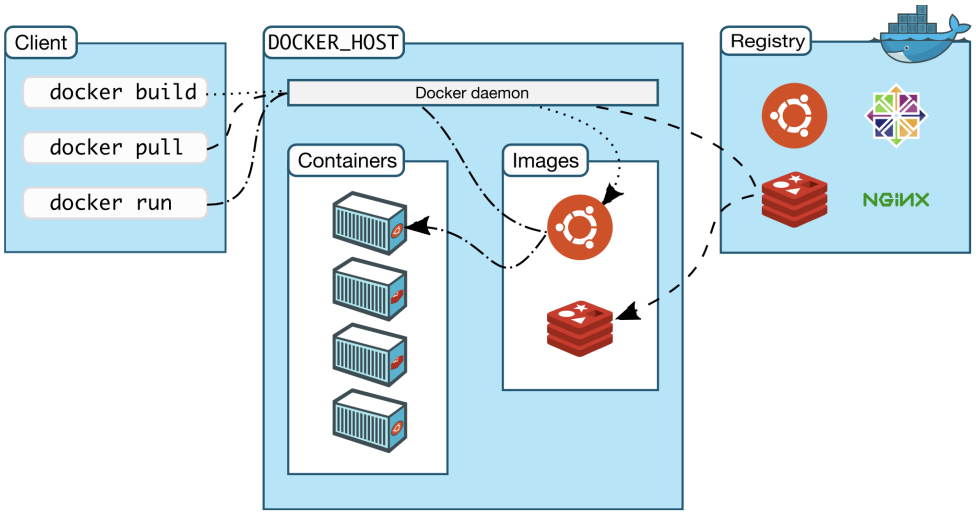

Figure 2.2: Docker containers life cycle. Image source: [24].

2.1.4 Networking

When a container is created, the default behaviour of Docker is to place them in the default *Docker network*, a subnet bridged with the host machine one that provides isolation with the rest of the network and

from other applications that runs on the machine. The user can create a custom Docker network to better segment containers. For security reasons, the Docker engine put a strict firewall on Docker network bridges to block all the external requests, so that only containers that run inside the same network can communicate with each other. To allows external access to a network, Docker exploits a mechanism similar to NAT port bindings between the container that should be accessed and the IP address of the host machine. For example, consider a container that exposes one service on port 80: port binding requires the host machine to bind one of its free ports to the container port 80. If the host chosen port is 8080, a client that is not on the docker network sends its requests to the host machine IP address on port 8080 then the docker engine forwards them to the right container on port 80.

2.1.5 Docker Compose

Docker makes running a full stack of applications as easy as downloading and running as containers the required images. To make this process even simple, *Docker Compose* has been introduced to automate the tedious task of running too many *docker run* command with the right settings. Compose allow defining the containers to run as well as their parameters, networking settings and volumes to mount in a single file that defines the entire cluster. Then it is sufficient to run the *docker compose* command specifying the file to run and docker will take care of running all the required operations to start the cluster. This enormously simplifies the deployment in operational environments of stacks of applications: it is sufficient to maintain a general-purpose Docker machine, copy on it the docker-compose file and then run the *docker compose* command. Docker Engine then automatically performs the following operations:

1. Downloads the images of the containers from the provided registry
2. Creates a network with specified settings: this keeps other application stacks running on the same host isolated
3. Creates volumes for containers persistence if needed
4. Instantiates as many containers as specified for each image, then it

connects them to the network, binds ports with host and mounts the volumes if required.

As many clusters as needed could be created on the same host, Docker Engine will take care of physical resources management.

2.2 Apache Kafka

Apache Kafka is an open-source distributed event streaming platform originally developed at Linkedin. The main purpose of Kafka is to provide a message broker that interconnects different applications but can also be used to provide distributed queues or buffers. Data streaming refers to a use case where the data to process comes to the system in real-time organized in chunks, also called messages or events. Streams are difficult to handle since the rate of new events can change rapidly in an unpredictable way. This makes the direct interconnection between different applications impossible to be performed classically like, for example, using a mechanism such as API. This is needed because if the load on one of the destinations is too high some messages could be discarded since the application could have not the resources to get and process it immediately. Also in a distributed environment where the work could be shared by many instances (or nodes) of the same software, it is necessary to have a load balance of the data to process between the nodes and also some kind of coordination that ensures that the same message is not processed two times by different instances. Apache Kafka has been developed to provide all these kinds of functionality. The main resource for this section is [22].

2.2.1 Kafka architecture

Kafka architecture is based on the publish-subscribe pattern: when a new message should be sent to another application, the source one, called *publisher*, publish the message on a specific topic [38]. A topic is a unique name that identifies an independent queue inside Kafka. The destination application, the *subscriber*, subscribes to the topic of its interest and is notified when new data are available in the queue. In this way, source and destination are completed decoupled and abstracted. This provides three main advantages: first, there is a central node that acts as a router and as an entry point for all the messages directed to anywhere in the system, making it easy for anyone who needs some specific stream, even coming from many different sources, to get it simply by subscribing to that topic. Second, the message broker acts as a buffer: the messages are always kept in the queue, and the subscriber can read them when it has the available resources to process

them. Third, Kafka can also act as a load balancer between multiple workers by exploiting its topic partition feature. Figure 2.3 shows the architecture of a Kafka broker.

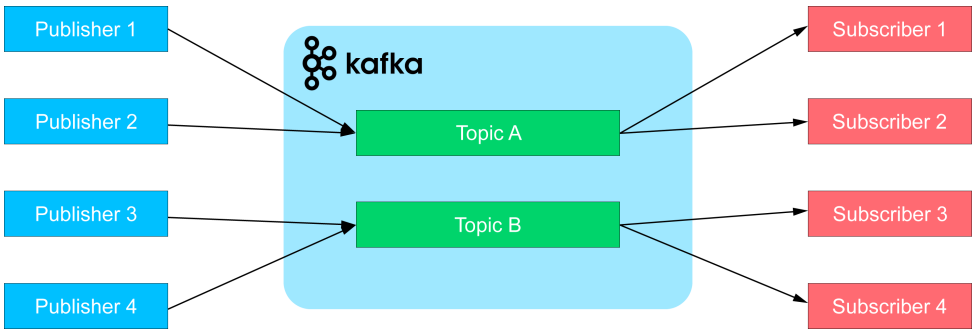

Figure 2.3: Architecture of Apache Kafka.

2.2.2 Topic structure

Kafka topics are organized as queues, but they differ from classical one since once consumed the messages are not discarded. Instead, they are kept in memory until a user-specified time has elapsed. This allows many different subscribers to consume the same message in any desired order. Kafka assigns to each message in the queue a unique progressive index, called *offset*, to identify it. The consumer must provide Kafka with the index of the message that it wants to read inside each request. Topics are fault-tolerant by design: messages are kept also on the disk, so if a crash appends, they can be recovered at the next start. Topics also support a multiple broker split mechanism to provide fault tolerance, called *partitioning*, that is discussed in the next section as part of the clustered deployment of Kafka.

2.2.3 Clustered deployment

Kafka allows the distribution in a clustered environment, where multiple instances of the broker allow workload split on the nodes. The load sharing is based on the split of the topics on multiple partitions, each one holding a subset of the messages that are published on the topics.

For each partition, one different broker is elected as a partition leader and keep the primary partition while the other nodes in the cluster keep a copy. Whenever a new publisher joins the topic, it is assigned to a primary partition of the topic on the partition leader broker as the destination for its messages. Partition mechanism allows both fault tolerance and load sharing: whenever a broker fails, a new one is elected as leader of the fault partitions and since it has already a copy of the messages this operation is very fast. Load sharing is guaranteed on the publisher side by assign to a different primary partition each producer, on the subscriber side by reading the topics from multiple brokers to get the full message queue. The number of partitions of a topic is a configurable parameter, not tied to the number of brokers in the cluster: also in a single instance deployment, there could be multiple partitions for a topic to exploit consumer groups, discussed in the next session.

2.2.4 Parallel processing of the topic

Kafka topic partitions allow the processing of one distributed queue by multiple subscribers that can consume different subsets of messages in parallel. This multiplies the throughput of the system but requires additional coordination between the subscriber and the brokers since each message should be processed exactly one time by one consumer only. This is done in Kafka by organizing the subscribers in *consumer groups*. Each consumer group member read a partition of the topic but the whole group consume the full amount of messages. Having many consumer groups allows multiple groups to read copies of the queue, but at the same time ensure that each member of the group read a subset of them, enabling parallel processing of the messages.

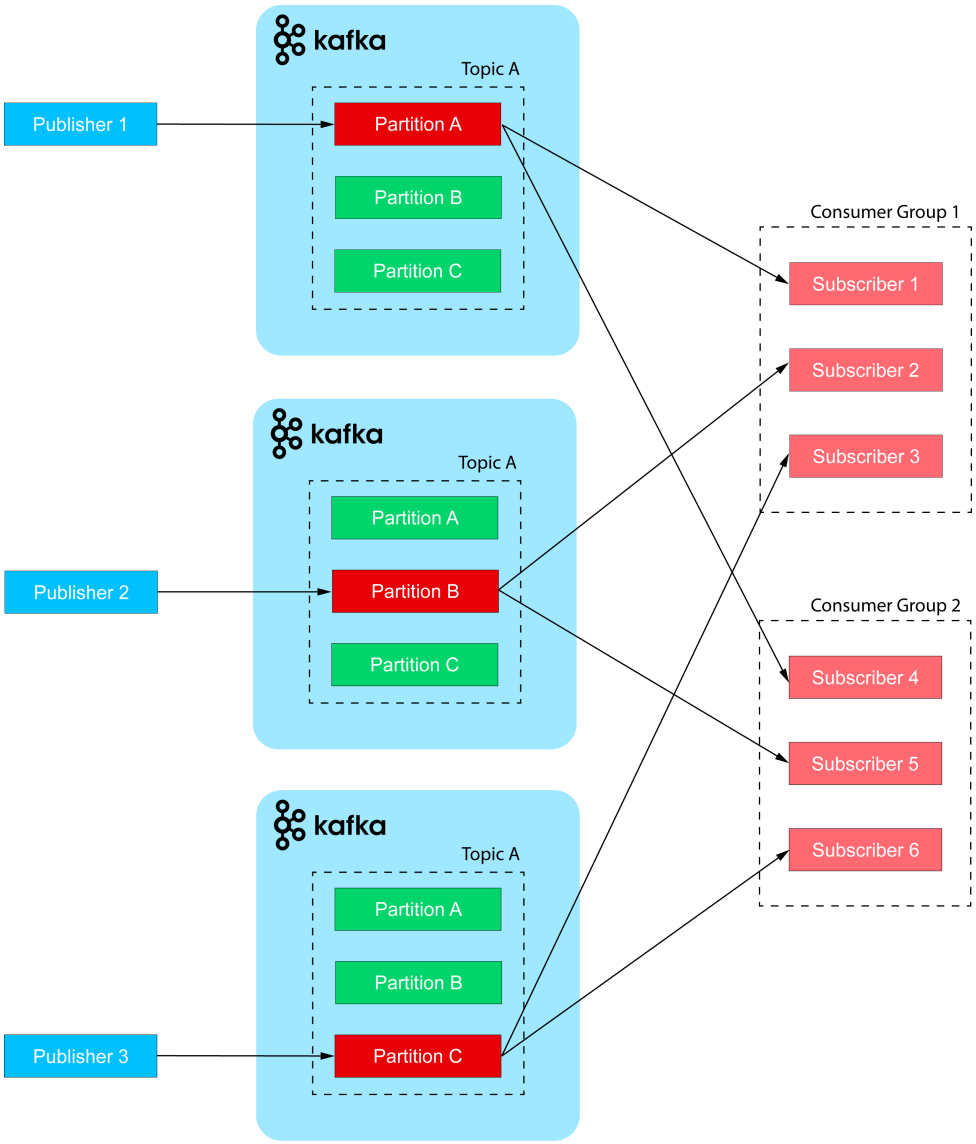

Figure 2.4: Clustered deployment of Apache Kafka. The red highlighted partitions are the leader one.

2.3 Logstash

Logstash is an ETL (Extract, Transform and Load) tool, designed to be part of a software pipeline, as it performs an intermediate transformation step in the data processing of streamed data. The main purpose of Logstash is to extract the data from a source, apply any kind of transformation and finally load them into the designed persistent data storage. It has a deep integration with Elasticsearch for storage and Kibana for data visualization. All three software are maintained by Elastic as the ELK stack. Logstash is easily configurable by defining a pipeline configuration file where the three ETL phases are defined. Main resource for this section is [31].

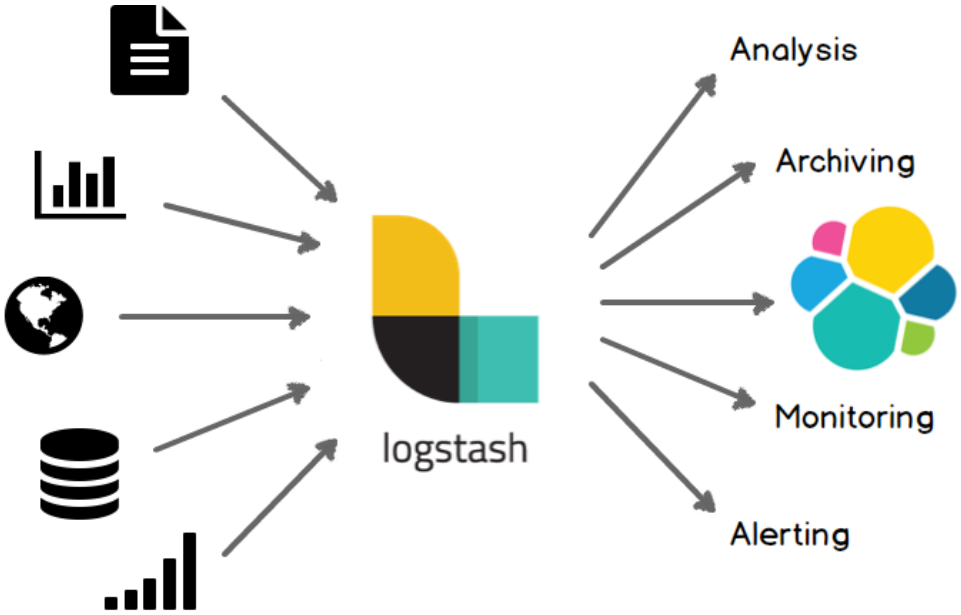

Figure 2.5: Logstash architecture. Image source: [31].

2.3.1 Data extraction

The first phase of the ETL pipeline is the definition of the inputs for the extraction of the data. Logstash is designed to work with streaming data sources, where data flows in real-time inside the application. It supports out of the box many common data sources, like Apache Kafka, SQL database, HTTP request or custom protocols over TCP/UDP. From each of these sources, it expects to have a flow of streaming events that will be passed to the transformation phase.

2.3.2 Data transformation

After the definition of the inputs, the streaming events start to flow inside Logstash. Incoming data can be divided into two categories [3]:

- *Structured data*: data comes organised in fields that can be interpreted by a machine. This is the case when sources are SQL databases, that are organized in tables, or JSON and XML.
- Unstructured data: in this case the data have no structure, and although they can be easily understood by a human, they are meaningless strings for a machine that does not know how to parse them. This is the case for logs or HTTP requests.

To use the data for tasks like analysis, aggregation or machine learning application, they must be structured. To handle unstructured sources, Logstash includes a parsing engine, called Grok, that is used to derive a structure from unstructured data. Grok includes out of the box patterns to match most of the common formats, but it is also configurable by hand by defining the regular expressions that match the fields to derive [29]. For both the structured and unstructured cases, the incoming data are then converted to JSON before going to the next phases. In case no parsing has been applied to an unstructured source, a one field JSON with the content of the event is emitted. Once structured, data can be further manipulated, by applying one of the many plugins available for the transformation phase, called *filters*. Examples spans from simple operations like adding fields like timestamp or deleting sensitive data, to more complex transformation likes data anonymize, where unique id replace user identifier or IP geo-localization where IP addresses are replaced with geographical location.

2.3.3 Data load

After the transformation of the data, Logstash sends them to the designed data storage, called *stash*, as JSON documents. Despite Elasticsearch is the most used stash, as it is designed to work smoothly with Logstash, data can be sent to a wide range of destinations, like no SQL databases, S3 buckets or even pushed back to another Kafka queue.

2.3.4 Parallel processing

The specification of the ETL pipeline is completely stateless: each event is a threat as independent from the previous and following ones, and no information despite the one contained in the event itself is required to apply the transformation phase. This makes it possible to process the flow of data in parallel by multiple Logstash instances without requiring any kind of coordination mechanism between the replica. For this reason, Logstash does not include any mechanism to form and maintain a cluster. To exploit parallelism, the data source must ensure that each replica gets different subsets of the event to avoid duplicate processing using a mechanism like Kafka partitions, discussed in section 2.2.4.

2.4 Elasticsearch

Elasticsearch is a distributed search engine, designed to store JSON documents and optimised to perform full-text search and complex aggregation on the stored data. Its main use cases include logs analytic, web search and infrastructure monitor. Main resource for this section is [26].

2.4.1 Elasticsearch architecture

Elasticsearch architecture is similar to the one of a No-SQL database, despite it does not provide all functions of a database, like transactions. Entries in Elasticsearch are schema-less JSON documents, which are loaded already parsed and organized in fields from ingestion applications like Logstash, discussed in the previous sections. JSON Documents have a great expressive advantage over the classical tabular field organization: they allow storing any kind of data structure, like arrays or nested objects, without needing to normalize them to a one-column one-value structure. Since Elasticsearch main purpose is to provide search over documents, fields of the documents are text-indexes to provide near real-time searching capabilities. Documents with something in common are organized on indexes, like SQL rows are organised in tables. Indexes are more flexible than tables, as Elasticsearch can handle thousands of them without losing performance: it is common on logging applications to have a separate index for each day, as search could be performed between multiple indexes. Data ingestion and querying from external applications are made possible by an HTTP REST API.

2.4.2 Difference from a database

Despite its No-SQL like structure, Elasticsearch is not intended to be a database since it lacks one of the main future of a database, transactions. A transaction is an operation on the data stored on a database, like an insert or an update, that must have some precise properties, summarised in the acronym *ACID* (Atomicity, Consistency, Integrity, Durability) [1]. These properties ensure that concurrent operations on documents do not place any incoherence in the database. The main motivation for this lacks is that Elasticsearch is not intended to be a

primary database with records that are updated frequently, so having a full-featured transaction mechanism on a cluster that is designed to scale horizontally on hundreds of nodes will make operations slow since distributed transactions require a huge amount of inter-nodes communication. Instead, Elasticsearch is optimised as a near real-time search engine, so the main focus is on the speed of reads, an operation that does not place any concern on consistency when concurrently executed by many clients. Another missing feature is the relations capability, but even if this feature is missed by most of the No-SQL databases, joins can be implemented by the user in most of them. On Elasticsearch there is no way to have something similar, since joins are the most expensive operations on a database, and this would impact too much on search speed. Since Elasticsearch is often used in conjunction with a classical database to have fast search and aggregation capabilities over the stored data [28], to emulate a real join the relations between tables are denormalized: this will leads to duplicates data but allows the maximum speed. Elasticsearch can be also used as primary storage in the case of mostly read-only workloads, where retrieval and aggregation speed is critical, like in the case of log analytics or monitoring of another application.

2.4.3 Clustered deployment

Despite Elasticsearch could be used in a standalone mode, it is mostly used in a clustered deployment: it is common for a distributed setup to hold petabytes of data in hundreds of nodes. The replication mechanism is based on the so-called shards, which are self-contained subsets of documents that belong to the same logical index. A shard can be:

- Primary shard: this hold when the shard is the main copy of the documents. When a search is performed, only primary shards are targeted. For each shard, there is only one that is primary.
- Secondary shard: these are backup copies of primary shards, that are kept on different nodes of the cluster. Each shard is independently indexed from the others parts of the same index, making the index only an abstracted logical grouping of shards. Elasticsearch cluster coordinators, called masters, are in charge of balancing the shards on the nodes of the cluster, to provide the maximum

availability and resiliency in case of disruption of the primary ones as well as managing the creation of indexes and the join of new nodes. If one master fails, another master is elected between the nodes and the secondary shard are promoted to primary and balanced on the cluster. Elasticsearch supports another replication system, called cross-cluster replication. In this case, a secondary dedicated site replicates the primary one: in case of unavailability of the main datacenter, the backup one takes its place.

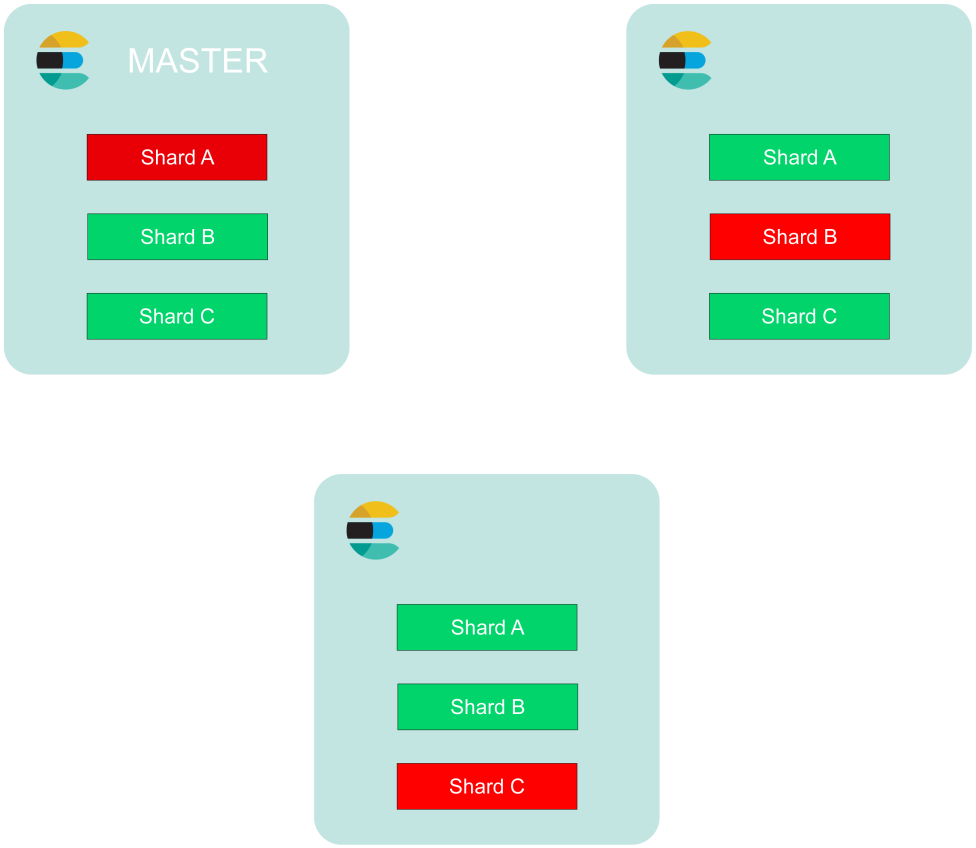

Figure 2.6: Elasticsearch cluster.

2.4.4 Search and aggregation

When new documents are added to Elasticsearch, they are automatically indexed. Indexes are a data structure that allows fast retrievals of specific documents without requiring to perform a full scan of the entire store to find the desired data. Indexes on Elasticsearch are optimised to perform full-text search on the fields of the documents, in contrast with databases where the indexes are optimised to retrieves the numerical key to having fast joins between tables. Full-text is the most common type of search performed by humans when dealing with data: the user provides a list of keywords and the engine provides the documents that match them in one or more indexed fields. The data structure used by Elasticsearch to index the stored documents is called *inverted index* [7]: each word that appears in a particular index is associated with the unique identifiers of the documents where that word appear. When the user provides a keyword, all that is done is to find the correct entry in the list of words and then retrieve the associated documents. Elasticsearch is also optimised to perform aggregation, a crucial operation in data analysis, on the matched documents: this means applying functions like sum, count, or average to get one value from many documents.

2.5 Kibana

Kibana is a data visualization framework part of the ELK stack. It features a web-based interface to visualizes dashboards that can be populated with widgets like charts, tables or even maps to show location-based data. It requires Elasticsearch as source and engine to retrieve and aggregate the data to be shown. Kibana is also used as a graphical interface to monitor and manage all the ELK stack components. The main resource for this section is [30].

2.5.1 Widget and dashboards

Kibana makes it easy to perform complex queries and aggregation from a graphical web interface against Elasticsearch documents. *Lens* function allows the creation of any kind of data visualization by selecting the index to visualize, drag and drop the field on the graph and selecting the aggregation function to use. Once created, the widget must be added to a dashboard to be used. Dashboards are collections of widgets that can be customised to deal with particular use cases or different users needs. Kibana can suit the needs of both expert users, with a background in data science, and non-expert users, that may need only some basic information: multiple dashboards can be created to differentiate the users, both by expertise and role inside the organization. For example, the IT team can access data relative to infrastructure performance, while analysts can access streams information.

2.5.2 Discover

Besides the dashboard view, Kibana allows expert users to perform a free analysis of the data through the *Discover* function: data analysts can select the index to visualise and perform aggregation through the time with functions like sum, average, variance, correlation and many more. This can be particularly useful to perform preliminary analysis for machine learning model development, directly on the full data and in real-time. Also, queries and aggregations designed in Discover can be easily converted into a widget that can be made available to non-expert users.

2.5.3 ELK stack monitor and management

Kibana serve also a centralised web interface to monitor and configure the full elastic stack. Two main sections are available:

- *Stack Management*: Allows management of all the main elements inside the ELK stack, like users roles and privileges, Elastcisearch indexes, Logstash pipelines, security and backup settings, all in a single web UI.

- *Stack Monitoring*: Allows the monitoring of the full ELK stack from a single web interface. The data includes common metrics like memory and disk usage, CPU load and custom metrics specific to the monitored applications. Also, a detailed breakdown of Logstash pipelines is available, to detect problems in the application of the filters to the data. Metrics collection is done by Metricbeat, introduced in the next section, and the acquired data are stored in a dedicated Elasticsearch index.

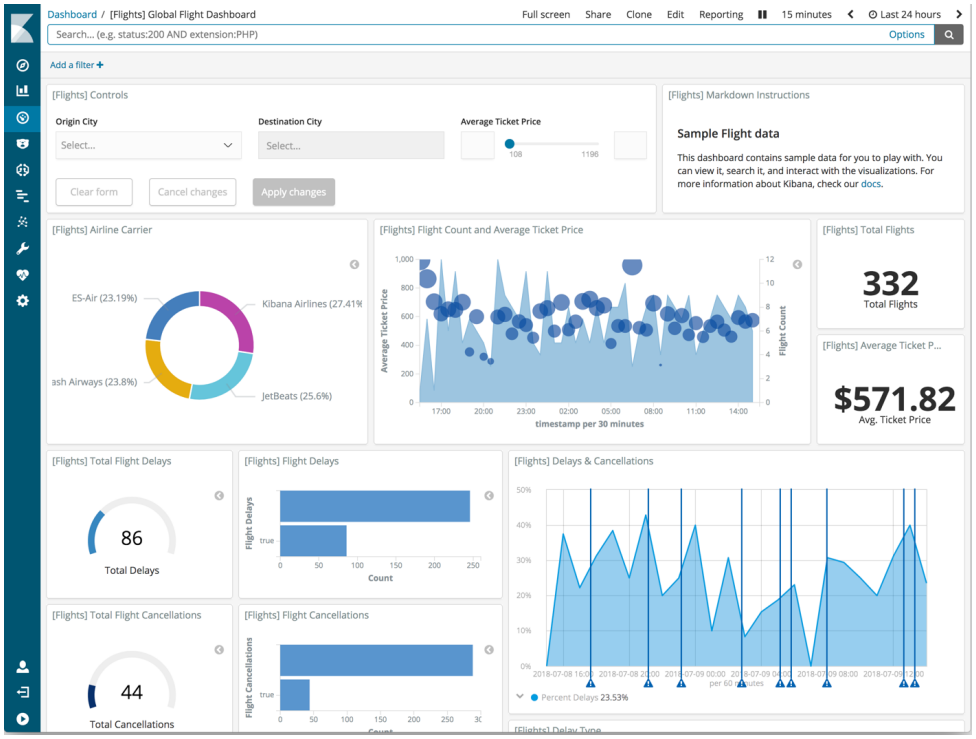

Figure 2.7: Sample Kibana dashboard.

2.6 Beats

Beats is a family of lightweight data shippers, developed by Elastic as an add-on for the ELK stack. A data shipper is an agent that can be installed on a server that runs another application to monitor the operating system and the programs that run on top of it. Data collected can be sent directly to Elasticsearch or published to a message broker like Apache Kafka and then processed with an ETL tool like Logstash. For our work two applications, Filebeat and Metricbeat, are used. Main resource for this section is [25].

2.6.1 Filebeat and Metricbeat

Filebeat and Metricbeat are agents that collect data produced by other applications on the same server where they are installed. They collect two types of data: Filebeats collects logs, which are records of the operations performed by an application like for example access to a system, while Metricbeat collects metrics of the applications, which are usage statistics like used RAM or CPU, available Disk space and so on.

Both can be easily configured by specifying the destination of the collected data and the format of the data to collect. Filebeat and Metricbeat have dozens of modules that allows collecting and parsing many common data sources: web server application like Apache or Nginx, databases like MySQL, cloud engines like AWS or Google Cloud. Even the ELK stack itself could be monitored by a dedicated ELK monitoring stack: a common deployment of these two applications is using them in combination with Elasticsearch for the storage and Kibana for the visualization. If the servers to monitor are many or with few resources, a broker like Kafka is employed for queuing and the parsing and injection are then offloaded to Logstash.

2.7 Apache Spark

Apache Spark is an in-memory distributed data processing engine used to perform general-purpose computation on large amounts of data, commonly referred to as *Big Data*. It provides programming API to access its functionality from Java, Scala and Python. Computation could be performed on unstructured data using low-level API or on structured data using an SQL like API, called SparkSQL. Main sources for this section are [39] and [16]

2.7.1 Challenges in big data processing

Developing applications that handle big of data is challenging. It is difficult to build them from scratch and it is better to use a dedicated framework like Spark, as some issues will fast arise:

- *Heterogeneous data sources*: data can come from a plethora of different sources, each one that has a different interface to access them, examples are SQL or No-SQL databases, unstructured sources like object storage systems, streaming data flows and many more. Sparks provide a unified interface to handle all kind of data sources.

- *Size of the data*: the term big data refers to the fact that the data to process does not fit one machine memory, so it must be distributed on multiple nodes. Spark provide a basic data structure called *RDD (Resilient Distributed Dataset)* that is build to distribute large amounts of data on multiple Spark instances in a fault-tolerant way. The great advantage of this data structure is that data are kept in memory: this makes Spark operation very fast if compared with alternative frameworks that keep data on disk.

- *Parallel processing of the data*: Also data processing and not only data storage should be performed in parallel on multiple machines to exploit all the available power of the cluster. Spark has an API that requires the programmer to define the operations to perform on the dataset like if only one machine is processing the data. Then the optimization engine automatically split the work into

smaller tasks and determine which operations can be performed in parallel and handle task distribution on the nodes of the cluster.

2.7.2 Spark cluster

Spark computations take place on multiple nodes. Inside the cluster, three main roles can be distinguished:

- *Driver*: This machine runs the program that is developed by the user. The programmer can access Spark functions through an object called *Spark Session*, which provides an entry point for all the available APIs and the way to access the cluster. The driver is in charge to plan how to execute the parallel computation by splitting jobs into atomic tasks that can be executed by the executors on the workers. It contacts the master to request resources allocations on the workers to run the computation. The driver could be the programmer PC on a development stage, but in operational environments is usually a dedicated machine.

- *Master*: The master is responsible for coordinating the cluster, managing the workers join and handle their eventual fault. The driver asks the master to allocate resources to the workers to perform computations.

- *Workers*: Workers are responsibly to run the tasks requested by the driver. The driver request resources to the master that asks workers to available executor on them. An executor is one process that can execute spark tasks, one worker has one executor for each core. Once executors are allocated, the driver directly contacts them to submit the tasks. Once the computation is done, results are sent back to the driver which can terminate the session or send other tasks.

Figure 2.8 shows the components of the cluster and the two phases of the allocation of tasks to executors.

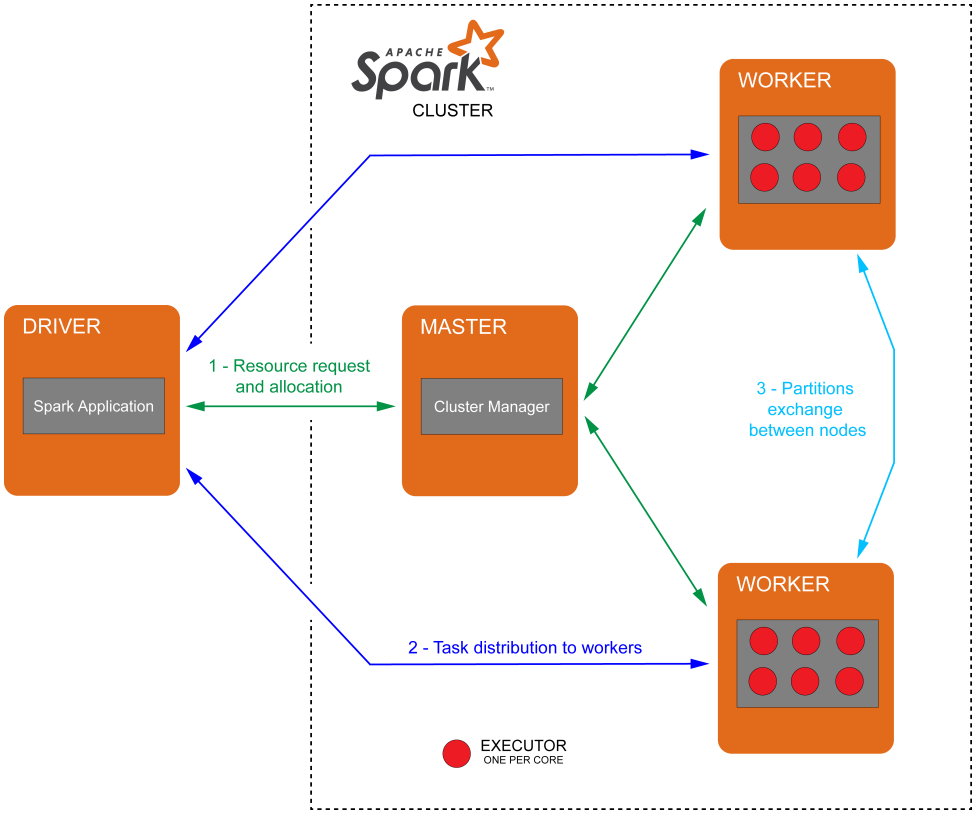

Figure 2.8: Spark cluster architecture.

2.7.3 Handling distributed data

RDD are the base data structure that represents an immutable, partitioned collection of elements organised in rows that can be operated in parallel on a Spark cluster. RDD partitions are distributed across the workers of the cluster and each partition has backup copies on other nodes to guarantee fault tolerance. Spark engine ensures that workers are allocated tasks that should access data only on partitions that are closer to them, minimizing the network bandwidth required for moving data between the nodes. RDD are unstructured data containers that could contain everything, from files to tables or JSON documents. Spark low-level operations, discussed in the next section, could lead to the need of accessing partitions that are kept on other nodes. In this

case, the worker that needs that data directly ask the worker who holds the partition for a copy of the data.

2.7.4 Broadcast variable

In a clustered environment there is no concept of global variables. A variable is called global when it has scope inside the whole program: since Spark applications are automatically decomposed in tasks executed on multiple nodes, variables are scoped only inside the task that is running on a specific node. RDD provides the data sharing between the nodes, but they are suited for a huge amount of data and not for common programming variables, like objects. To handle this lack, Sparks introduces *broadcast variables*: these variables are automatically distributed and always kept synced between all nodes inside the cluster. This feature is usually used to distribute parameters or machine learning models to perform parallel prediction on big data.

2.7.5 Low-level unstructured Spark API

Low-level Spark API is based on simple operations that take has parameter callbacks defined by the programmer, called user-defined functions, that operates directly on the rows of the RDD. These three basic operations could be chained together, since each function output another intermediate RDD. These operations are:

- *Map*: For each entry in the RDD, the callback is applied and one or more results are emitted, a common pattern is to output some keys with associated values. The user-defined function takes one parameter, the entry of the RDD, and emit one or a list of results. Since the function is defined on a single entry of the RDD, maps operations can be executed in parallel. For example, consider the simple word count task in figure 2.9: if the number of occurrences of a word in a file should be counted, the user-defined function mapped to each entry could split the words of a single file, count them and emit a list of words with the associated count.

- *Reduce*: Multiple RDD entries with the same key are reduced into one value. The user-defined function gets two RDD entries and outputs one value and is applied recursively until one value

is left. In the word count example, the intermediate word counts are grouped by word and each group count is summed and finally, one count per word is emitted.

- *Filter*: The entries of the RDD are filtered based on a condition. In the word count, we can, for example, decide to consider only files with a length greater or equal to a parameter.

By exploiting these basic operations, which together form the *MapReduce* [12] pattern, any kind of aggregation or processing on any kind of data could be defined.

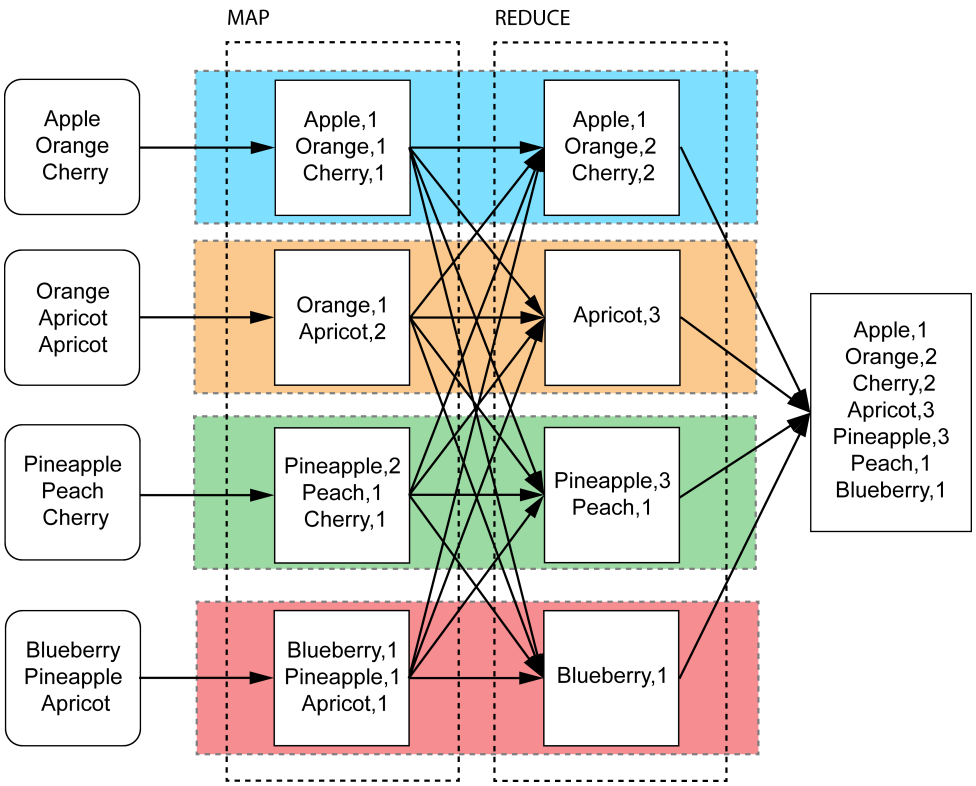

Figure 2.9: Word count example. The different color represents different nodes that works in parallel.

2.7.6 Structured Spark API

The problem with unstructured API is that user-defined functions are like a black box for Spark optimization engine: only the programmer knows what that function does to the data, making it impossible to run any optimization on the task plan. Furthermore, since no structures are associated with the data, neither the type of them is known: Spark must threat them as raw bytes sequences, making any compression technique application impossible. For this reason, unstructured APIs are rarely used. To overcome these problems SparkSQL structured API is introduced. SparkSQL provides an interface compatible with SQL syntax to perform queries and common operations on the underline data with predefined elementary SQL-like functions and it is built upon low-level unstructured one as an abstraction level. Since RDD can only handle unstructured data, a new data structure called *DataFrame* is built upon RDD to handle structured data using tables with fixed schema and type, like in a classical SQL database. Adding a set of predefined elementary functions makes the Spark optimization engine knowing the operations that are performed on the data, allowing an optimised operation plan that makes SparkSQL performs much better than low-level user-defined functions [40]. High-level API has the same expressiveness of user-defined function: map could be realised with the group by SQL function, filtering with the where clause and for reduce aggregations a huge number of functions like sum, average or count are already available. Despite using provided aggregation function is always preferable for performance reasons, sometimes they do not cover all the use cases. In this case, the programmer can still code custom user-defined functions: an example of such a case is performing machine learning training or prediction on big data: since Spark supports Python, libraries like Scikit-Learn or Tensorflow could be wrapped inside user-defined functions and used to perform such tasks by exploiting the distributed computation provided by Spark.

2.7.7 Distributed execution of jobs

Once the application is ready to be deployed, the driver machine optimization engine converts the application into a set of jobs that can be executed in parallel, for example, operations on different RDD or

DataFrame could be run in parallel. Each job is represented as a *DAG (direct acyclic graph)*, a sequence of elementary actions, called *stages*, that needs to be executed serially. In most cases what define the boundary of a stage is the data transfer between partitions to perform that stage. For example, a *count* executed after a *group by* is split into more stages since the group by operations require moving the entries that belong to each group to the node that handles that group. Only after groups are made Spark can perform the aggregation in the next stages. Each stage is then divided into *tasks*, the basic units of work that could be run in parallel on multiple executors. Continuing the previous example, each group could be handled in parallel by different executors on different workers. After building the DAG, the driver contacts the master to asks for available resources in the cluster, then the master allocates executors on the workers. Finally, the driver submits the tasks to the executors. Once the computation is done, the executors send the results back to the driver which assembles them into the outcome of the stage, then another stage is initiated.

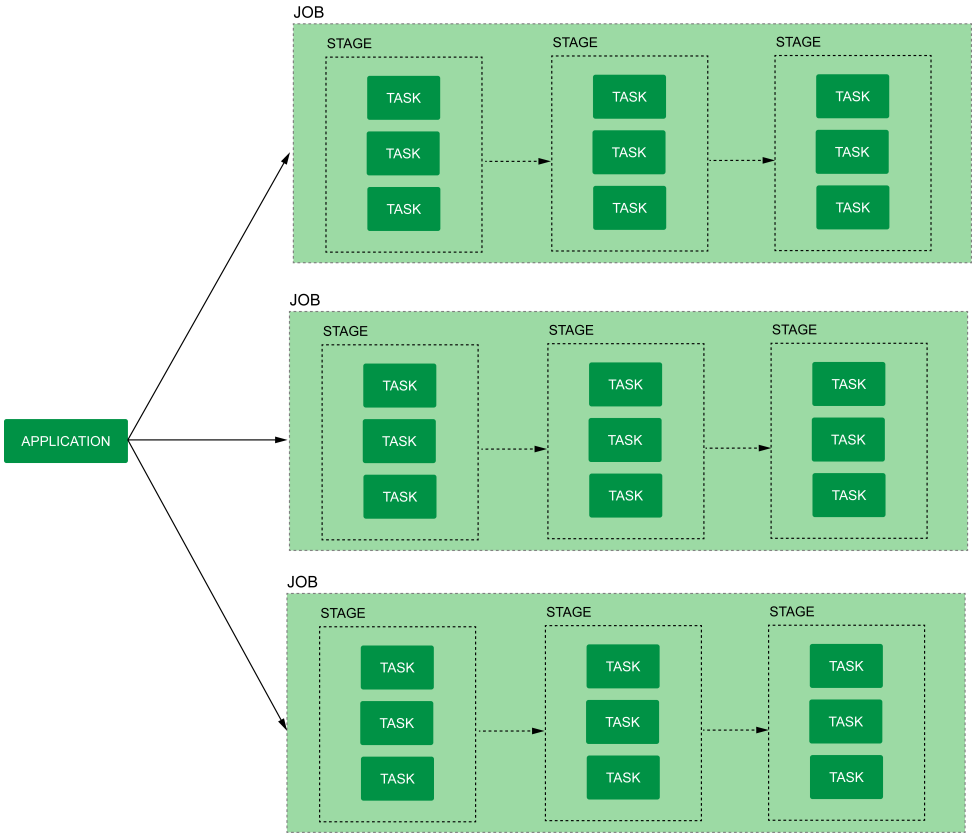

Figure 2.10: Spark direct acyclic graph (DAG).

2.8 Summary

In this section, we described the applications that are used in our work. In the next chapter, we describe how the introduced applications are used in the proposed machine learning pipeline architecture.

Chapter 3

Design of a software pipeline for machine learning on data streams

In this chapter, we describe how we design a general-purpose data ingestion pipeline supporting machine learning model deployment, that can be applied to a variety of use cases. First, an overview of software pipelines and streaming data is provided, then the proposed pipeline architecture is described.

3.1 Software pipeline and streaming data

A *software* pipeline is a chain of applications that process a flow of data sequentially, i.e., the output of one application is the input of the following one. This structure is particularly suited to process streams of data. Data streaming refers to limited amounts of data, also called *events*, usually with the size of a few kilobytes, that is processed by the various applications of the pipeline as soon as the source of data collects it [11]. Streamed data are particularly difficult to handle, since they do not have a start or an end, and their generation can be unpredictable. If there are multiple sources of data, it can be impossible to know in advance how many new data will reach the pipeline input in a given instant. Some examples of streaming data are IoT data, the interaction of users with a website, bank transactions or working logs of devices in a communication network, which is the use case considered in our work.

The most common operations that are performed on streaming data are processing, storage and visualization. However, due to the increasing interest in online data processing enabled by efficient machine learning algorithms and processing platforms, such as in the case of IoT device data analysis, online payment fraud detection, online user tracking, often the ML data processing step is included in the pipelines, adding further complexity to the infrastructure. In this chapter, we propose a generic software pipeline that can be used to handle data streams, which includes the classic pipeline operations as well as machine learning applications on streams of data. We divide the proposed pipeline into four groups, that are analysed separately and are listed below:

- Data Ingestion: Applications that support the ingestion of the data into the system. It has three sequential steps: buffering, preprocessing and storage.
- Machine learning application: application of a machine learning pre-trained ML model to the stream of data.
- Data Visualization: allows users to look at the data and to know machine learning outcomes in a web-based interface.
- Health monitoring: monitors the status of the pipeline applications to detect bottlenecks or software failures and reacts to them.

In particular, the data ingestion and machine learning application phases of the proposed pipeline are a particular case of a service function chain [18] that can be defined as a vector T(a, b, c, d) where a=queuing, b=data preprocessing, c=data storage, d=machine learning model application. These steps will be detailed later in this chapter.

Figure 3.1 shows an overview of the proposed pipeline, with the interaction between the components.

3.1.1 Main requirements for software pipelines

Running a software pipeline in an operational environment, where other users rely on the services provided, poses additional problems compared to a development environment. In particular, we focus on the following main requirements, which are the most important for any distributed system [8]:

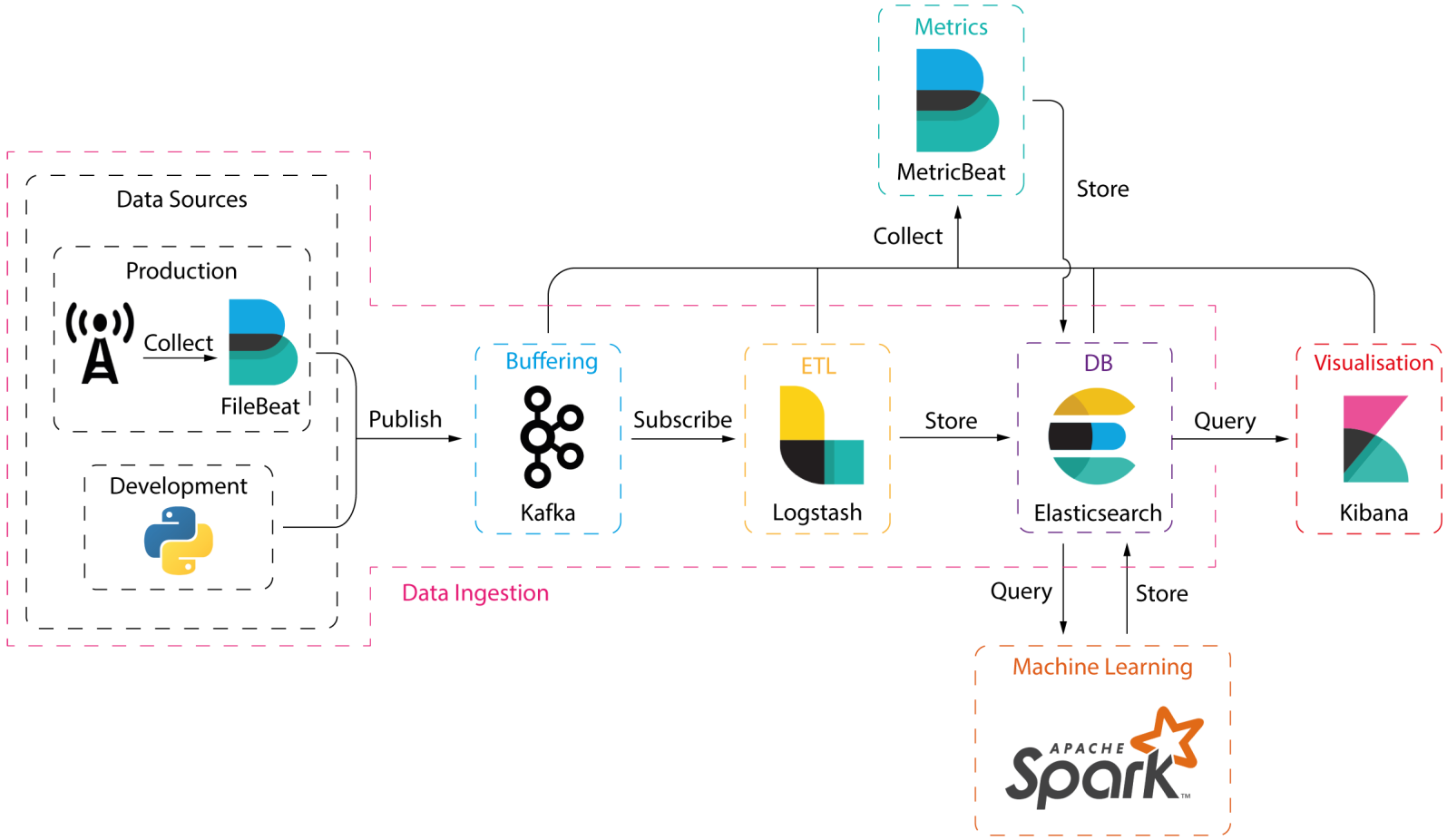

Figure 3.1: Pipeline overview.

1. *Horizontal scalability*: The pipeline should be able to handle thousands of data sources that emit many independent streaming data flows. The number of sources can change at any time, with more or fewer data producers, so also the pipeline should be able to scale accordingly. Horizontal scaling means adding more instances of an application in parallel, in contrast with Vertical scaling, when the hardware power of a single instance is incremented. Horizontal scalability is always preferable over Vertical one, as a physical machine supports a limited amount of upgrades and, more importantly, to perform power scaling, the machine must be powered down, thus the main drawback of this technique is service outage.

2. *Fault tolerance*: The pipeline should be always working, even in case of fault on systems or hardware, without losing any stored data. This requirement is fundamental since there are users that rely on the services provided by the pipeline, so no full-service outage could be tolerated in an operational environment. Not only the system should be always online, but being fault-tolerant also means that in case of a disaster no data should be lost.

3. *Concurrent access to the system*: Having thousands of sources that could potentially try to access the system at the same time could lead to data loss and missed streaming events, due to the system overload. The pipeline should be able to handle concurrent access and peaks of work without losing performance.

4. *Security*: This is one of the most important requirements since malicious attacks are becoming more and more frequent. An unsecured pipeline could lead to stolen data or a full-service outage. In an operating pipeline, guaranteeing the so-called CIA triad [15] is key. *Confidentiality* ensures that communication between the pipeline building blocks must not be read or made available to non-authorized entities. *Integrity* concerned data consistency, i.e., data should be not altered by malicious users. *Availability* consists of the ability of the system to be always working and is related to fault tolerance discussed above. Security is handled both by the application developers, which must ensure a vulnerability free software and by the software users that must apply security best

practices when deploying it and must exploit all the security features provided, like encryption on API communication or of the stored data.

5. *Near Real-Time*: The processing of a packet of data should be as fast as possible, as taking too much time could block the system. The data ingestion pipeline proposed in the thesis aims to this particular objective of near-real-time performance, with latency in the order of seconds.

6. *Parallel Execution*: The streams of data that comes from different sources can be often treated as independent, which means that different streams can be processed in parallel. Even when some aggregations between different sources is required, some parallelism can be exploited after that operation. To get the maximum speed and achieve near real-time processing, all the possible parallelism must be exploited.

In the next sections, we describe the proposed pipeline design by explaining each section individually, with particular reference to how to satisfy these requirements.

3.1.2 Notes on the pipeline development

The pipeline is fully developed using Docker as a virtualization tool, to allow fast development in a resource-constricted environment. Each application of the pipeline is already available as an image on the public registry, so no installation effort is required. To have a fast and easy boot of the entire cluster, a docker-compose file is provided with the full specification of the pipeline. Details on Docker can be found in Section 2.1.

3.2 Design of the data ingestion part

The data ingestion part supports operations from the collection of the data from the sources to the storage: it includes all the steps necessary to prepare the data before the application of machine learning algorithms and the analysis of obtained results after data processing. It is split into the following phase, each corresponding to one different application in the processing chain:

1. *Collection of data*: This is the only phase that is located outside the datacenter where the pipeline is executed. In this phase, data are extracted from the sources and sent to the pipeline. This step is performed by Filebeat, introduced in Section 2.6.

2. *Queuing and buffering*: Data are published inside a queue where they wait before being processed in the subsequent element of the chain. This is the entry point of the pipeline for the streams of data. Queueing and buffering are performed by Apache Kafka, introduced in Section 2.2.

3. *Data cleaning and enrichment (Preprocessing)*: Data inside the queue are processed by an Extract, Transform and Load (ETL) application that adds some metadata and loads them inside the designed storage. This step is performed by Logstash, introduced in Section 2.3.

4. *Data storage*: Processed data are kept in persistent storage sever, to be processed by the machine learning model or analysed with the data visualization tools. The storage application is Elasticsearch, introduced in Section 2.4.

3.2.1 Collection of data

Before entering the pipeline, the data must be extracted from the sources. There are two ways to perform this operation. The first assumes that the sources can publish data to a Kafka broker, in this case, it is sufficient to transform the data into JSON and publish them as messages on the Kafka broker (From now, we use the word message to identify the streaming events handled by Kafka, that includes the original

JSON data and some metadata used to manage the topic). This can be the case of a new application that is developed to be compatible with the pipeline, or of a legacy one that allows integration with custom-developed plugins. In the case of a legacy application that is not compatible with Kafka, Filebeat can be used. It can be installed directly on the same machine that produces data, if it runs on top of an operating system, or on a second one that is placed onsite, like a small computing device like a Raspberry Pi. Then, it allows extraction of any kind of data with one of the dozen of modules already available, from textual logs to more complex custom API polling. Filebeat directly integrates with Kafka, so, once extracted, data are converted into Kafka streams.

3.2.2 Queuing and buffering

Once extracted from the sources, data are published on a predefined topic on a Kafka broker in JSON format. Different topics could be used to split different types of sources, allowing differentiated processing in the next steps. The main purpose of placing a broker between the sources and the pipeline is to provide a buffering queue. This is necessary since, in case of peaks of incoming data or too many concurrent accesses to the system that puts an excessive load on the next steps of the chain, some data can be lost without a buffering stage. Kafka can handle thousands of concurrent publish operations on a given topic, dozens of times more than the processing capacities of the pipeline, thus making Kafka an efficient buffer stage that ensures that the incoming data are stored safely until an available application processes them. However, buffering can only handle short overloaded time intervals, otherwise, the buffer occupation grows indefinitely and the pipeline is not able to handle the streams. To achieve fault tolerance and workload distribution, in the proposed pipeline multiple brokers are clustered together and the topics are partitioned (see Section 2.2.4) to allow parallel streams processing. In practice, each data source registers as a publisher on the Kafka cluster that assigns a partition as the destination for the published data, as shown in Fig. 3.2.

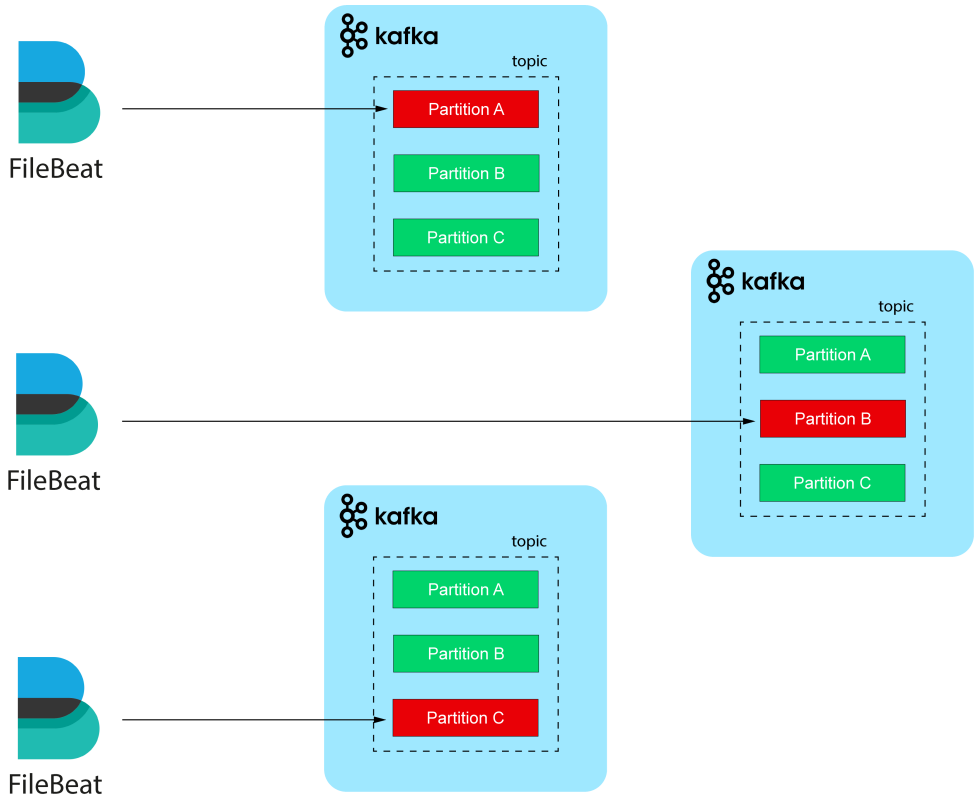

Figure 3.2: Events publish on Kafka.

3.2.3 Data cleaning and enrichment

After data has been published on the topic, they wait there until one available Logstash instance process the events. In this step three main operations, known as ETL pattern, are executed on each message in the queue:

1. *Extract*: Logstash requests the next message queued in the topic to the Kafka Broker.

2. *Transform*: Logstash applies the processing on the extracted event. Logstash allows the definition of multiple transformations steps, called *filters*, that includes common operations like parsing, anonymizing or enrichment and many more. One step that is always required to have compatibility with the machine learning model application part of the pipeline, is to define a filter that adds the *spark_processed* field to each event, as it will become fundamental to distinguish the data that have not already been processed by the machine learning model. To apply the machine learning model more complex processing steps can be required, like aggregation between multiple data streams or events. Apply them directly in this phase could create a huge bottleneck in the pipeline, so they are done before the application of the ML model when data are already in the storage, right before the machine learning model (see Section 3.3). The main scope of Logstash in this phase is to fast consume the Kafka topic and save the event into Elasticsearch to not let the queue grow too much, so only basic processing can be applied. Since no aggregation between events is performed, no other messages are needed except the one that is been processed. Also, the processing order of the events inside the queue is not important, since each of them has a timestamp field already applied by the source before publishing that allows reordering in the next steps if required. These two properties allow parallel log processing by multiple Logstash instances without requiring any coordination between nodes, as detailed in section 2.3.4. As many instances as needed could be fast instantiated by creating a new Logstash Docker container, allowing a fast reaction to workloads changes and making this solution horizontal scalable and fault-tolerant. Parallel processing is supported by Kafka topic partition,

as detailed in sections 2.2.4: by placing all the Logstash instances in the same consumer group and adding as many partitions of the topic as the number of processing nodes, Kafka automatically distributes different events on the topic to each instance, as shown in figure 3.3.

3. *Load*: Logstash stores the processed event inside the destination index on Elasticsearch. As in the initial queuing, multiple destination indexes can be defined to separate different streams. One common operation is to have separate Logstash configurations that consume different topics, apply specific processing for that stream and then store it inside a dedicated index.

3.2.4 Data storage

The destination of the data cleaning and enrichment is Elasticsearch, where Logstash sends the transformed data into the destination index exploiting the REST API. Elasticsearch is chosen since it allows fast data aggregation and search, fundamental to perform data analysis. The Elasticsearch query engine can run complex aggregations on the data by using common operators like sum, average, count and so on. This allows analysis of the data by the data science team, both through an external application using the API or by using Kibana. Details on data visualization in the pipeline on Section 3.5. Elasticsearch cluster should support concurrent access, as four applications access the storage:

1. *Data injection*: multiple instances of Logstash load data inside Elasticsearch.

2. *Data visualization*: The data science team can perform data analysis and aggregation over time mainly by using the Kibana graphical interface, which uses Elasticsearch as the back-end.

3. *Machine learning model application*: When new events arrive, Apache Spark downloads the new batch of stored events to apply the ML model, and then stores the results back.

4. *Pipeline monitoring*: The metrics of the pipeline components are loaded inside Elasticsearch for monitoring purposes, where system

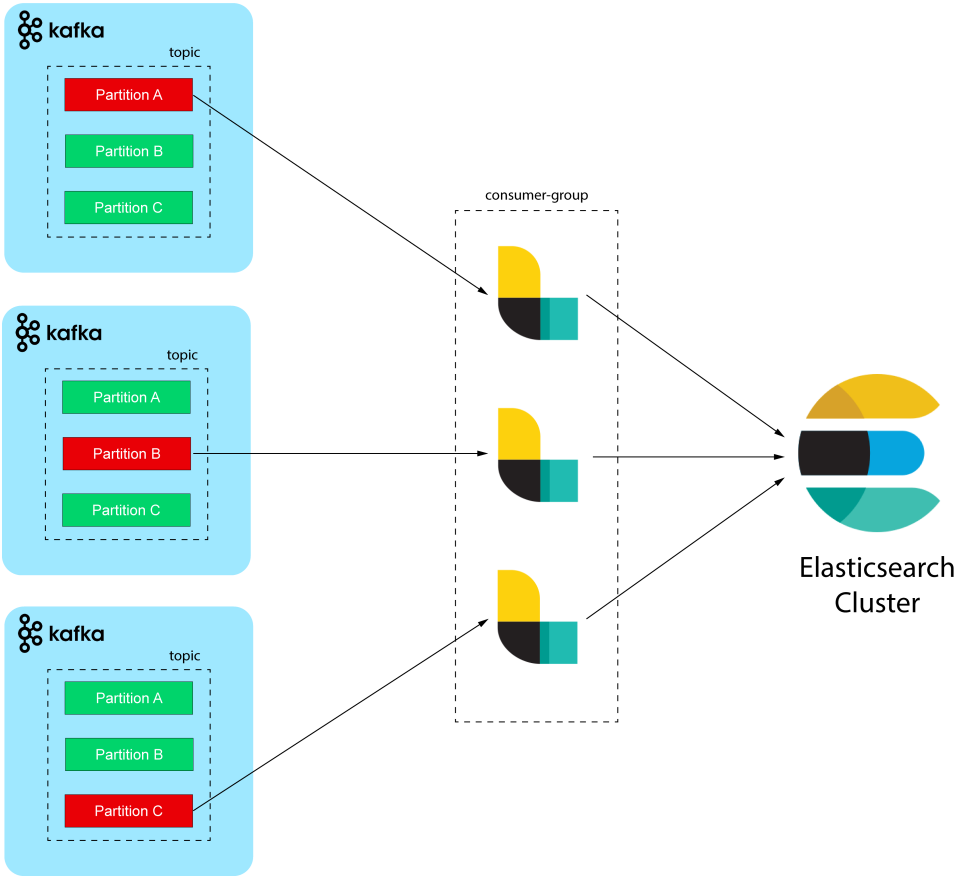

Figure 3.3: Parallel processing of topic.

administrators can explore the collected data using a dedicated Kibana dashboard.

This huge amount of concurrent access could move the bottleneck from the processing phase to the storage one. To avoid this, data storage is handled in a distributed way on multiple nodes that shares the workload and the data. Fault tolerance is ensured by the native Elasticsearch sharding mechanism (see Section 2.4.3): in case of one node failure, there is no data loss.

3.3 Design of machine learning model application part of the pipeline

In the previous section, we described how the streams of data flow from the sources to the persistent storage inside Elasticsearch. Once some events have been stored, they are transferred to the prediction server to be processed by the machine learning model. This kind of processing is called *batch*. The most common operations performed on streaming events are either classification when the events are assigned to one discrete class of a predefined set, or regression when a real number is associated to the data stream as an output of the ML model, that has been already trained on smaller datasets. In the proposed pipeline Apache Spark is used as framework and computational engine to apply machine learning to the data. Spark allows distributed computation on many nodes, which translates into fast data preprocessing and prediction, allowing near real-time ML model application. The distributed execution makes also the prediction cluster fault-tolerant to failures and outages. Since Sparks applications are written in Python, the framework supports all the existing libraries like *Tensorflow* and *Scikit-Learn*, so the same tools used in development can be also used in the operational environment exploiting the power of the Spark optimization engine that automatically determines which tasks could be executed in parallel and distributes them among Spark nodes in the cluster. The Spark application that runs the ML model is where most customization is needed to adapt it to the use case. This is in some way unavoidable as any machine learning model is different from the others, especially in the data preprocessing phase, which cannot be abstracted from the particular data of the considered use case. The only operation that is equal in any use case is the transfer of data from Elasticsearch to Spark and the writing of machine learning outcomes back into the storage. In general, the Spark application that runs the ML model inside the pipeline is constituted by the following steps, that can be used as a guideline to develop an application that suits the considered use case:

1. *Request new data*: Spark request a new batch of data to Elasticsearch from the index where they have been stored by Logstash. The acquisition of data is done using the Elasticsearch-Spark library by Elastic [27], as Spark does not support Elasticsearch out

of the box. To determine the new data batch to collect and avoid duplicate processing the field of the log *spark_processed* is checked to be false during the query to the Elasticsearch index, and it is set to true after the computation. Collected data are automatically converted into a Spark dataframe, the tabular format where SparkSQL operations can be applied.

2. *Preprocessing*: Preprocessing of the acquired data to prepare them for the ML model application. This step is the most use-case-dependent and requires huge customization: common operations are scaling around mean and variance, one-hot encoding of labels, handling "null" values. Preprocessing should be done using the SparkSQL API where possible, as they are the most efficient Spark API, avoiding the as less efficient user-defined functions [40], but this could be unfeasible as many common preprocessing operations are not available in SQL. More details can be found in Section 2.7.

3. *Prediction*: The ML model is applied to the data. This phase is quite standard and dependent on the chosen machine learning library. It must be coded inside a Spark user-defined function, that is applied to each row of the preprocessing dataframe.

4. *Save results*: Spark request Elasticsearch to update the original batch data with the machine learning results. The operation called *upsert* is performed on the Elasticsearch index: Spark checks if each document needs to be updated, and updates only the fields that have been changed. This is faster than sending back the full processed batch. After these steps, Spark downloads a new batch and performs all the operations again from the beginning.

3.4 Performance metrics and health monitoring

Monitor the performance and the health of running applications is fundamental in any operational environment to detect and prevent malfunctioning on the deployed software. Monitoring the performance of an application means collecting metrics about its functioning and about the hardware resources that it is using, while with the health of a distributed system, we refer to the working status of each software application. An application is healthy when it is not in an error state and when it can communicate with the other nodes in the cluster. In a distributed software pipeline, two main problems could arise and must be detected as soon as possible [13]:

- *Fault of an instance*: Any application can stop working at any time. This does not result in a complete service outage, since all the clustered deployment integrates some kind of fault protection, but there is a limit in the number of instances that can simultaneously fail inside the cluster without having a full outage. So we should be able to fast detect faulty containers, and fast replace them with a new instance. This issue is partially solved by the Docker engine, which can detect containers where the running application is in an error state and fast run a substitute one.

- *Bottlenecks*: This is a problem that arises when an application processing rate is less than the produced data rate. When data processing components are not able to handle all the received data, some of them are discarded, thus causing service degradation. A queuing system like Apache Kafka can limit this problem but only for small time intervals, but if this persists the topic queue will grow fast and can overflow. This problem is easily solved if detected earlier with horizontal scaling of the application that is causing the bottleneck.

Metrics collection is performed using Metricbeat, introduced in section 2.6, the collected data are sent to Elasticsearch and they can be visualised in a Kibana dashboard. In a real operational environment it is common practice [33] to separate the cluster that is monitored, in our case all the pipeline, and the monitoring cluster, that is in charge

of handling only the metrics collected. Since this leads to two different Elasticsearch-Kibana stacks and we are working in a resource constraint environment, we reuse the already-deployed instances and put the metrics in a dedicated index with a dedicated dashboard to visualise results. Metrics visualization is explained in Section 3.5.3.

3.4.1 Available metrics

The collected data includes common metrics like memory and disk usage, CPU load and metrics specific to the monitored applications. Metricbeats provides modules [32] that allow the collection of metrics from the following applications of the pipeline:

- *Docker metrics*: They include all the statistics on the host that is running the Docker engine, as well as on the running containers. This module is fundamental to detect which containers are in an error state as well as how many resources are available for adding new instances of any application.

- *Kafka metrics*: They include statistics like the topic queue size, the publish event rate, the connected publishers and subscribers and the network bandwidth used by the application. The most important one for bottlenecks detection is the *consumer group lag* metric: it is defined as the difference between the total amount of events in the queue and the offset (current index in the queue) of the consumer group and corresponds to the number of messages that the consumer group has still to process. Since the consumer group in the pipeline is composed of Logstash applications that store the data inside Elasticsearch, the consumer group lag is the most important indicator of a bottleneck in the data ingestion part of the pipeline. It should always ensure that the average lag is the smallest as possible, else there could be an excessive load either on Logstash or Elasticsearch and scaling of these applications could be required.

- *Logstash metrics*: They reports three main metrics, *events received rate*, the rate of new logs downloaded from Kafka, *events emitted rate*, the rate of logs saved inside Elasticsearch and *events latency*, the average time required to process a log. By analysing these

statistics it could be determined if Logstash has some problems: in particular, events received rate less than the incoming message rate of Kafka can mean that Logstash is a bottleneck. Also, a huge events latency could mean that some of the processing steps of Logstash are abnormally taking too long to complete. Also, a breakdown of the transformation steps applied to data is provided inside the Stack Management area of Kibana (Details in 3.5.3), allowing an easy fix of these problems.

- *Elasticsearch metrics*: They include documents count for each index, storage usage for the nodes, rate of data query, loads of all the nodes and many others. These metrics allow detecting if additional nodes are needed to support the ingestion. Also, the health status of the indexes in the cluster is reported. Each index can be in three health statuses:

  - *Green*: If the shards of the index are replicated on many nodes. This is the desired state that allows fault tolerance in case of failure.
  - *Yellow*: All the primary shards are assigned, but one or more replica shards are unassigned and, if a node in the cluster fails, some data could be unavailable. When an index is on this status, the Elasticsearch master tries to rebuild the shard replica and sends them to a node in the cluster.
  - *Red*: This indicates a severe data loss and coincides with the total unavailability of the index. This condition is irreversible and needs recovery from a backup.

  To ensure that Elasticsearch is healthy, all the indexes must always be in the green states, if a persistent yellow state is reported maintenance to replace failed nodes can be required.

- *Kibana metrics*: They reports statistics of web access to dashboards and the latency to serve web pages. If many users rely on Kibana to visualise the data, adding more instances can improve the performance and speed of the data visualization web front-end.

- *Metricbeat metrics*: Metricbeat can self-monitor, reporting failures in collecting metrics from other applications.

- *Spark metrics*: Metricbeat is not compatible with Spark, but to overcome this issue the Spark Web UI can be used. It reports detailed information on the running jobs and task execution on each node, allows DAG visualization, as well as the time required to perform each stage. To detect Spark bottlenecks, other than web UI, the custom *Spark lag* metric is introduced. It is defined as the count of data that have the *spark_processed* field false, which corresponds to the events that are still to process with the ML model. It has a role similar to the Kafka consumer group lag for topics queues in the detection of bottlenecks but for the machine learning model application part of the pipeline. If this metric is too high and constantly increase, a scaling of the Spark cluster is needed. Finally, to measure the performance of the pipeline, the *latency* metrics is defined as the time required to fully process one log from the data ingestion to the prediction. It is calculated as the difference of the timestamp added before publishing the event to the pipeline from the source application and the time at when Spark applies the ML model and is calculated by the Spark application before sending classification results back. Latency metrics are used to test the proposed design, so they are better detailed in the experimental results chapter. These two metrics can be simply inserted into any kind of Spark application for any pipeline use case.

3.5 Data visualization

What we have discussed until now is the so-called *back-end*, and constitutes the core of the pipeline functionality, while data visualization is the *front-end*, the point where users can access all the pipeline functionality in an easy-to-use graphical interface. The front-end must be as simple as possible, to completely hide the complexity of the back-end. Users may not be interested in the details of the pipeline implementation, but they should be able to comprehensively exploit all the pipeline functionality. Data visualization allows access to the three data types available inside the data pipeline, described in the previous chapters:

- *Stream visualization*: Visualise the events that are stored inside the pipeline on graphs and charts to perform real-time or historical analysis of the data streams.
- *Machine learning visualization*: Visualise the results of the ML model application.
- *Metrics visualization*: Visualise the collected metrics and monitor the status of the pipeline.

The application used for visualization is Kibana, introduced in Section 2.5, which works as the front-end of Elasticsearch, which executes queries and aggregation and returns the data. Visualization of the data is done through dashboards, populated with widgets like charts and graphs, that can be customized to suit the use case. On the same dashboard, multiple data categories can be mixed and different dashboards can be created to differentiate users, to provide only data streams data to non-technical users or pipeline metrics to the administrators. The dashboards and their widgets are use case dependent and need to be designed accordingly: Kibana allows easy dashboard creation with an intuitive drag and drops approach.

3.5.1 Stream visualization

Analysing raw data can be useful both to the non-developer and to data scientist: the former can visualize key insights of the streamed data, for example, in an IoT use case, visualizing real-time or historical sensors measures, while the latter can find complex patterns to exploit in

machine learning model design. Kibana allows both the use cases: Data scientists can use the Discover section to perform free exploration on the data using aggregation with functions like count, average, variance, correlation and visualize results on charts and graphs, while for everyday operation and monitoring a dedicated dashboard can be populated with widgets that perform some pre-registered queries and aggregation previously designed in Discover.

3.5.2 Machine learning visualization

Detailed ML model outcomes can be visualized on any kind of widget. The design of visualization of machine learning is use case dependent, mostly dependent on the type of ML model deployed.

3.5.3 Metrics visualization

Metrics collected through Metricbeats can be analysed to help in the detection of failures or bottlenecks. Kibana provides two ways to visualise metrics:

- *Stack Monitoring*: Metrics from the ELK stack (Elasticsearch, Logstash, Kibana and Metricbeats) benefits of powerful predefined instruments, and no custom dashboards are required. In this section, the status of every application can be monitored, from the CPU, RAM and disk statistics to application-specific ones. Logstash is also provided with a tool to analyse the latency of each step of the transformation phase.

- *Custom dashboards*: For other applications, the metrics can be handled through the widgets in the dashboards. The only application in the pipeline that cannot benefit from Stack Monitoring is Apache Kafka, but we made a dedicated dashboard to show the main parameters.

3.6 Summary

In this chapter, we explained the proposed pipeline architecture, from data collection to machine learning ML model application on the col-

lected data, concluding with data visualization and performance monitoring. In the next chapter, we show a reference implementation considering the failure management in microwave networks use case.

Chapter 4

Design of the pipeline for failure management in microwave networks

In this chapter, we explain the failure management in microwave networks problem, and we provide a reference implementation of the pipeline on this use case. Since the logs provided by the equipment can be considered as streamed data, they are an ideal example for testing our design. however, the proposed pipeline design can be applied to any case that involves machine learning on data streams. A deeper treat of failure management in microwave networks topic can be found on [17], used also as a reference for the customization of the general pipeline architecture. After the introduction to the topic, we explain how the ML model developed to solve this problem is deployed in an operational environment, exploiting the pipeline features. For each section of the pipeline, introduced in the previous chapter, it is provided how it has been customized to suit the fault management in microwave network use case.

4.1 Machine-Learning-Based Failure management in microwave networks

Failure in microwave networks involves the detection of failures inside a network composed of microwaves links and the subsequent classification

of the failure. A microwave radio link is a bidirectional point-to-point connection between two geographical points, performed using equipment placed in the line of sight that transmits on a high frequency [4]. The network is a real one deployed in Italy with equipment provided by SIAE Microelettronica company, composed of 10841 links that provide status and power measurement logs once every 15 minutes in a streaming fashion.

4.1.1 Input dataset

To test the functionality of the pipeline, a dataset of pre-registered logs is employed to emulate the network. The information provided by the links in the logs are:

- *Measurement information*: Information that identifies the link that produces the log like the unique id of the link, branch (direction) of the measure, IP addresses of both ends of the link.

- *Project information*: Information of the settings of the link, fixed in time. Contains equipment type, lowest and highest modulation formats allowed in the link, ACM (Adaptive Code Modulation) settings, nominal received and transmitted power, protection scheme adopted, frequency and bandwidth of the link.

- *G.828 performances*: These fields contain information that conforms to the G.828 ITU standard [35], which describes the main performance indicator to measure performance and availability of radio links. In our use case three indicators are reported:

  - *ES, Errored Seconds*: Number of one-second periods with blocks of received data that have at least one error.
  - *SES, Severely Errored Seconds*: Number of one second periods where at least 30% of the blocks have error.
  - *UAS, Unavailability Seconds*: When ten or more consecutive SES are detected, the link is declared unavailable. This field measures the number of seconds in unavailable status.

- *Propagation measures*: Contain, for both the transmission directions of the link, the minimum and maximum transmitted and

received power and the lowest modulation reached in the 15 minutes window.

- *Quality flags*: Three flags indicate if the measures contained in the log are reliable. Only the logs with all flags reporting correct measurement are processed.

- *Last second of the window measures*: They provide the signal-to-noise-ratio, the modulation used and the received power in the last second before producing the log.

Table 4.1 shows the most important fields reported by the links.

Table 4.1: Selection of data reported by the links.

Reported data		
Parameter	**Type**	**Description**
Idlink	Measurement information	Unique identifier of the link
Ramo	Measurement information	Identify the side of the link, A(0) or B(1)
Data	Measurement information	Time and date of log production
ip_a	Measurement information	IP address of side A
ip_b	Measurement information	IP address of side B
Ptx	Project information	Nominal transmitted power when the minimum modulation format is used (dBm)
freqband	Project information	Link frequency
bandwidth	Project information	Link bandwidth
protection	Project information	Protection technique on link
LowThr	Project information	Minimum received power tolerated on the link with any modulation format used (dBm)
Ptx	Project information	Nominal transmitted power when the minimum modulation format is used (dBm)
Thr_min	Project information	Minimum received power threshold tolerated by the link with its current modulation format (dBm)
RxNominal	Project information	Nominal received power at the maximum modulation format (dBm)
acmEngine	Project information	Flag that indicates if the ACM is enabled on a given microwave link

ES	G.828	Number of one-second periods with blocks of received data that have at least one error
SES	G.828	Number of one second periods where at least 30% of the blocks have error
txMaxA	Propagation measures	Maximum power transmitted from site A in in the 15-minutes slot (dBm)
txminA	Propagation measures	Minimum power transmitted from site A in the 15-minutes slot (dBm)
rxmaxA	Propagation measures	Maximum power received at site A in the 15-minutes slot (dBm)
rxminA	Propagation measures	Minimum power received at site A in the 15-minutes slot (dBm)
txMaxB	Propagation measures	Maximum power transmitted from site B in in the 15-minutes slot (dBm)
txminB	Propagation measures	Minimum power transmitted from site B in the 15-minutes slot (dBm)
rxmaxB	Propagation measures	Maximum power received at site B in the 15-minutes slot (dBm)
rxminB	Propagation measures	Minimum power received at site B in the 15-minutes slot (dBm)
acmMax	Propagation measures	Lowest modulation reached in the 15 minutes measures

4.1.2 Machine learning model

The ML model used to perform classification has been developed on a labelled subset of the previously described dataset. The labels are applied by domain experts to time windows of 45 minutes, i.e., consisting of measures collected in three consecutive 15-minutes slots. Only windows that report a UAS in the last slot are used for ML model training since failures only occur if there has been an error. Therefore, the same rationale is applied when using the ML model "in-field", i.e., only windows that report a UAS in the last position are classified. The following

six possible labels are used in the classifier, and correspond to different failures in the radio equipment:

- *Class 0, Deep Fading*: It is a severe channel attenuation, that makes communication between the two radio equipment impossible. This condition is random and can depend on geographical position, seasonality or weather condition. Some of the possible causes are the loss of the line of sight between the equipment, caused for example by the vegetation, or severely adverse weather conditions, that especially at higher frequencies can interfere with the radio signal. Most of the time deep fading resolves alone, but if it persists a field intervention could be required, for example, to cut the vegetation.

- *Class 1, Extra attenuation*: In normal working conditions the transmitted power should match the received one, with small differences possible. When the gap is too high, the link is in the extra attenuation status: this usually required technical intervention to fix the problem, either on the field or remotely.

- *Class 2, Interference*: Interference is caused by other radio sources that operate on a frequency band that overlaps with the link one, making the signal noisy. This condition could be temporary, but if it is persistent it could be needed to change the carrier frequency of the link.

- *Class 3, Low margin*: A generic problem that is always caused by human error in the configuration of the equipment, that has not been performed as recommended by the producer. It could be easily corrected by changing the equipment configuration.

- *Class 4, Self-interference*: The link transmission is full-duplex, this means that the communication is bidirectional and both sides of the link have a transmitter and a receiver. In normal working condition, the two radio signal does not interfere each other since they are on a different band. Sometimes, due to non-linearity in the filtering electronic components, some frequencies that belong to the other radio path could be transmitted, causing self-interference to the receiver on the same side. This problem always

arises as a consequence of bad link design and need intervention to be fixed.

- *Class 5, Hardware failures*: This class includes all the generic equipment faults that are not easily recognizable by just looking at the radio measurements. They could be both temporary or permanents failures, and they require human intervention, often on the field, to locate the problem.

The ML model that has been developed is based on artificial neural networks, with one hidden layer of 50 neurons. The framework used for development is Scikit-Learn on Python. In order to be used, the ML model requires preprocessing of the inputs logs. This is done sequentially by applying the following steps:

1. *Logs Windowing*: The ML model requires windows of three logs as input, with all the logs belonging to the same link. This corresponds to 45 minutes of measures, where the last one must present a UAS greater than zero. Windowing is performed by simply concatenating all the columns of the logs data. At the end of this phase, useless information like link identifier and date and time are discarded.

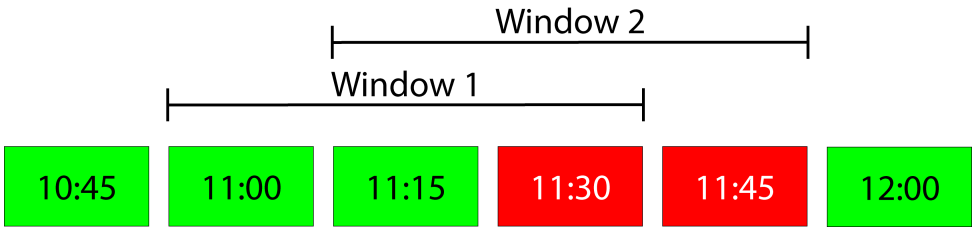

Figure 4.1: Log windows. Green log reports no error, while red one have at least one UAS second.

2. *Encoding of string attributes*: The type of modulation used by the link is transformed into an integer value, i.e., one progressive number is assigned to each of the possible modulation types.

3. *Fill missing values*: Missing power measurement values are forced to -150 dBm and 100 dBm for minimum/maximum received and transmitted power, respectively. These values are chosen far from the real minimum/maximum values so that the ML algorithm can handle numerical values, still allowing the ML model to capture the situation that such values have not been correctly collected in the system.

4. *Features normalization*: Each feature of the dataset is scaled around the mean and the variance, leading them to have a distribution centred around zero and with variance between -1 and 1. This greatly helps during training the ML model, as machine learning algorithms perform better on normalized data [5].

4.2 Customization of the pipeline for failure management in microwave networks

In the next part, we show the customization of the pipeline to suit the failure management in microwave networks use case. Each part refers to the corresponding one introduced in the previous chapter.

4.2.1 Design of data ingestion

In this section, we focus on how we can configure the data ingestion part of the pipeline to suit this specific use case. Figure 4.2 shows the data ingestion part of the pipeline customized to suit the failure management in microwave network use case.

Data sources

To deal with the failure management in microwave network use case, small customization is required to the data ingestion part of the pipeline. Most of the work has to be done on the sources, as they must be adapted to the Kafka interface provided by the pipeline. In an operational environment, the network equipment could store the logs as files inside a folder or send them to a centralized network controller. To support this and also other possible cases, Filebeat could be used, since it allows to get the logs using already available modules, like the file module

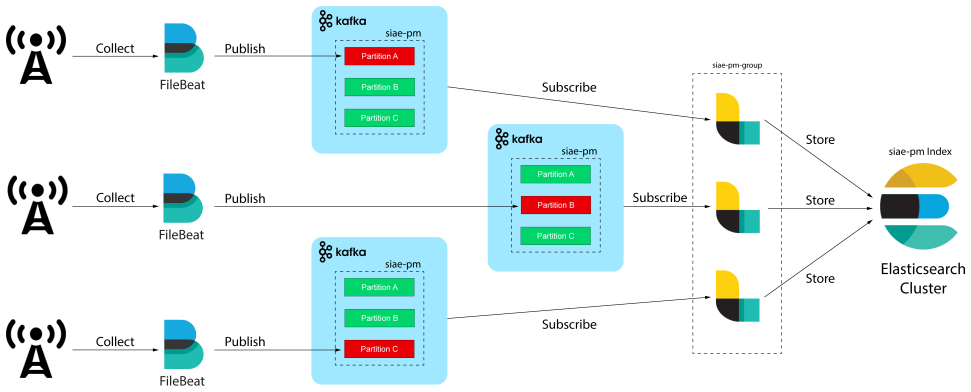

Figure 4.2: Data ingestion part of the pipeline for failure management in microwave network use case.

or a custom one for specific equipment API. In the development of the pipeline, we use the dataset described in section 4.1.1 to execute simulations of the pipeline behaviour without needing any radio equipment, since simulating with a huge number of physical hardware would be unpractical with the hw/sw used in this work. Using this dataset, we could potentially test the pipeline with 10841 different sources in a very simple way. To simulate the log production by each link, we develop a Python application, called Netsim, that injects the events in the pipeline. Netsim allows choosing the number of links that produce logs and the interval between each event, e.g., to simulate intervals smaller than the original 15 minutes and perform a sort of "stress-test" for the pipeline. To run Netsim, a folder with a CSV file for each of the links is required, then the application sends each log to the Kafka broker as a JSON message, emulating what would be done by Filebeat in an operational environment.

Kafka and Elasticsearch configuration

Kafka and Elasticsearch are quite general-purpose, so the standard configuration works well for most of the use cases. The configuration is only

updated with the correct topic and index name, in this case, both called *siae-pm*. Finally, it should be configured the number of partitions of the topic, that should match the number of Logstash instances to exploit the parallel topic processing.

Logstash configuration

Logstash requires the configuration of all the three stages of ETL, however, extraction and load are the same for every use case and they only require the modification of the source topic and the destination index, which should match the Kafka and Elasticsearch configuration. The transform phase is the one that requires to be customized and adapted to the specific use case at hand. In our case, the following operations are performed:

1. The *"date"* field, including date and time of the log, is parsed and converted to the date format accepted by Elasticsearch, which stores times in the Unix format. Then, the field is copied in the *@timestamp* field. These steps are fundamental since Elasticsearch requires the *@timestamp* field to perform aggregation on time windows.

2. Quality flags fields are removed since they are useless for our purpose.

3. Two fields to support machine learning classification are added: *spark_processed* of type Boolean, initialized to false, that will be used to distinguish which logs have been already classified by the machine learning model, and *prediction* field of type integer, initialized to -1 (means not classified), that will contain the predicted class of failure.

4.2.2 Deployment of the model

Deploying the ML model in the pipeline as a Spark application is the most use-case dependant step, as every ML model requires different prepossessing steps. In the next paragraphs, we describe how the Spark application for the failure management in microwave networks ML model is implemented, following the guide steps defined in the previous chapter.

Request new data

The logs that are still to be processed are stored inside the *siae-pm* index of Elasticsearch after been injected by Logstash. The logs that have the field *spark_processed* equal to false are collected inside a Spark dataframe. Figure 4.3 shows an example of a new batch that is requested from Elasticsearch, for a simpler visualization some fields are omitted. The queried logs are automatically collected inside a Spark dataframe in an SQL-like table.

Data	Idlink	Pwr/Uas..	Processed	Prediction
10:00	1	...	false	-1
10:00	2	...	false	-1
10:00	3	...	false	-1
10:15	1	...	false	-1
10:15	2	...	false	-1
10:15	3	...	false	-1
10:30	1	...	false	-1
10:30	2	...	false	-1

Figure 4.3: Datafrae created with the new batch queried from Elasticsearch.

Data preprocessing

This is the phase where most of the work is done inside Spark. After this phase, the data are clean and normalised, ready to be classified. Preprocessing phases are the same as development, but some optimization is applied to achieve better performance and make them compatible with the batching system:

1. *Aggregation*: This is the most computationally expensive step, and

it is not required when developing the ML model since the data are already grouped by link identifier. Logs are grouped based on the *idlink* (identifier of the link) and *branch* (identifier of the direction of the communication, two directions per link), then they are ordered according to date and time. This step requires Spark to move data between the nodes to make the groups, so it is also the slowest one and cannot be executed in parallel with any of the next phases. To aggregate the logs, the *groupBy* and the *OrderBy* SparkSLQ functions are used. After this step, a new dataframe is created along with the original one. All the next steps are applied to this new dataframe, referred to as *preprocessed dataframe*, and the original data are kept separate to be used in the last phase. Figure 4.4 shows the aggregation of logs, to better visualize we consider a link with only one communication direction and on red we highlight the log that presents a UAS, we also consider the case of link 3 not reporting the last time window to explain how the ML model handles incomplete data situations.

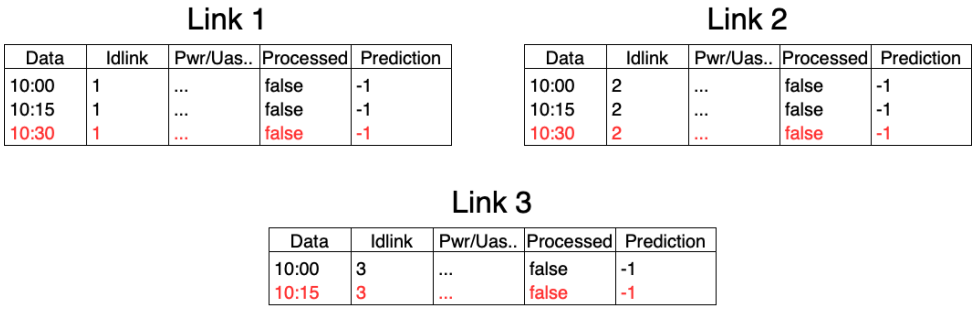

Figure 4.4: Aggregation of logs.

2. *Filtering*: Since the time windows must be composed of three logs each, groups with less than three members are discarded. Also, groups that do not report any UAS are not processed. This step requires two operations: first, the *count* aggregation function is applied to each of the groups, then the *where* operator filters groups with less than three logs. Figure 4.4 shows the filter phase:

link three is discarded as it has only two logs.

Link 1

Data	Idlink	Pwr/Uas..	Processed	Prediction
10:00	1	...	false	-1
10:15	1	...	false	-1
10:30	1	...	false	-1

Link 2

Data	Idlink	Pwr/Uas..	Processed	Prediction
10:00	2	...	false	-1
10:15	2	...	false	-1
10:30	2	...	false	-1

Figure 4.5: Filtering.

3. *Make windows*: The logs of each group are concatenated to form three logs window, where in the last one a UAS event must be present. This step is executed in parallel on multiple nodes, each handling a subset of different groups, as each group is independent of the others. In this phase, windows that do not present any UAS in the last 15 minutes are discarded. As a result, it remains a subset of the original dataframe with valid windows. This step cannot be executed exploiting any SQL operator and it is executed inside a user-defined function.

Link 1 - Window

DataN	DataN-1	DataN-2	Idlink	Pwr/UasN..	Pwr/UasN-1..	Pwr/UasN-2..	Processed	Prediction
10:30	10:15	10:00	1	...	...	...	false	-1

Link 2 - Window

DataN	DataN-1	DataN-2	Idlink	Pwr/UasN..	Pwr/UasN-1..	Pwr/UasN-2..	Processed	Prediction
10:30	10:15	10:00	2	...	...	...	false	-1

Figure 4.6: Windowing.

4. *Encoding of string attributes, Adding non reported features, Fill missing values, Data Normalization*: These steps are the same as those executed at the development stage, and they are performed only on the valid windows (the one that presents a UAS on the last log) created in the previous phase. Since these steps cannot

be executed exploiting SQL operators, they are executed inside a user-defined function. Scaling of data is performed using the mean and variance of the development dataset.

Classification

Now the preprocessed dataframe is ready to be classified. This step is executed inside a user-defined function on multiple nodes, since now each time window is independent of the others, allowing a very fast application of the ML model. Classification is performed using the scikit-learn library: the ML model is previously trained on a development environment, using a fixed-size dataset and then it is exported and broadcasted to all the Spark cluster members, where it is finally used to do classification. After this phase, the result is the *predicted dataframe*, which adds a new column with an integer between zero and five that corresponds to the class of failure to the preprocessed dataframe. Figure 4.7 shows the window with the prediction applied.

Link 1 - Predicted

DataN	DataN-1	DataN-2	Idlink	Pwr/UasN..	Pwr/UasN-1..	Pwr/UasN-2..	Processed	Prediction
10:30	10:15	10:00	1	...	...	...	false	4

Link 2 - Predicted

DataN	DataN-1	DataN-2	Idlink	Pwr/UasN..	Pwr/UasN-1..	Pwr/UasN-2..	Processed	Prediction
10:30	10:15	10:00	2	...	...	...	false	1

Figure 4.7: Classification.

Save results

The only interesting column of the predicted dataframe is the prediction one, all the others are discarded. To add the prediction column to the original data, a join between the original dataframe and the predicted dataframe is performed, using as a key the unique identifier that Elasticsearch adds to each document during the injection phase. As a result

of the join, the original data now have the class label in the *prediction* field. After this step, the logs are marked as processed by changing the field *spark_processed* to true, with an important exception: for each group, the last two logs are marked as not processed, since on the next batch they are required to form the first window of each group. Finally, the results are stored back inside Elasticsearch. Figure 4.8 shows the results of the join on the original and the predicted dataset, ready to be stored back. The *spark_processed* field is set to true only on the first log of each link, as explained before. Also on link three, no flag is changed since it needs to wait for another log to make a window, as now the logs are less than three.

Data	Idlink	Pwr/Uas..	Processed	Prediction
10:00	1	...	true	-1
10:00	2	...	true	-1
10:00	3	...	false	-1
10:15	1	...	false	-1
10:15	2	...	false	-1
10:15	3	...	false	-1
10:30	1	...	false	4
10:30	2	...	false	1

Figure 4.8: Final dataframe with machine learning results.

4.2.3 Design of data visualization

For the failure management in microwave network, we made a dashboard that can be used as support for everyday operation, shown in figure 4.9: on the left side information on the network and time evolution of power metrics are shown, while on the right side of the dashboard network administrators can visualise the real-time status through a widget that

shows the last available prediction for each link, as well as the number of links that reports an error.

4.3 Summary

In this chapter, we introduce failure management in microwave network use case and we show a reference implementation of the pipeline on this specific issue. In the next chapter, we present the results of the test made on this use case pipeline implementation.

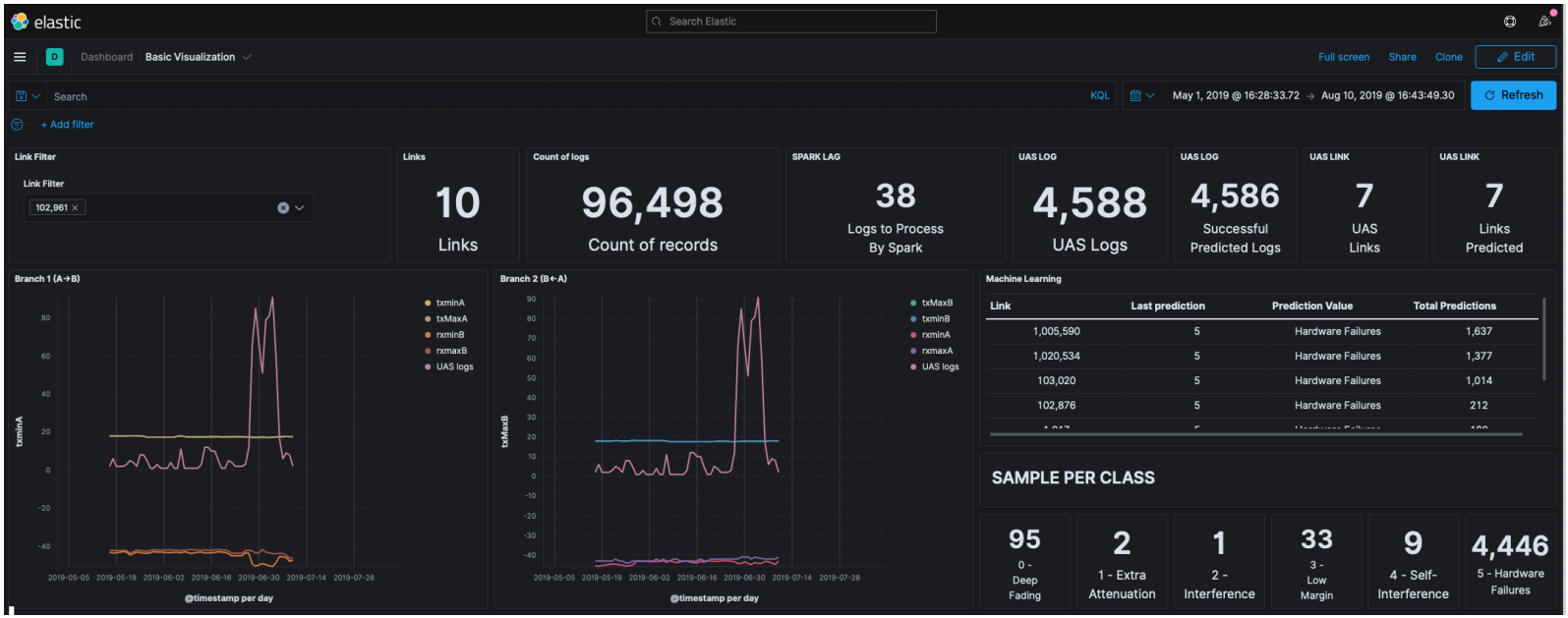

Figure 4.9: Data visualization dashboard.

Chapter 5

Experimental results

In this chapter, we evaluate the performance of the proposed pipeline architecture, in terms of latency, and using as a test use case the failure management in microwave networks.

5.1 Setup

To present the experimental results, we consider the failure management in microwave networks use case, discussed in Chapter 4. The data sources are the registration of real logs provided by deployed equipment, described in Section 4.1.1, which allows simulating a network with up to ten thousand links. Simulation of operational environment is done using *netsim*, introduced in Section 4.2.1. This allows us to simulate the real-time ingestion from a tunable number of network devices, with a custom interval between each event in the data stream. The pipeline has been deployed on a c5a.8xlarge EC2 instance on Amazon AWS, with the following hardware:

- 32 virtual core based on AMD EPYC 7002
- 64 GiB of RAM
- 50 GB of SSD storage

This setup is highly optimized for compute-bound applications that benefit from high-performance processors [23], so it perfectly fits our use case that involves data processing through Logstash and Apache

Spark. All the tests are executed on the full version of the pipeline, which includes replication for all the main applications involved. In particular, the following containers are deployed:

- 3x Elasticsearch.
- 3x Kibana.
- 3x Logstash.
- 1x Metricbeat.
- 1x Zookeeper: required by Kafka to manage the cluster.
- 3x Kafka brokers.
- 1x Spark master.
- 3x Spark workers.
- 1x Spark Driver, running the custom application that executes the machine learning classifiers.

Also, to avoid issues with bad clock synchronization and to discard network latency, Netsim is run on the same machine but outside the Docker environment.

5.2 Performance indicators

As for performance indicators, we consider the average latency of the logs that are processed by the pipeline. We define the *end-to-end* latency for the processing of a single log as the time interval between the creation of the log by Netsim and the classification of the log by the Spark cluster. This comprehensive metric is then broken down into the following latency contributions:

- *Kafka latency*: Time spent in the topic queue before a Logstash instance can consume the log.
- *Logstash latency*: Time spent inside Logstash to pre-process the logs and store them into Elasticsearch.

- *Spark latency*: This metric includes the time that is required by Spark to classify the failure with the machine learning model. It also includes the time that is spent in Elasticsearch before Spark can process the new batch. Spark latency value is derived from the end-to-end latency minus the other metrics.

Figure 5.1 shows the breakdown metrics referred to the corresponding pipeline components. Our evaluation is performed considering an increasing number of microwave links that transfer logs at the pipeline input. Each published log presents UAS so that all the windows are valid and processed by the machine learning model. For each processed log, we evaluate the end-to-end latency and its corresponding breakdown, and in the results, we show the average latency of all the processed logs. To avoid bottlenecks, for each test the interval between the publishing of two batches has been set to a value at least twice the average end-to-end latency.

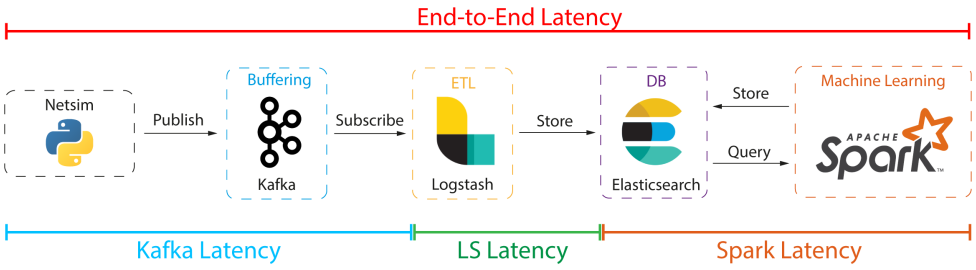

Figure 5.1: Correspondence between latency and the pipeline.

5.2.1 End-to-end latency results

Figure 5.2 shows the values of the end-to-end latency values as a function of the number of active links that publish logs on the pipeline. As expected, the more active links the more time is required to process a new batch by the system. The plot makes evident the linear dependency between the number of client links and the end-to-end latency of the

pipeline. Note that this result is highly dependent on the used hardware that runs the pipeline. Data shows that even with ten thousand links, i.e., corresponding to the overall set of links in the considered microwave network, we observe that end-to-end latency is thirty times lower than the fifteen minutes original intervals between two consecutive logs: this means that for this use case the pipeline can handle even more links than the available ones and that the machine learning results are fast available for further analysis by the network operator.

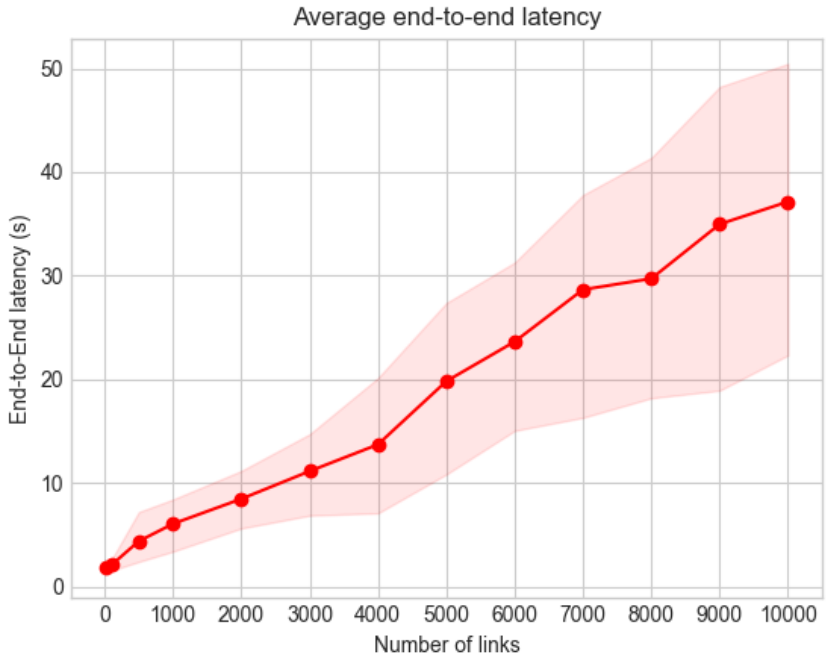

Figure 5.2: End-to-end latency plot, with the 10%-90% percentile interval in light red.

5.2.2 Breakdown of the end-to-end latency

To better understand which section and components of the pipeline most contribute to the end-to-end latency, we now analyze the breakdown into the various latency contributions. The first contribution is Kafka latency, which measures the time spent in the queue before a

Logstash instance processes the log. We can see in the plot in figure 5.3 the average latency as a function of the number of links. We first notice that the average Kafka latency tends to converge to a stable value as the number of links increase. The light blue area in the plot shows the interval that contains most of the values, i.e., the interval between the 10% and 90% percentile: this interval is quite high, meaning that the values are unequally distributed. To explain this large variance, we can look at Figure 5.4, which shows the Kafka latency of some consecutive logs consumed from the topic for the case of ten thousand data sources. In a balanced queuing system, we expect a latency that is first increasing and then convergent to a value, while in this case, we can clearly see a periodic oscillating behaviour, with latency that increases and decrease as time pass. This behaviour is the consequence of the fact that that not all the logs in a batch are published to the pipeline at the same instant, but instead, they come in small groups: so the first logs that are published have the time to be consumed by the Logstash instances and the queue becomes empty, then another group arrive, and this repeats periodically. When the number of active links increases, this small delay between the real publishing of each log makes the average latency converge. This is in some way representative of the real operational environment case: the logs never comes at the same instant as different links have small clocks deviation and different network latency. Figures from 5.5 to 5.9 shows the distribution of the Kafka latency for some key active links numbers: even if the average is convergent due to the non-instantaneous arrival of a new batch, from the distribution we can notice that the most populated interval is the between zero and 100 ms up to 250ms for a high amount of active sources, while a smaller amount of logs are processed by Kafka in periods of duration laying on a long tail above this values, especially as the number of links increases.

The following contribution in the pipeline is relative to the amount of time required by Logstash to process one log and store it inside Elasticsearch, called *Logstash latency*: this is a value that does not depends on the number of active links but only on the complexity of the preprocessing operations that are done on each log. For this setup, its average is around 3.5 ms, as the defined Logstash filters are quite simple.

The last component of the end-to-end latency is the *Spark latency*. This is, as expected, the most important contribution. The ML model

requires the logs in windows, and this requires aggregation of the data in groups corresponding to the sources and transmission direction: this is quite expensive from a computational point of view. As a consequence, it requires a huge amount of time to be completed. Figure 5.10 shows the average value of the Spark latency metric, which increases linearly as the number of active links increases. However, for use cases that do not require aggregation between different logs, this value is significantly smaller. Finally, figure 5.11 shows all the components contributing to the end-to-end latency in a bar plot. As explained before, Spark latency is predominant in the breakdown of the end-to-end latency.

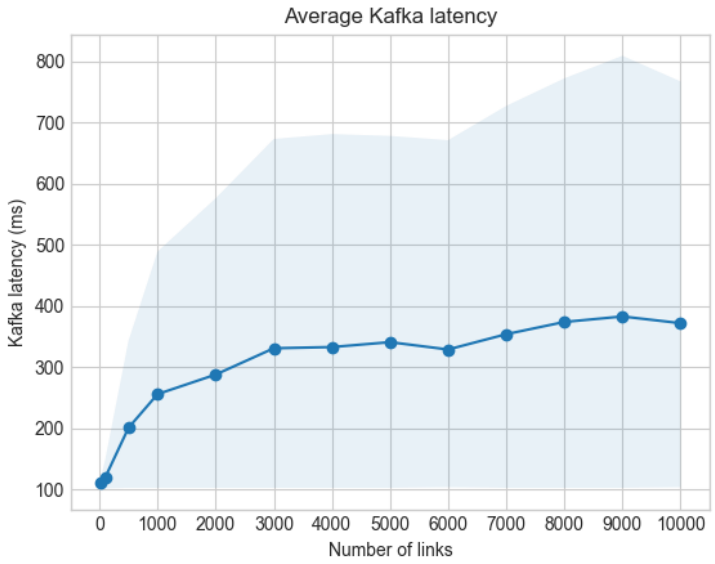

Figure 5.3: Average Kafka latency plot.

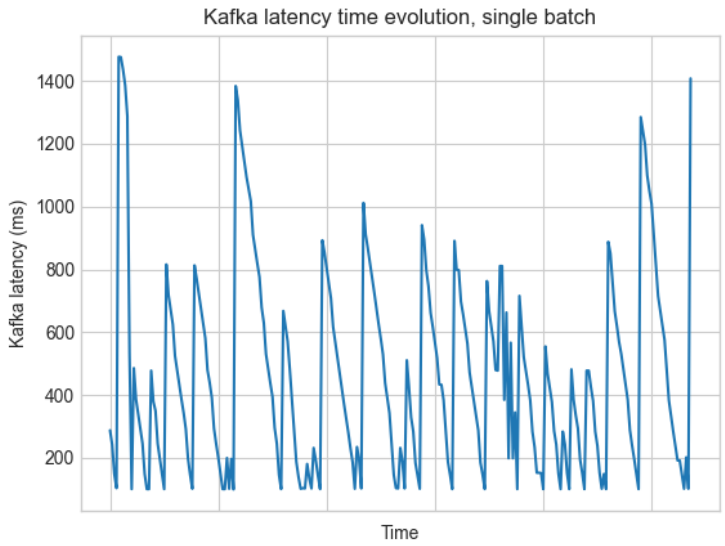

Figure 5.4: Average Kafka latency time evolution, single batch.

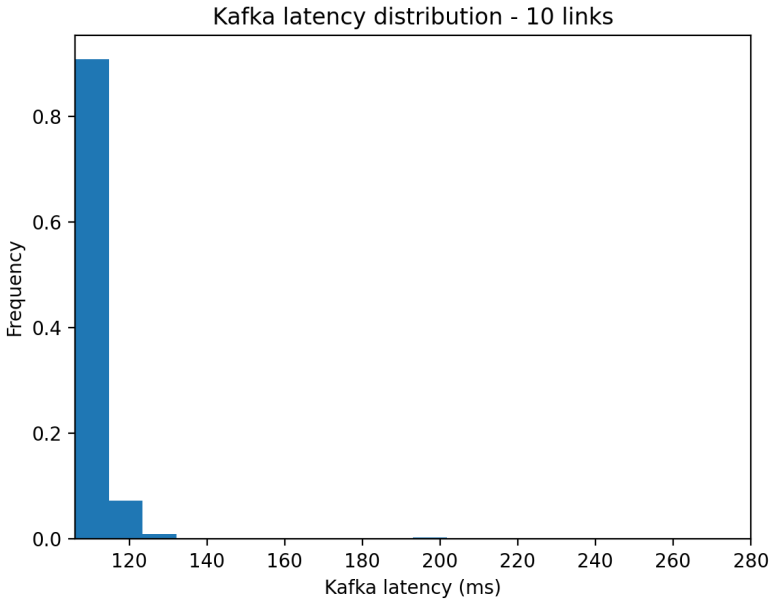

Figure 5.5: 10 Links Kafka latency distribution.

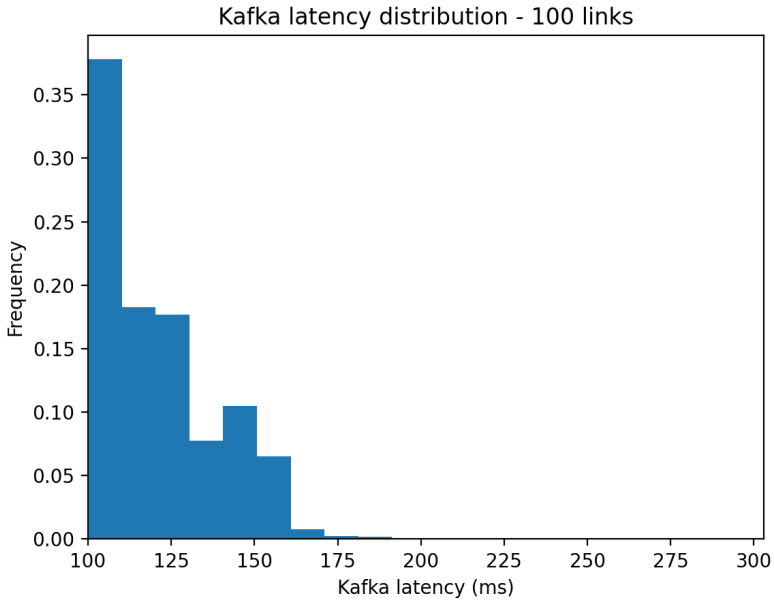

Figure 5.6: 100 Links Kafka latency distribution.

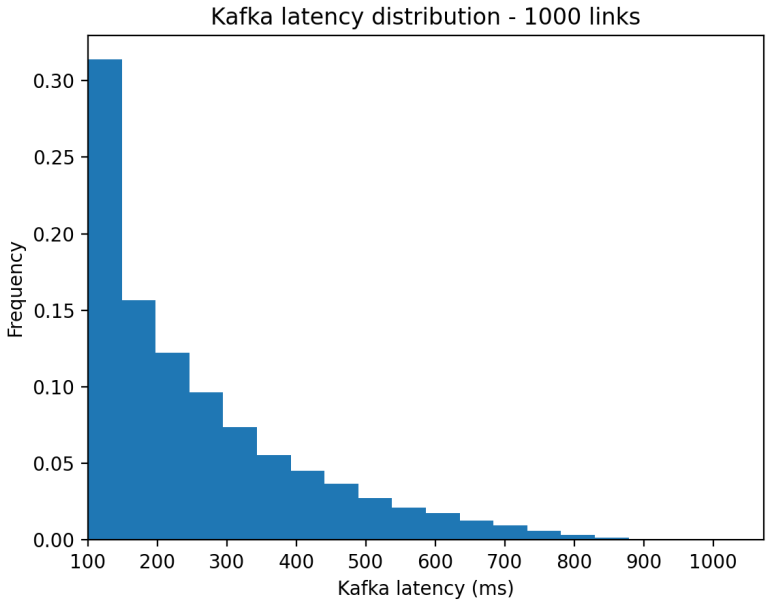

Figure 5.7: 1000 Links Kafka latency distribution.

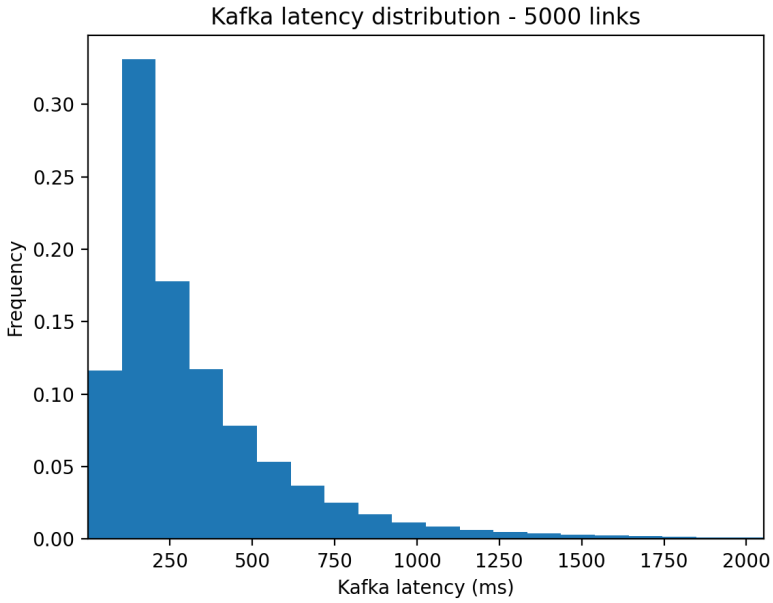

Figure 5.8: 5000 Links Kafka latency distribution.

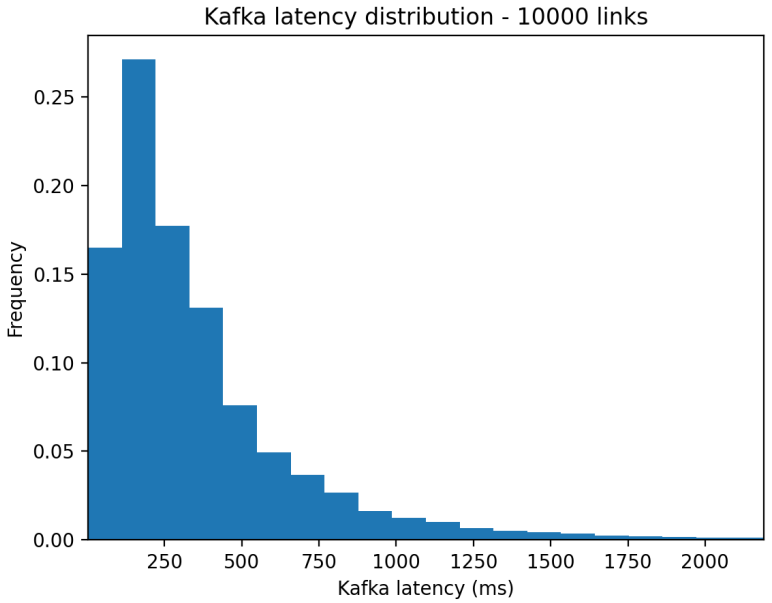

Figure 5.9: 10000 Links Kafka latency distribution.

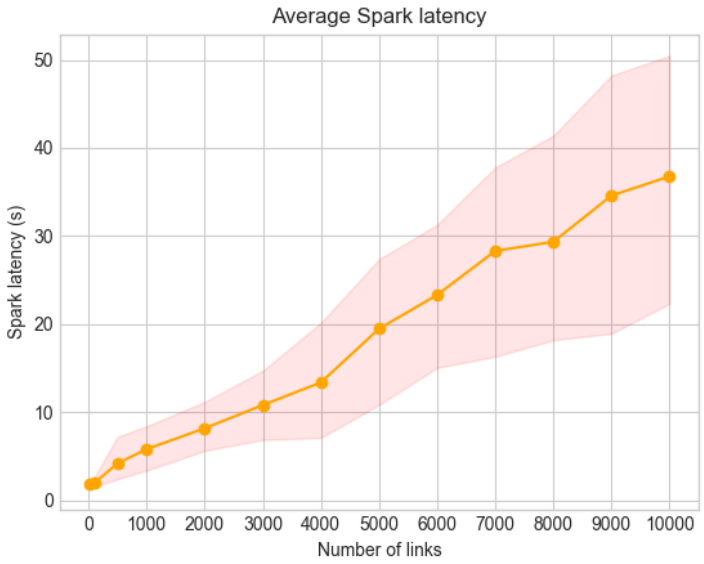

Figure 5.10: Average Spark latency plot.

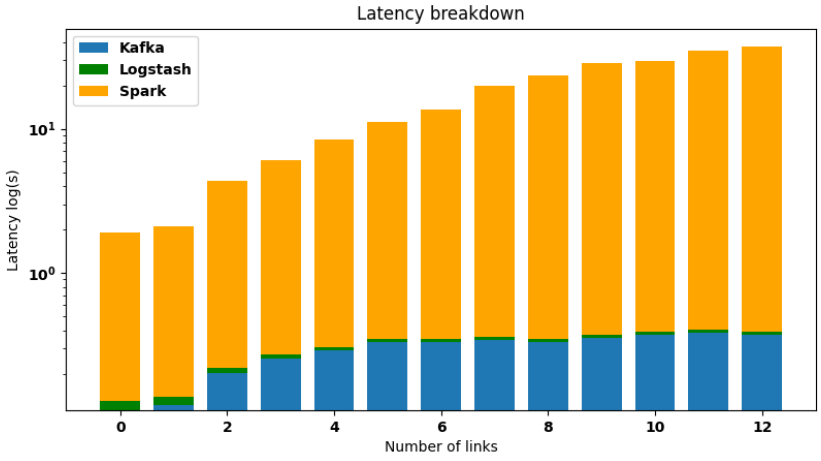

Figure 5.11: Breakdown bar plot. Latency value is in logarithmic scale.

Chapter 6

Conclusion and future work

In this thesis, we design a pipeline architecture to support the data ingestion and machine learning model application on streams of data. We first introduced the best practices and requirements for distributed software pipelines, and then we select the application to answer this issue, highlighting their most important features. We describe how these applications interact in the pipeline to solve the data ingestion part. Then, we explain how the general-purpose computational framework Apache Spark can be used for the application of machine learning models on streams of data. We also explain how our design allows visualization of both the original data and the ML model outcome with simple but powerful dashboards, allowing data exploration also by non-technical users. For every pipeline part, we proposed a design that is fault-tolerant and horizontally scalable, which allows even better performance with more powerful hardware or on multiple machines setup. We also demonstrated how the use of containerization can make management of complex distributed systems easy and how to monitor the functioning and the performance of the pipeline to allow a fast reaction to faults and bottlenecks that could arise during normal operations in an operational environment. Finally, we presented a reference implementation of the proposed architecture on the failure management on microwave network problem and the results on this use case: we were able to achieve near real-time performance on thousands of different independent data streams, with a maximum average latency between 30 and 40 seconds in a ten thousand streaming source test environment, which is thirty times lower than the time between two consecutive streaming events.

The latency breakdown shows that the most important contribution to the end to end latency is due to the machine learning preprocessing and ML model application, while the data ingestion phase contribution is negligible.

As future research direction we can identify the following ones:

- *Hyperparameter selection and ML model training on Spark cluster*: In our work, we describe how Apache Sparks can be used for the application of pre-trained ML models to data that requires complex aggregations and preprocessing. The Spark cluster huge processing power can be also used to train hundreds of ML models with different parameters in parallel to select the most accurate one. This process is called hyperparameters tuning or selection [6]. Scikit-Learn provides the GridSearchCV class [37], that automatically performs grid search over a parameter grid with cross-validation on the dataset and selects the optimal ones. By using *Joblib-Spark* library Scikit-Learn grid search can benefit the power of Spark to perform the hyperparameters tuning in parallel on multiple nodes with a dataset that could potentially not fit one machine memory [34].

- *Online Learning*: In our work, we considered the case of machine learning models that are pre-trained on a static labelled dataset and that does not change over time. If we can provide labels during the ingestion of the data bot through a human expert or other kinds of automatic analysis, we can also continuously train the deployed ML model with the newly available data. The ML model benefit as during the time evolution new patterns could arise from the data, as also data streams can evolve as time pass: for example, an ML model that predicts the products that a user can buy is subjected to change of seasonality or modes. Online learning is faster than the offline retraining of the ML model, as it does not require the full dataset to train the ML model.

- *Deployment on Kubernetes*: In our work, we considered multiple instances and replication for all the components on the pipeline, but only at a Docker abstraction level on a single host. To be fully fault-tolerant, also the Docker host must be replicated. Kubernetes [36] is an orchestration engine originally developed by

Google, that makes possible the clustering of multiple Docker hosts and the automatic deployment of containers on them, providing fault tolerance at the host level. It also supports automatic horizontal scaling of applications that have excessive loads as well as replacement of faults instances. This allows setups with thousands of containers on multiple machines and can be used for handling hundreds of thousands of data streams that are ingested inside the proposed pipeline.

Bibliography

[1] P. Atzeni et al. *Database Systems: concepts, languages and architectures*. McGraw-Hill, 1990. ISBN: 0077095006.

[2] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. "Virtualization vs Containerization to Support PaaS". In: *2014 IEEE International Conference on Cloud Engineering*. 2014, pp. 610–614. DOI: 10.1109/IC2E.2014.41.

[3] W.H. Inmon and Daniel L. *Data Architecture: A Primer for the Data Scientist*. Elsevier, 2015. ISBN: 9780128020449.

[4] Commscope. *Microwave communication basics*. 2017.

[5] Google Developers. *Data Preparation and Feature Engineering for Machine LEarning*. 2017.

[6] Aurlien Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 1st. O'Reilly Media, Inc., 2017. ISBN: 1491962291.

[7] S. Pranav and Sharath Kumar M. N. *Learning Elastic Stack 6.0*. OReilly, 2017. ISBN: 9781787281868.

[8] M. Van Steen and A. S. Tanenbaum. *Distributed Systems*. 2017. ISBN: 978-1543057386.

[9] Eberhard Wolff. *Microservices: Flexible software architecture*. Addison-Wesley Professional, 2017. ISBN: 0134602412.

[10] Gianpaolo Cugola. *Fault Tolerance in Distributed Systems*. 2019.

[11] T. Kolajo, O. Daramola, and A. Adebiyi. "A. Big data stream analysis: a systematic literature review." In: *Big Data* (2019). DOI: 10.1186/s40537-019-0210-7.

[12] Alessandro Margara. *Big Data Platforms*. 2019.

[13] Alessandro Margara. *Consistency and Replication in Distributed Systems*. 2019.

[14] Jacob R. Sutton et al. "PhysOnline: An Open Source Machine Learning Pipeline for Real-Time Analysis of Streaming Physiological Waveform". In: *IEEE Journal of Biomedical and Health Informatics* 23.1 (2019), pp. 59–65. DOI: 10.1109/JBHI.2018.2832610.

[15] Stefano Zanero and Marco Carminati. *Introduction to Computer Security*. 2019.

[16] J. Damji et al. *Learning Spark: Lightning-Fast Data Analytics*. OReilly, 2020. ISBN: 978-1492050049.

[17] Francesco Musumeci et al. "Supervised and Semi-Supervised Learning for Failure Identification in Microwave Networks". In: *IEEE Transactions on Network and Service Management* (2020). DOI: 10.1109/TNSM.2020.3039938.

[18] L. Velasco et al. "Intent-Based Networking for Optical Networks". In: *Journal of optical communications and networking* (2020). DOI: 10.1364/JOCN.99.099999.

[19] Armin Catovic et al. *Linnaeus: A highly reusable and adaptable ML based log classification pipeline*. 2021. arXiv: 2103.06927 [cs.LG].

[20] Andrea Sgambelluri et al. "Reliable and scalable Kafka-based framework for optical network telemetry". In: *IEEE/OSA Journal of Optical Communications and Networking* 13.10 (2021), E42–E52. DOI: 10.1364/JOCN.424639.

[21] L. Valcarenghi et al. "A Scalable Telemetry Framework for Zero Touch Optical Network Management". In: *2021 International Conference on Optical Network Design and Modeling (ONDM)*. 2021, pp. 1–6. DOI: 10.23919/ONDM51796.2021.9492488.

[22] *Apache Kafka Documentation*. URL: https://kafka.apache.org/documentation/.

[23] Amazon AWS. *Amazon EC2 Instance Types*. URL: https://www.amazonaws.cn/en/ec2/instance-types/.

[24] *Docker Documentation*. URL: https://docs.docker.com.

[25] Elastic. *Beats Platform Reference*. URL: https://www.elastic.co/guide/en/beats/libbeat/current/beats-reference.html.

[26] Elastic. *Elastic Stack and Product Documentation*. URL: https://www.elastic.co/guide/index.html.

[27] Elastic. *Elasticsearch for Apache Hadoop*. URL: https://www.elastic.co/guide/en/elasticsearch/hadoop/current/reference.html.

[28] Elastic. *Elasticsearch synchronization with a relational database*. URL: https://www.elastic.co/blog/how-to-keep-elasticsearch-synchronized-with-a-relational-database-using-logstash.

[29] Elastic. *Grok parser reference*. URL: https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html.

[30] Elastic. *Kibana Guide*. URL: https://www.elastic.co/guide/en/kibana/index.html.

[31] Elastic. *Logstash Reference*. URL: https://www.elastic.co/guide/en/logstash/master/index.html.

[32] Elastic. *Metricbeat Reference*. URL: https://www.elastic.co/guide/en/beats/metricbeat/7.x/index.html.

[33] Elastic. *Monitoring in a production environment*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/monitoring-production.html.

[34] Vihag Gupta and Pradeep Reddy. *Boosting Parallelism for ML in Python using scikit-learn, joblib and PySpark*. URL: https://www.qubole.com/tech-blog/boosting-parallelism-for-ml-in-python-using-scikit-learn-joblib-pyspark/.

[35] ITU-T. *G.828 : Error performance parameters and objectives for international, constant bit-rate synchronous digital paths*. URL: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-G.828-200003-I!!PDF-E&type=items.

[36] Kubernetes. *Kubernetes Reference*. URL: https://kubernetes.io/docs/reference/.

[37] Sci-Kit Learn. *Tuning the hyper-parameters of an estimator using grid search*. URL: https://scikit-learn.org/stable/modules/grid_search.html.

[38] OGC. *Publish/Subscribe Interface Standard*. URL: http://docs.opengeospatial.org/is/13-131r1/13-131r1.html.

[39] Spark. *Spark Documentation*. URL: https://spark.apache.org/docs/latest/.

[40] David Vrba. *Performance in Apache Spark: Benchmark 9 Different Techniques*. URL: https://towardsdatascience.com/performance-in-apache-spark-benchmark-9-different-techniques-955d3cc93266.