Experimental Big Data Processing Pipeline Design for 5G Mobile Networks

Muhammad Sulaiman
David R. Cheriton School of Computer Science, University of Waterloo, Ontario, Canada
m4sulaim@uwaterloo.ca

***Abstract*—The modern world generates vast amounts of data on a daily basis, making it increasingly difficult for single machine-based setups to effectively process and store it. To address this issue, many organizations are turning to clustered setups, which leverage distributed data storage and processing to handle large-scale data. Large-scale data manipulation imposes different types of requirements based on the type of analysis being performed. For instance, training machine learning models from historical data may require a different approach than visualizing streaming data in real-time. On the other hand, for some applications, the sensitive nature of data requires that it stays on-premise i.e., it cannot be stored in the cloud. The monitoring and management 5G network is one such application dealing with big data where strong guarantees about its safety are required. In this project, I describe the design, implementation, and testing of an experimental big-data pipeline for Rogers on the CN Cluster at Davis Centre (DC) at the University of Waterloo.**

***Index Terms*—5G, C-RAN, Network Slicing, Admission Control, Multi-agent Reinforcement Learning**

I. Introduction

Modern mobile networks serve a wide variety of applications such as low-latency remote control of autonomous vehicles and high throughput such as 4K video streaming. However, traditional mobile networks use a monolithic infrastructure to serve all these diverse applications, leading to inefficient resource utilization and a diminished quality of experience (QoE) for the users. To address this issue, Network Softwarization principles are being adopted, allowing for programmable networks. In 5G networks, the infrastructure provider can use network function virtualization and software-defined networking to create virtual networks called ”slices” that are isolated but hosted on a shared infrastructure. Intelligent management and orchestration (MANO) of these slices can improve resource utilization and QoE. Autonomous management and orchestration of 5G slices is a significant challenge in 5G networking. 5G networks can be complex and difficult to model and manage. In my research, I have surveyed the literature on autonomous 5G network management and identified six key challenges that must be addressed for the effective management of 5G slices. One of these challenges is that of big data processing.

Different algorithms for 5G slice management may have varying data requirements, such as processing online streaming data or learning from offline historical data, and a standardized and flexible way to interact with this data is needed. Additionally, mobile networks generate large volumes of data that can be in the terabytes, making it difficult to store and process using traditional methods. To address this challenge, I have developed a big data processing pipeline for 5G mobile networks. The pipeline consists of four components: data ingestion, data storage, data processing, and data visualization. It has been implemented using open-source technologies such as Kafka, Logstash, Elasticsearch, Hadoop HDFS, Spark, and Kibana. The pipeline is currently being hosted on the CN cluster in Davis Centre, and it is being used by my colleagues at the Hardware Lab to deal with the data generated by our in-lab 5G testbed. This pipeline allows for efficient and flexible processing of large amounts of data, which is crucial for the effective autonomous management of 5G network slices.

The remainder of this report is structured as follows: In Section II, I provide an overview of 5G networks and discuss the specific big data-related challenges that must be addressed when designing a big data processing pipeline for a 5G network. In Section III, I describe in detail the open-source tools that we used to address these challenges. In Section IV, I discuss the issues encountered during the practical implementation and highlight the key learnings from this process. In Section V, I validate the functioning of the different components of the pipeline. Finally, after addressing the critiques by fellow classmates in sections **??**, I conclude the report in section VI and discuss potential areas for future work.

II. Big Data Challenges in 5G Mobile Networks

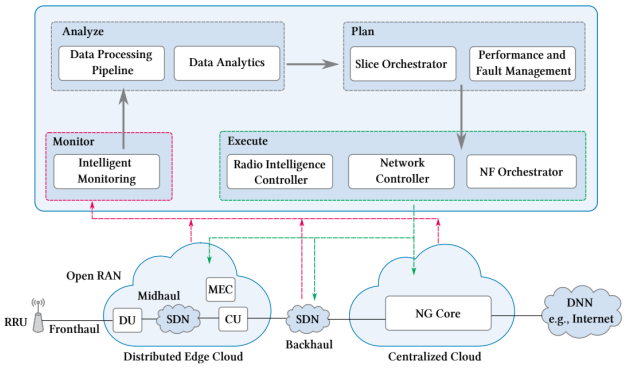

Fig. 1: System Control Loop [1]

As discussed in Section I, the virtualization of mobile networks presents opportunities for intelligent network management and orchestration [2, 3, 4]. In [1], the authors propose a closed-loop system that can be integrated with a 5G network for this purpose. The system design is based on the MAPE (monitor, analyze, plan, execute) control loop, as shown in Fig. 1.

The monitor module is responsible for collecting data from the substrate network and forwarding it to the analyze module, which hosts the data processing pipeline. This pipeline is responsible for ingesting the incoming data, storing and visualizing it, and facilitating data analysis. Based on the raw data received, the analyze module interacts with the data processing pipeline to calculate relevant key performance indicators (KPIs) and any additional metrics useful for visualization or performance management decisions. The analyze module then forwards the processed data to the plan module. The plan module hosts intelligent algorithms such as intelligent 5G slice orchestration and performance and fault management. Based on the data received from the analyze module, these algorithms produce output decisions, which are then forwarded to the execute module. The execute module is responsible for interacting with the network and applying these actions.

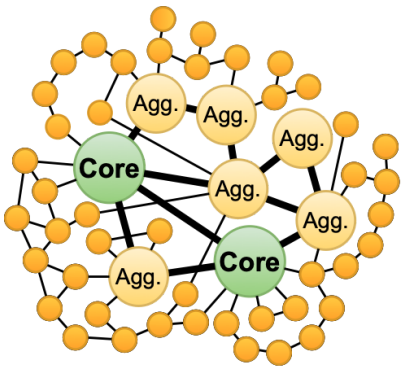

Fig. 2: 5G substrate network topology

In a 5G mobile network, incoming data from users must be processed by multiple network functions (NFs) within a Radio Access Network (RAN) and a core network (CN) before it can be routed to the wider internet. These NFs must be hosted on virtual servers, as they require compute resources to operate. These NFs can be hosted on decentralized servers located closer to users or on centralized servers, which can benefit from multiplexing gains that reduce compute resource demand.

In modern metropolitan networks, the substrate network consists of the aforementioned servers and follows a ring-and-spur topology, as shown in Fig. 2. In this topology, decentralized sites host NFs that require low latency due to proximity to users, while aggregation and core sites host NFs with less stringent latency requirements. This structure is replicated across the entire country, with centralized sites located in large cities and decentralized sites located in surrounding areas closer to users. Due to the complexities of networking, each site has a significant number of components that need to be monitored, which presents challenges for data ingestion and storage. Since each component at these sites generates data in real-time, a single link handling all the incoming data can quickly become a bottleneck. For example, if a single server is used to enqueue incoming data, the link to that server becomes the bottleneck. Additionally, as the number of data consumers increases, a single server may not be able to serve all of them in a short amount of time. Therefore, handling such cases requires a distributed data ingestion system with multiple servers to enqueue arriving data and prevent link bandwidth and server compute resource bottlenecks. Furthermore, incoming data is often unstructured and contains a significant amount of redundant information, so the data ingestion solution should be able to parse and clean the data.

Once the data has been ingested, it needs to be stored for later access. As the size of this data increases, traditional data storage become ineffective at storing this data. This is because traditional methods for data storage utilizing hard disk drives (HDDs) or solid-state drives (solid-state drives) can only provide limited data storage and retrieval speeds. If the size of the data is in terabytes, just storing and retrieving this data can take days. Additionally, for commercial applications, the data needs to be stored for a long period of time. however, the traditional storage methods have a high failure rate. This causes a high risk of data loss. Therefore, there is a need for a data storage mechanism that can not only provide high data storage and retrieval speeds but can is also fault tolerant. To address these challenges, there are open-source solutions available such as Hadoop distributed filesystem (HDFS). However, these solutions usually trade-off their data storage and retrieval throughput with their data access latency. In other words, even though they provide high input/output throughput, the latency to search for a specific record within the database can be quite high. This can be unsuitable for applications that require low-latency access to the stored data. Therefore, in addition to addressing the previously mentioned challenges, the data storage solution for 5G networks must also provide low-latency access to the relevant data.

Once the data has been ingested and stored, it needs to be processed. Traditional data processing methods operate on the data sequentially. This means that the computation time grows linearly with the size of the data. In this case, even simple operations on the data can take a significant amount of time. For example, calculating the distribution of a particular feature within the data requires iteration through the entire dataset. Therefore, there is a need for a data-processing solution that can leverage parallel processing in order to minimize the computation time. For instance, in the aforementioned example of calculating the distribution for a given feature, multiple workers can be instantiated in parallel and a divide-and-conquer strategy can be leveraged. Additionally, heterogeneous algorithms constituting intelligent management and orchestration of 5G networks can require multiple complex calculations. Therefore, any solution used

for distributed data processing must be able to implement any arbitrarily complex calculation that an algorithm may require.

To extract useful insights from the data analysis, it is important to visualize it. Creating custom code for reading the data, calculating relevant metrics in a given timescale, and then creating a dashboard to visualize them in a presentable fashion can be quite cumbersome and time-consuming. This produces a need for a tool that can be integrated with the data storage solution and can easily be used to create presentable dashboards and visualize the data.

Finally, the data gathered from mobile networks can be quite sensitive as it can contain information related to the end-users. Therefore, it is important that it is stored in a highly secure private cloud rather than in a public cloud. This constitutes the final big-data-related challenge in 5G networks: on-premise data storage.

In conclusion, the major challenges in the design of big-data processing pipelines for 5G mobile networking include:

- Distributed data ingestion
- Distributed, fault-tolerant data storage
- Distributed data processing
- Easy data visualization
- Data security

III. Data Processing Pipeline Design

Fig. 3 shows the software architecture of the proposed data processing pipeline. This section details the design of the different components of the data processing pipeline in order to address each of the challenges mentioned in section II.

A. Data Ingestion

There are several options for data ingestion frameworks, such as RabbitMQ, Apache Flume, and Apache Nifi. However, as illustrated in Fig. 3, our data processing pipeline utilizes Apache Kafka and Logstash for data ingestion.

Kafka is an ideal choice for our application due to its ability to handle high volumes of data with low latency and its strong durability guarantees. It is optimized for high throughput and low latency, making it suitable for real-time data ingestion scenarios. Additionally, Kafka has robust integration with a variety of stream processing frameworks, making it easy to incorporate into existing data pipelines.

At a high level, Kafka operates as a distributed, scalable message broker. It is commonly used in many data processing applications as a way to collect and distribute incoming data [5, 6]. Kafka allows producers to publish data to one or more topics, while consumers can read the published data from these topics. However, behind the scenes, Kafka stores the data across multiple nodes and supports data replication to ensure that data is not lost if a node goes down. This makes Kafka a powerful tool for building distributed, scalable, and fault-tolerant data pipelines and streaming applications.

As mentioned in II, the arriving data is often unstructured and contains a significant amount of redundant information. Integrating Logstash with Kafka allows us to effectively process and manage unstructured data that is often redundant. Logstash reads data from Kafka topics, parses it, and cleans it by removing redundancies and, enriches it with any missing information. This ensures that the data is structured and consistent before it is fed back into a new Kafka topic for further processing. While Logstash may not offer complex data manipulation capabilities, it is a powerful tool for managing and transforming real-time data streams in Kafka. Finally, we also utilize Logstash to move the data from Kafka queues to the appropriate storage location.

B. Data Storage

Big data storage is a critical component of designing a big data processing pipeline, as it determines how the data will be stored and accessed. Traditional data storage methods, such as relational databases, are not well-suited for storing very large amounts of data due to their limited scalability and performance. Therefore, specialized tools that are specifically designed for storing and processing large amounts of data are required.

One such tool is the Hadoop Distributed File System (HDFS), which is a big data storage solution that is part of the Hadoop ecosystem. HDFS is designed to store and manage large amounts of data in a distributed manner, allowing it to scale to very large data sets and support distributed processing. HDFS stores data in the form of blocks, which are typically 64 MB or 128 MB in size. Storing data in blocks allows HDFS to efficiently store large files, as it can divide the file into blocks and store the blocks on different nodes in the cluster. This also allows HDFS to improve the read throughput of the data, as different blocks of data can be read in parallel from different nodes in the cluster. Additionally, HDFS replicates each block across multiple nodes in the cluster to provide fault tolerance, ensuring that the data is always available and can be accessed even if a node fails. However, there are also some drawbacks to storing data in blocks. For example, small files that are smaller than a block size must be padded with unnecessary information to fill out a block, which can waste storage space. Additionally, altering or deleting a small part of a large file in HDFS typically requires reading the entire block, modifying the part, and writing the entire block back to the cluster, which can be time-consuming and resource-intensive. This is because the entire block containing the data must be altered and synchronized across the nodes in the cluster.

On the other hand, Elasticsearch is a big-data storage framework that is optimized for fast data access and supports storing and updating small files efficiently. Rather than storing data in blocks like HDFS, Elasticsearch stores data in the form of documents, which are self-contained units of data that represent a single record or entity. Multiple documents that belong together are stored in a collection called an index, which is similar to a table in a traditional database. Due

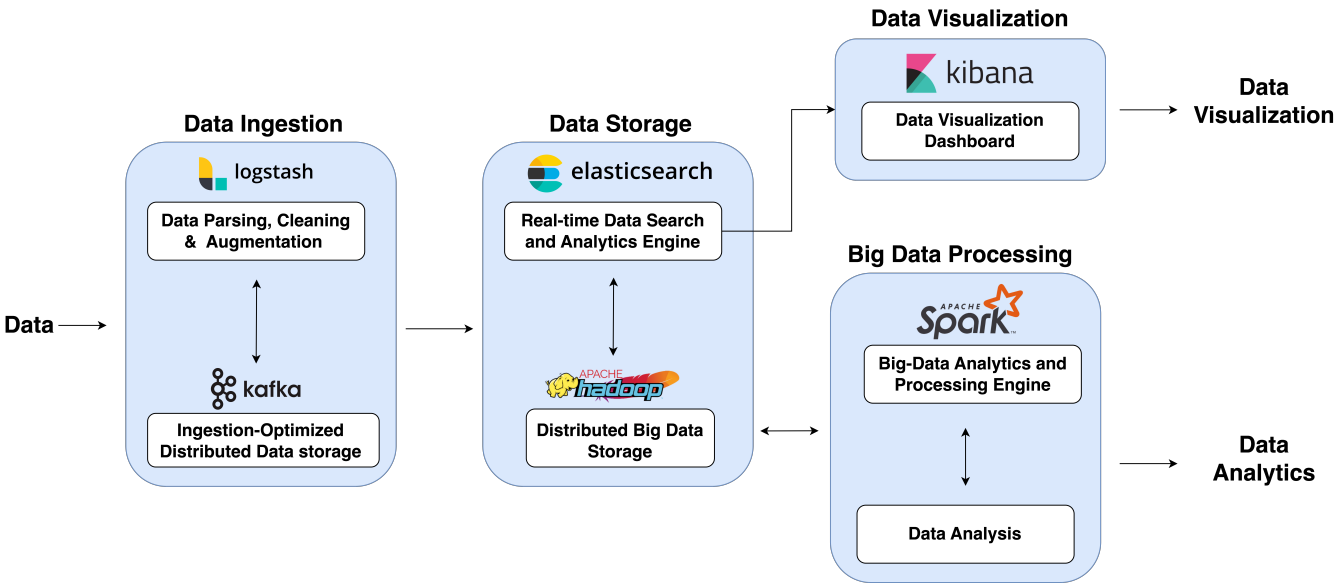

Fig. 3: Data Processing Pipeline Architecture

to this data storage abstraction, Elasticsearch can quickly retrieve and manipulate individual documents. It also makes several optimizations and uses data structures such as an inverted index to support fast search and retrieval of documents based on their contents. However, these optimizations and data structures come at the cost of lower storage efficiency compared to some other storage solutions. In our preliminary experiments, we observed that Elasticsearch may use as high as six times the storage compared to the actual size of the data for a time series.

As discussed previously, 5G networks can generate large amounts of data and therefore have higher data storage requirements than traditional big data applications. To meet these requirements, it is important to use a data storage solution that is highly efficient in terms of storage space usage. One such solution is the Hadoop Distributed File System (HDFS), which is designed to store and manage large amounts of data in a distributed manner and can scale to very large data sets. However, as mentioned earlier, HDFS has some drawbacks related to data access and storage latency. These drawbacks may not be suitable for some applications that require fast data search and retrieval. In such cases, a different big data storage tool may be needed. Elasticsearch is highly optimized for fast data search and retrieval and is tightly integrated with Logstash. As a result, it can be used as an alternative to HDFS for data storage in situations where fast data search and retrieval are required.

In our data processing pipeline, shown in Fig. 3, both Elasticsearch and HDFS are used for data storage. The choice of data storage framework is determined at Logstash, which routes the data to the appropriate framework based on custom logic based on the publisher or the contents of the data. This allows the data processing pipeline to store historical data efficiently, while also acting as a platform for heterogeneous algorithms that may have different data storage and processing requirements.

C. Data Processing

Once the data has been stored, it has to be processed. The MapReduce programming model is a powerful tool for processing large volumes of data in a parallel, distributed manner. It is particularly well-suited for big data processing tasks, where processing the data in a sequential manner can take a significant amount of time. The MapReduce model consists of two distinct stages: the Map stage and the Reduce stage. In the Map stage, the input data is divided into smaller chunks and processed in parallel by multiple workers, and in the Reduce stage, the intermediate results are merged to produce the final output. One of the key benefits of the MapReduce model is that it leverages the distributed storage of data to process the data in parallel across multiple nodes in the cluster. This makes it possible to scale the processing of large data sets by adding more workers to the cluster, which allows us to process very large data sets in a reasonable amount of time, even on relatively modest hardware.

For example, if we wanted to count the occurrence of a specific word in a large corpus of text, we could use the MapReduce model to process the data in parallel across multiple workers. In the Map stage, each worker would map each occurrence of the word to the value 1, and in the Reduce stage, these values would be summed up to get the final count.

Building on the Map Reduce model, Apache Spark proposes a new data abstraction called Resilient Distributed Datasets (RDDs). An RDD is a fault-tolerant collection of objects distributed across a cluster, which can be processed in parallel. Backed by the files present in the storage, an RDD

is created by performing operations on data. RDDs support lazy execution i.e., even though the computations on the objects can be defined in advance, processing only takes place when the results are required. As a result, when an action is called, Spark can analyze the entire computation graph and create an efficient execution plan. For example, multiple sequential *filter* operations can be executed in a single pass over the data. Additionally, Spark allows for data sharing by allowing frequently accessed data to be persistently stored in the memory so that it does not need to be read again from storage through slow I/O links. This can result in up to 100x speed up in interactive queries, and iterative algorithms [7]. The last significant factor differentiating Spark from other frameworks is its approach to fault tolerance. Traditional frameworks secure the processed data by saving a checkpoint in storage over slow I/O or network links. Spark, on the other hand, only keeps the computation (lineage) graph used to process the data and simply reruns it if the processed data is lost.

Since Spark builds on the Map Reduce model, it can emulate arbitrary computation on distributed data [7], but at the same time, it includes the tools that help it avoid inheriting the drawbacks traditionally associated with Map Reduce. Spark provides a set of high-level libraries which save users from the need to perform low-level programming while preserving Spark's state-of-the-art performance. The four main libraries that come bundled with the base Apache Spark package are *Spark SQL, Spark Streaming, GraphX* and *MLlib*. *Spark SQL* allows users to manipulate RDDs abstracted as dataframes, which are well-known to Python and SQL programmers. *Spark Streaming* can be used to implement incremental stream processing using a "discretized streams" model. *GraphX* allows for easy manipulation and storage of graph-based data. Finally, as apparent by its name, MLlib provides a distributed implementation of the most common machine learning algorithms and can be utilized to easily implement custom ones. Since RDDs undergird all the functionality implemented in these libraries, these can be combined for even higher gains in performance in complex tasks. This makes Spark one of the most utilized frameworks for batch and stream processing, interactive queries, and scientific applications dealing with big data.

Due to the aforementioned reasons, as shown in Fig. Fig. 3, we integrate Spark with the HDFS in our data processing pipeline.

D. Data Visualization

Data visualization is a powerful tool that can aid in the development of clever algorithms, provides useful insights, and enables real-time monitoring of the network. One tool that is commonly used for big-data visualization is Kibana, which is often used in conjunction with Elasticsearch. Kibana offers an easy-to-use interface for creating dashboards and various types of data visualization graphs, even for those with little experience in data visualization. Since Kibana provides robust integration with Elasticsearch, it presents a nature to be used as our data visualization solution.

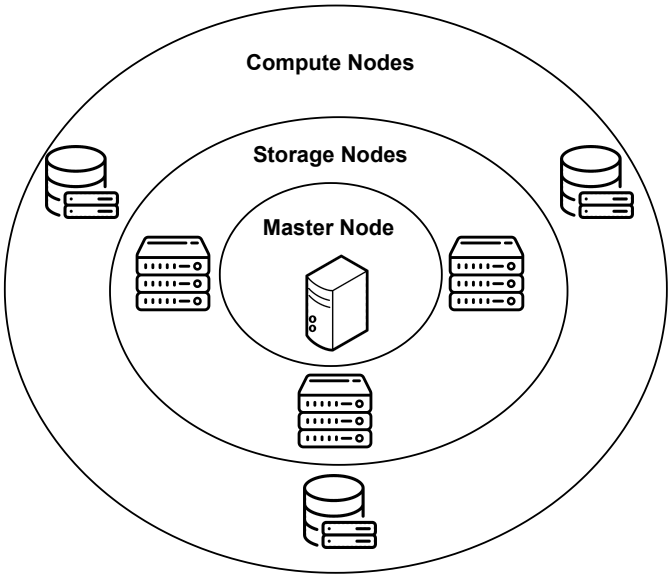

Fig. 4: Data Processing Pipeline Hardware Architecture

E. Hardware Stack

One of the challenges outlined in section I is the issue of data security in mobile networks. Mobile networks often contain sensitive data, which means that not only is it important to secure access to this data, but it is also important to consider where the data is stored. In certain regions, there are laws in place that require individuals' data to be stored within the same geographical location, such as the General Data Protection Regulation (GDPR) in Europe, which requires that users' data must be stored within the European Union. As a result, it is necessary to implement the data processing pipeline on a local cluster in order to ensure compliance with such laws and ensure the security of the data.

For this purpose, we obtained a set of five nodes from the Computer Science Computing Facility (CSCF) at the Davis Centre at the University of Waterloo, which we refer to as Node 1 through Node 5. Each of these nodes is equipped with 16GB of RAM, 8x Intel Xeon 3.30GHz cores, and nearly 1TB of storage, and runs Ubuntu 16.04. In a typical setup, there are usually multiple nodes that serve as compute nodes, storage nodes, and ingestion nodes, with some nodes being shared for different purposes. Additionally, there are typically at least two master nodes, one serving as the primary and the other as a backup. This topology is illustrated in Figure 4. However, due to the limited number of nodes available, we to modified our setup slightly to make the best use of the resources. We used two nodes for storage, three nodes for compute, one node for ingestion, and one node as the master node, with some nodes serving multiple purposes. The software stack for each node is listed in Table I.

Node#	Applications
1	Hadoop Master Kibana Kafka Elasticsearch Master Spark Master
2	Elasticsearch Worker Logstash
3	Elasticsearch Worker HDFS Worker Spark Worker
4	HDFS Worker Spark Worker
5	Spark Worker

TABLE I: CN Cluster Nodes and Corresponding Applications

IV. Issues and Key Learnings

During the implementation of the data processing pipeline, we encountered several issues which proved non-trivial to be addressed. Some of the significant ones are detailed below.

A. Framework Integration

Integrating different frameworks can be a complex and challenging process, particularly when it comes to achieving compatibility between the various tools and libraries involved. One issue that we have encountered in our work is the difficulties involved in integrating PySpark with Apache Spark, as well as Spark's integration with Elasticsearch. These integration challenges can be time-consuming and require a significant amount of troubleshooting to resolve, as errors that may arise during the integration process can be difficult to debug.

One lesson we have learned from this experience is the importance of carefully considering the specific version requirements of the various frameworks and tools we use. For instance, installing a specific version of Java or Scala may restrict the version of Spark that can be used, which may only be compatible with certain versions of HDFS. By taking the time to carefully consider these version requirements in advance, we can better ensure a smooth and successful integration process.

B. Debugging

Debugging an end-to-end data processing pipeline can be challenging due to the complexity of the various frameworks involved. For instance, issues that arise when running a Spark operation on distributed data stored in HDFS can have multiple potential causes, such as networking issues between the nodes, problems with Spark's configuration, or difficulties with integration with HDFS. While there are resources available online to help troubleshoot these issues, we have found that they can be heavily dependent on the specific versions of the frameworks being used. This can make debugging more difficult, especially if the frameworks are not commonly used together and are not tightly integrated.

To mitigate these challenges, it is important to consider framework integration when designing the pipeline. By choosing frameworks that are well-integrated and commonly used together, it can be easier to find debugging resources and support. This can save time and effort which can be wasted when trying to piece together frameworks that are not well-suited to work together.

C. Automation

Setting up a distributed data processing pipeline involves installing and configuring various frameworks on multiple nodes, which can be a time-consuming and tedious process. One way to streamline this process is to use Docker containers to containerize the different frameworks, and use technologies such as Kubernetes to orchestrate those containers across multiple clusters. However, this approach introduces an additional layer of complexity when initially setting up the pipeline, which can increase the time needed for debugging if issues arise with containerization or container orchestration.

To address this challenge, we took a different approach. Instead of manually installing and configuring the frameworks on each node, we developed scripts that can be used to automatically carry out these tasks. This allowed us to easily deploy the frameworks on new nodes by simply running the scripts, rather than manually repeating the setup process each time. This solution not only saved time but also helped to ensure consistency across the different nodes in the cluster. By automating the deployment process, we were able to streamline the setup of our distributed data processing pipeline and reduce the time needed for debugging.

V. Validation

For this project, our goal was to design and implement a data processing pipeline for 5G networks, which included the deployment of the pipeline in a local cluster. To demonstrate the successful implementation of the pipeline, we used a variety of methods to showcase the functioning of the different components. These include printing out logs, observing dashboards, and successfully making visualizations. These methods allowed us to validate that the pipeline was working as intended and provided a clear understanding of the various stages of data processing and the tools and technologies being utilized. The following subsections show the results of the aforementioned validations.

A. In-lab 5G testbed

To validate the big data processing pipeline, we set up a 5G testbed at the Hardware Lab in the Davis Centre. While the details of this testbed are beyond the scope of this report, it provides a valuable platform for testing and validating the data processing pipeline.

To carry out the subsequent validation tests described in this report, we installed Metricbeat on each of the servers hosting the 5G testbed. This software publishes the server's metrics to the appropriate Kafka queue at specific intervals, providing a steady stream of data that we can use to test and validate the data processing pipeline. Unless otherwise noted, we will be using this data for the validation tests described in the following sections.

```
pipeline@cn121:~$ /opt/kafka_2.13-2.8.0/bin/kafka-topics.sh --list --bootstrap-server master-node:9092
__consumer_offsets
metricbeat
temp
```

Fig. 5: Kafka Topics

Fig. 6: Logstash logs

B. Data Ingestion

As described in section III, we use Kafka and Logstash for the data ingestion stage of our pipeline. The data is being published by Metricbeat, so we create a Kafka topic called ”metricbeat”. This is illustrated in Fig. 5. Subsequently, we configure the Metricbeat client to publish the data to this topic at 1-second intervals. Once the raw, unstructured data is received at the ”metricbeat” topic, Logstash is able to parse and clean it. The corresponding logs are shown in Fig. 6. Afterward, Logstash routes the cleaned data to both Elasticsearch and HDFS for storage and further processing.

C. Data Storage

We validate HDFS by copying 1TB of data to the distributed storage and verifying the metrics shown in the dashboard. Fig. 7 shows the dashboard for HDFS in our data processing pipeline. We can see that it is using two nodes for distributed data storage, with each node having a storage capacity of nearly 1TB. However, there is additional data stored on these nodes, resulting in a total configured storage of 1.76TB. Additionally, we can see that our data occupies half of the storage space. It is also worth noting that even though the underlying storage is distributed across two nodes, HDFS provides an abstraction of a single storage device, making it easy to interact with the storage using simple commands such as ”ls”, ”mv” or ”cp”. This also simplifies the process of routing data from Kafka to HDFS using Logstash, as we only need to provide the HDFS master node address and HDFS handles the distributed storage automatically. This allows us to take advantage of the scalability and reliability of distributed storage without the added complexity of managing it ourselves.

Since Elasticsearch is tightly integrated with Kibana, we can utilize Kibana to monitor the Elasticsearch database. As previously mentioned, Elasticsearch stores related documents together as indexes, and for our pipeline, we have configured each day’s data as a separate index. This means that we can view the documents collected using Metricbeat on different days as different indexes in Elasticsearch. This can be observed in the dashboard shown in Fig. 8. This enables us to easily track and monitor the data being stored in Elasticsearch, as well as identify any potential issues or abnormalities in the data. Additional validation for the successful implementation of Elasticsearch can be obtained by exploring the stored data and viewing the features present in the documents being published by Metricbeat. These are shown in Fig. 9. We can see that the features include ”os”, ”kernel” among others, which are as expected.

D. Data Processing

For data processing, we use Spark in our pipeline. In order to validate the capabilities of Spark, we focused on two main

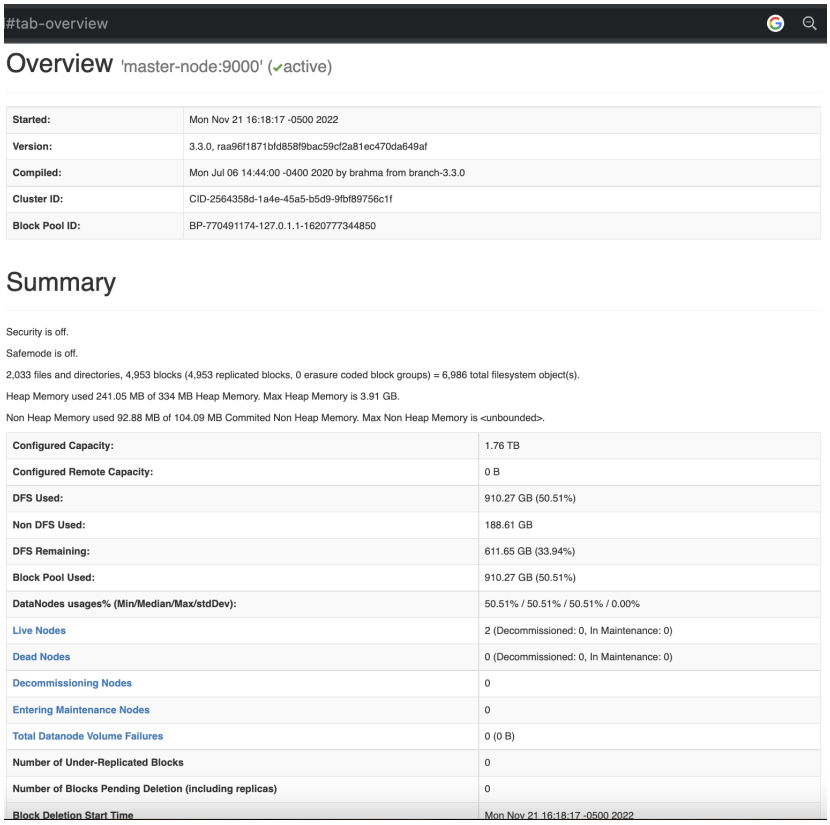

Fig. 7: HDFS dashboard

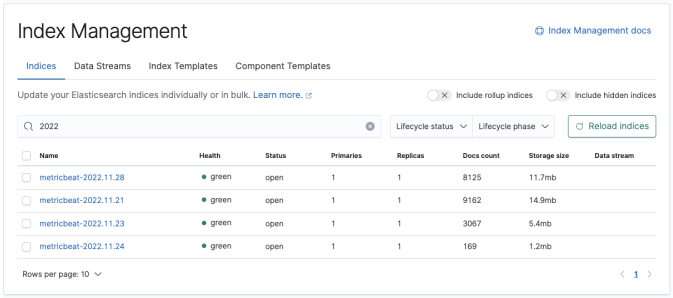

Fig. 8: Elasticsearch indexes

cases: the complexity of operations that can be performed using Spark, and the performance improvements that can be achieved by using Spark compared to traditional methods.

To demonstrate the first case, we set out to calculate the rolling average over the stored data, which involves reading the files, organizing the rows based on ingestion time, and then calculating the moving average. Fig. 10 shows the MapReduce-based code used to accomplish this task, as well as the results obtained. By manually calculating the moving average over the values shown, we can confirm that the calculations are correct and that Spark is capable of handling complex operations such as moving (rolling) averages.

To evaluate the performance improvements that can be achieved using Spark, we compared the time required to count the rows in a file, stored in HDFS, with 3M rows when using Spark versus a traditional Python program. Figure Fig. 11 shows the code used for both methods and the results obtained. We can see that Spark was able to complete the task in 107 seconds, while the Python program was still running after 44 minutes and had only counted 1/7th of the rows. These tests validate the proper integration of Spark with HDFS.

E. Data Visualization

The final component of the data processing pipeline that needs to be validated is data visualization. To facilitate this, we create a dashboard in Kibana to visualize the average of the system load metric being published by the Metricbeat client. This is illustrated in Figure Fig. 12. By visually inspecting the dashboard, we are able to confirm that the data is being visualized in real time.

VI. Conclusion and Future Work

In conclusion, modern 5G networks generate vast amounts of data, and efficient management of these networks requires the ability to handle and process this data effectively. To address this need, we have designed and implemented a big data processing pipeline for 5G mobile networks. This pipeline consists of four main components: data ingestion, data storage, data processing, and data visualization. These components are made up of various technologies, including Kafka, Logstash, Elasticsearch, HDFS, Kibana, and Spark, which work together

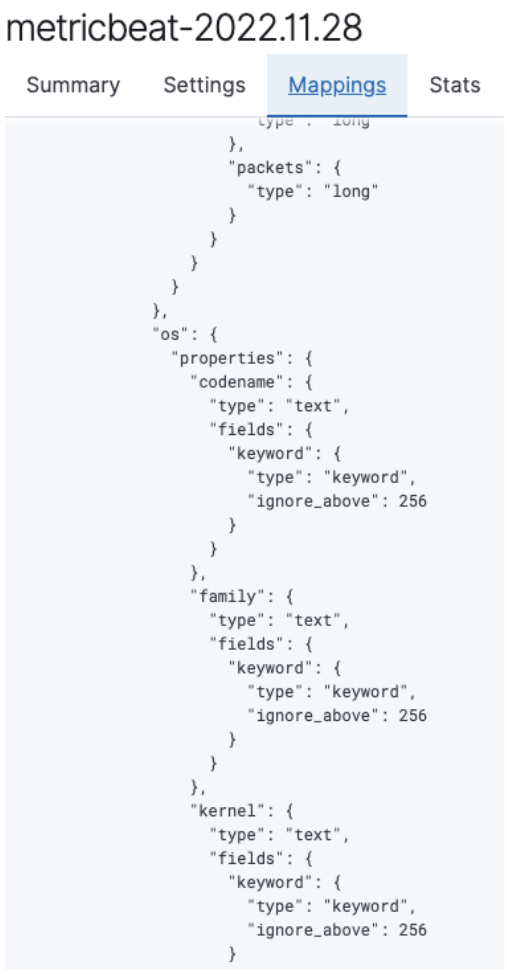

Fig. 9: Metricbeat index mappings

to collect, store, process, and visualize data from 5G networks. We have validated the successful operation of the pipeline through the use of various validation tests, demonstrating its effectiveness in handling and analyzing data from 5G networks.

The design and practical implementation of a big data processing pipeline is a complex and time-consuming task, and our work on this project has been a significant undertaking. However, we have now reached a stage where we have validated the successful implementation of the pipeline, and we can move forward to design more rigorous analytical tests to evaluate its performance. This will allow us to gain a deeper understanding of the capabilities and limitations of the pipeline and identify any areas for improvement. Another important improvement that is needed is the containerization of the pipeline, which will enable us to manage and scale it more easily.

ACKNOWLEDGEMENT

Although this report has been written by me in its entirety, the design and implementation of the data processing pipeline have been a combined effort by me and my colleague Arash Moayyedi, working at the Hardware Lab. My contribution to the pipeline was the deployment of Kafka, HDFS, and Spark, and their proper integration with the rest of the pipeline. In addition to Spark's integration, I also learned the MapReduce programming model and used PySpark to implement the validation tests related to Spark.

```
small_df = small_df.withColumn('time', F.to_timestamp(small_df['TimestampDateTime'], 'yyyy-MM-dd HH:mm:ss.SSS'))
small_df = small_df.na.drop(subset=['time'])

#Creating Rolling Window spec
#create window by casting timestamp to long (number of seconds)
#30*60 is the number of seconds in 1 hour
windowSpec = (Window()
     .orderBy(F.col("time").cast('long'))
     .rangeBetween(-1*30*60, 0))
small_df = small_df.withColumn('rolling_average', F.avg("feature1").over(windowSpec))

print(small_df.count())
small_df.sort("time").show()
```

```
10000
+--------+--------------------+--------------------+------------------+
|feature1|   TimestampDateTime|                time|   rolling_average|
+--------+--------------------+--------------------+------------------+
|  255822|2021-05-09 00:00:...|2021-05-09 00:00:...|          255822.0|
|  144374|2021-05-09 00:00:...|2021-05-09 00:00:...|          200098.0|
| 8580402|2021-05-09 00:00:...|2021-05-09 00:00:...|2993532.6666666665|
|19469328|2021-05-09 00:01:...|2021-05-09 00:01:...|         7112481.5|
|  171902|2021-05-09 00:01:...|2021-05-09 00:01:...|         5724365.6|
|   31940|2021-05-09 00:01:...|2021-05-09 00:01:...|         4775628.0|
|  555016|2021-05-09 00:02:...|2021-05-09 00:02:...|4172683.4285714286|
|    7910|2021-05-09 00:03:...|2021-05-09 00:03:...|         3652086.75|
|  491077|2021-05-09 00:04:...|2021-05-09 00:04:...|3300863.4444444445|
|   24779|2021-05-09 00:05:...|2021-05-09 00:05:...|         2973255.0|
| 1431763|2021-05-09 00:13:...|2021-05-09 00:13:...|2833119.3636363638|
| 2848311|2021-05-09 00:13:...|2021-05-09 00:13:...|2834385.3333333335|
|   86726|2021-05-09 00:14:...|2021-05-09 00:14:...| 2623026.923076923|
|   27018|2021-05-09 00:14:...|2021-05-09 00:14:...| 2437597.7142857143|
|   24324|2021-05-09 00:15:...|2021-05-09 00:15:...|         2276712.8|
|   13927|2021-05-09 00:15:...|2021-05-09 00:15:...|       2135288.6875|
|   28716|2021-05-09 00:16:...|2021-05-09 00:16:...|2011372.6470588236|
|  403256|2021-05-09 00:17:...|2021-05-09 00:17:...|1922032.8333333333|
|   74680|2021-05-09 00:23:...|2021-05-09 00:23:...|1824803.7368421052|
|  485752|2021-05-09 00:24:...|2021-05-09 00:24:...|         1757851.15|
+--------+--------------------+--------------------+------------------+
only showing top 20 rows
```

Fig. 10: Spark Rolling Average Code and Results

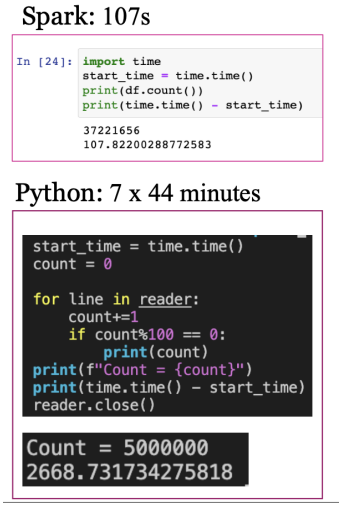

Fig. 11: Spark vs. Python row counting time

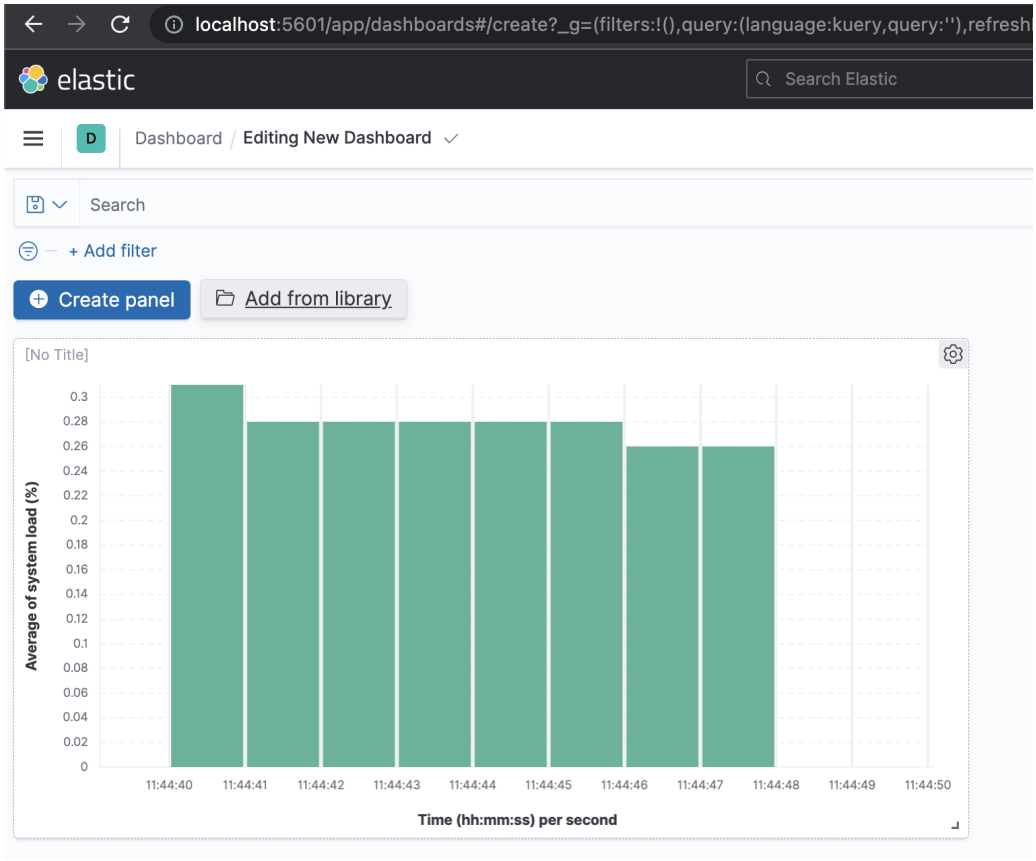

Fig. 12: Kibana Data Visualization

References

[1] R. Boutaba, N. Shahriar, M. A. Salahuddin, S. R. Chowdhury, N. Saha, and A. James, "AI-driven closed-loop automation in 5G and beyond mobile networks," in *ACM Workshop on Flexible Networks Artificial Intelligence Supported Network Flexibility and Agility (FlexNets)*, 2021, p. 1–6.

[2] N. Saha, A. James, N. Shahriar, R. Boutaba, and A. Saleh, "Demonstrating Network Slice KPI Monitoring in a 5G Testbed," in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, 2022, pp. 1–3.

[3] M. Sulaiman, A. Moayyedi, M. A. Salahuddin, R. Boutaba, and A. Saleh, "Multi-Agent Deep Reinforcement Learning for Slicing and Admission Control in 5G C-RAN," in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, 2022, pp. 1–9.

[4] M. Sulaiman, A. Moayyedi, M. Ahmadi, M. A. Salahuddin, R. Boutaba, and A. Saleh, "Coordinated Slicing and Admission Control using Multi-Agent Deep Reinforcement Learning," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2022.

[5] G. van Dongen and D. Van den Poel, "Evaluation of stream processing frameworks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1845–1858, 2020.

[6] F. Carcillo, A. Dal Pozzolo, Y.-A. Le Borgne, O. Caelen, Y. Mazzer, and G. Bontempi, "Scarff: A scalable framework for streaming credit card fraud detection with spark," *Information Fusion*, vol. 41, pp. 182–194, 2018.

[7] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A Unified Engine for Big Data Processing," *Commun. ACM*, vol. 59, no. 11, p. 56–65, oct 2016. [Online]. Available: https://doi.org/10.1145/2934664