What to Do when Privacy Issues Screw It Up: Ingestion Refactoring in a Big-Data Pipeline

Gian Paolo Jesi, Nicola Spazzoli, Andrea Odorizzi and Gianluca Mazzini
Lepida ScpA, Via della Liberazione 15, 40128 Bologna, Italy

Keywords: Big-Data, Ingestion.

Abstract: Privacy is an increasingly important concern especially in the European Union (EU). With the growing use of technology individuals' personal information is being collected and processed by companies and organizations on a massive scale. In order to be compliant with Privacy regulations and General Data Protection Regulation (GDPR) in particular, we could no longer use a software tool, LOGGIT, for several reasons. This tool was a cornerstone in one of our Big-data pipeline ingestion. We did our best to comply with this requirement as soon as possible. In this work, we discuss how we refactor our pipeline architecture several times in order to find a balance between our requirements in terms of reliability and the regulations of the GDPR.

1 INTRODUCTION

This paper tackles the problem of refactoring a BIG DATA pipeline in or Company. In particular, the ingestion step is the part affected by the refactoring. A sequence of exogenous events triggered the decision to abandon a specific software tool - which is called "LOGGIT" - and the need to find a suitable substitute adopting in-house knowledge and open source tools. The contribution we aim to provide is a real world use case which might be useful for other companies when dealing with similar issues.

Our Company, Lepida ScpA(lep, 2022), a subsidiary of the Emilia-Romagna Region in Italy and it is the main operational instrument regarding the implementation of the Regional ICT Plan. It provides a set of specialized services aimed to local Public Administrations (PA) and citizens producing a huge amount of unbounded heterogeneous data, such as: (public) WiFi access locations, Regional healthcare and IoT environmental monitoring data.

This flux of information is constantly growing and has the potential to create many opportunities not just for monitoring and managing each single sub-domain, but also for the creation of new business models involving public and private organizations and citizens.

Understanding the strategic importance of being able to exploit these data, in 2018 Lepida ScpA started with the creation of its first implementation of a BIG DATA infrastructure in order to continue its tradition of being the technological reference for the PA in the Emilia-Romagna Region.

Recently, Lepida ScpA joined the Regional Big data Platform "Big Data HPC" of the Emilia-Romagna Region. This platform project is founded with 3,5 M€ by the Regional "Piano Sviluppo e Coesione" (PSC)[1]. This platform is going to be ready at the beginning of 2024.

In this work, we focus on the specific use case of WiFi Regional data. In fact, in recent years, Lepida ScpA installed thousands of WiFi Access Points (AP) in public places (e.g., train stations, plazas, libraries, schools, and PA offices), conforming to the guidelines of the European Digital Agenda. This Regional WiFi (ssid: *EmiliaRomagna-Wifi*) service is 100% free and open to the public without any need of authentication.

One of our BIG DATA pipelines reads the data flux coming from our DHCP server logs in order to track the locations and connection time of the devices connecting to the Regional network. This particular pipeline exploited a software product, LOGGIT, developed by a local external Company as its main component. As anticipated at the beginning, due to a series of events that go beyond the scope of this document, our management decided to do not renew the contract with this Company and stop using the software. This issue forced us to find a new in-house solution in a short amount of time. In this work, we discuss the new requirements we need for the ingestion, the engineering choices we made and their evolution over time.

[1] In English: "Plan for Development and Cohesion".

Jesi, G., Spazzoli, N., Odorizzi, A. and Mazzini, G.
What to Do when Privacy Issues Screw It Up: Ingestion Refactoring in a Big-Data Pipeline.
DOI: 10.5220/0012554800003690

In *Proceedings of the 26th International Conference on Enterprise Information Systems (ICEIS 2024) - Volume 1*, pages 209-215
ISBN: 978-989-758-692-7; ISSN: 2184-4992

The remainder of this paper is organized as follows. We first discuss our problem in Section 2. Section 3 discusses the choices taken for our architecture and its implementation, Finally, 5 concludes the paper.

2 THE PROBLEM

The pipeline we are considering is designed to ingest data coming from our Company's DHCP server logs. By the way, the DHCP servers are actually two working in parallel. By the analysis of these logs we can extract each distinct device which connects to the Regional WIFI network, the time of the event and its location. In fact, since each WIFI AP deployed in a zone is associated with a specific sub-network IP, we can guess each device location by checking in which subnetwork its IP is laying. Starting from this basic information, many other can be inferred, such as tracking a device movement over the Regional territory[2].

We consider a device MAC address as its unique identifier. Unfortunately, especially in newer devices, a MAC address can change over time when a device connects for the first time to a new or unknown wireless SSID network. When this happens, the same device will be considered multiple times, but we have no practical options to avoid this behavior. However, we consider the probability of this event quite low.

Figure 1 shows the web user interface (UI) of LOGGIT, the actual component that we have to substitute. The following is a summary of the operations carried out by this software:

- It receives log messages from our Networking Department, where DHCP servers are hosted, through a listening Syslog service
- (pseudo) anonymize any MAC address in messages
- parse each log message and creates a JSON object populated with standard DHCP log parameters
- exploits an ElasticSearch (Mitra and Sy, 2016; Zamfir et al., 2019) instance to store the parsed messages in a specific index created with a retention of 365 days. This index also fuels the UI shown in Figure 1
- sends parsed messages to the next step of the pipeline which is an Apache Kafka (Hiraman et al., 2018) instance

[2]According to the GDPR we never track a single device/MAC, but an aggregation of at least three devices.

The decision to abandon this tool and go for our internal solution, has nothing to deal with the reliability and effectiveness of the previously adopted tool (i.e., LOGGIT), but rather with privacy issues concerning sensitive data flowing to a system which is open to an external Company.

In fact, DHCP log strings contain device mac addresses which are considered *personal data* and hence *sensitive*. EU rules are pretty strict (gdp, 2023) when dealing with privacy and Companies not compliant to the rules are running the risk of a substantial fine or even worst.

In addition, other technical choices have defined, such as using a distinct hashing algorithm for MACs.

Therefore, we are forced to engineer an in-house solution to implement all the features LOGGIT was providing. We have to say that the availability of a (web) UI has been a tremendous bonus from LOGGIT. The possibility to literally "see" what happens in almost real time and the option to make simple queries over the data by filtering parameters gave us the chance to spot problems before it was too late. Re-implementing this particular feature turned out to be the hardest part and, while we pave the road to solve it (see Section 5), it is still an open question. For this reason, we are going to discuss it later in Section 5.

3 ARCHITECTURE

Figure 2 shows the previous architecture. It is clear that LOGGIT implements half of the entire pipeline. The data stream, after reaching the Kafka queue, is then moved to a relational (Postgres) database by a Logstash instance. It is from the database that we extract knowledge via SQL processing.

Actually, when moving to the new Regional BIG DATA infrastructure we stated in Section 1, the relational database will be substituted by a Spark process writing to a Delta Lake table over HDFS stable storage. In this manner, we can keep the same SQL processing interface using tools such as Sparl-SQL (Armbrust et al., 2015; Shaikh et al., 2019), Hive (Camacho-Rodríguez et al., 2019) or Trino (Fuller et al., 2022).

In order to achieve our goal, we have to exploit open source tools and the expertise in our Company's area to obtain a new architecture featuring the same functions provided by LOGGIT since we can no longer have a service exposing sensitive data.

Our methodology is to move towards our new solution in a step by step manner. More precisely, we plan to obtain our goal by successive iterations and we in this work we describe and explain our choices

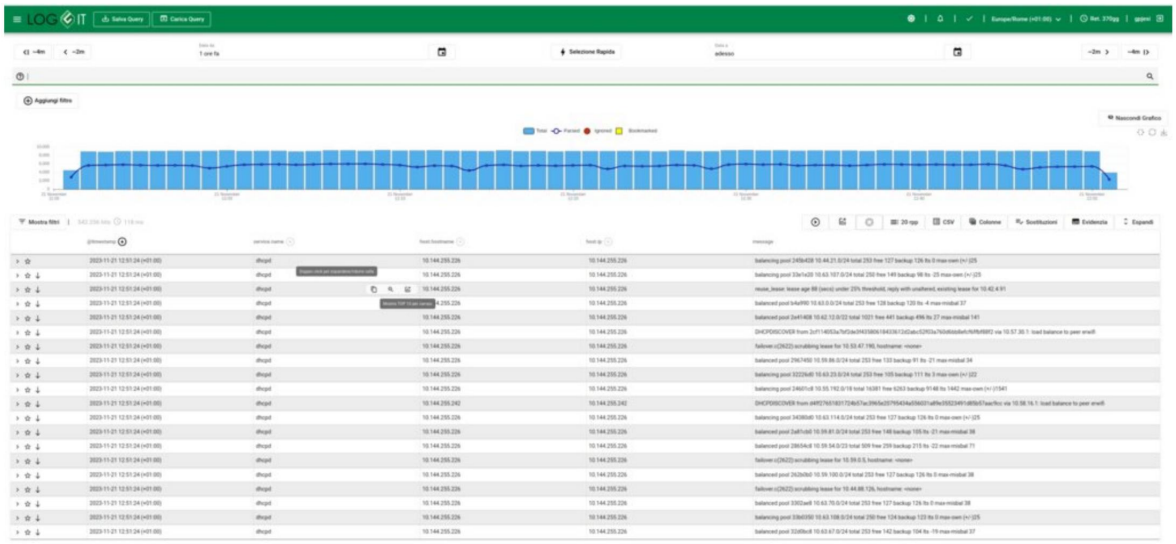

Figure 1: A LOGGIT GUI snapshot. LOGGIT is the tool we have to substitute in our pipeline. In addition to data ingestion, it allows to visualize and query ingested data.

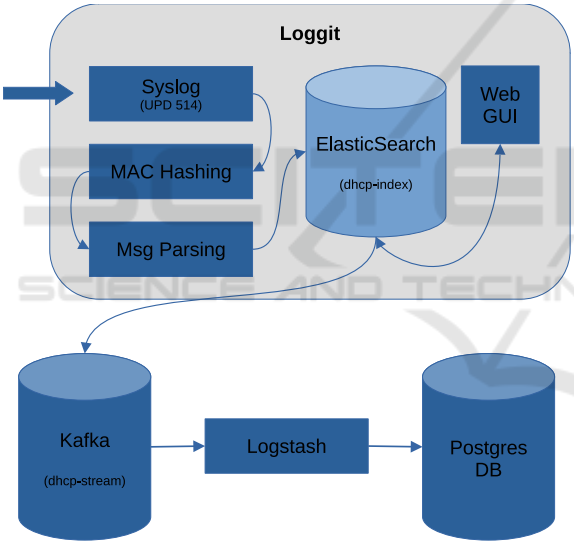

Figure 2: Block diagram of the previous architecture exploiting LOGGIT.

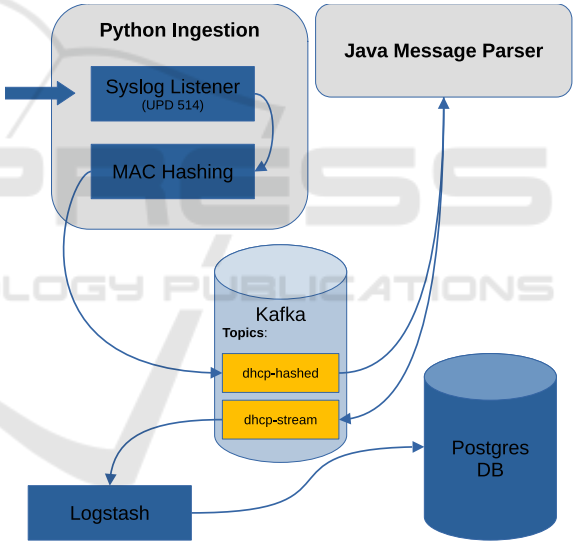

Figure 3: Block diagram of our first iteration of the architecture. Two custom in-house developed software have been adopted. The former performs pseudo-anonymization (e.g., hashing) over sensitive data, while the latter parses syslog message content.

during this path.

However, at the beginning, our main problem is time: we have to bypass LOGGIT without any downtime in the ingestion service as soon as possible.

3.1 Custom Architecture

Our first iteration of the refactored architecture is depicted in Figure 3. We labeled this attempt as "custom" because we wrote some custom software in order to switch from LOGGIT as soon as possible.

Here, a specific virtual machine substitute the one running LOGGIT and hosts a custom program written in Python language that listens to dhcp message log sent by the Network Department Syslog server. Each message is pseudo-anonymized by applying a hash function and then it is sent to a topic (e.g., "hdcp-hashed") on a Kafka queue.

Actually, we have to apply two distinct hash function to the sensitive information (i.e., MAC addresses). In fact, while LOGGIT was using a SHA-256 hash, we decided to adopt Fowler/Noll/Vo (FVN) (fvn, 2023; Sadeghi-Nasab and Rafe, 2022) algorithm

since it is better optimized (in space) for our application. In addition, SHA-256 was not in accordance with what it was written in our Privacy Statement which was actually referring explicitly to the FVN algorithm.

The reason to adopt Python here is twofold: (i) we are proficient with this language and (ii) an implementation of the FVN algorithm is easily available in Python.

The message parsing process is carried out by another ”custom” program written in Java. We have developed this program a few years ago during the early days of our BIG DATA area. Its adoption was the fastest route to have a new and working ingestion pipeline.

This program is a Kafka consumer that reads from ”hdcp-hashed” topic and writes to the ”dhcp-stream” one after having parsed each message. Both syslog and dhcp tokens are taken into account during parsing.

By keeping the ”dhcp-stream” topic as interface, the rest of the pipeline remains untouched.

This approach can substitute the previous tool and we achieved a working, production-ready pipeline in a few days (testing inclusive). Essentially, we accomplished our first goal to bypass the previous tool in a short amount of time. However, this approach has some weaknesses.

Fist, our Data protection Officer (DPO) pointed out that even if we have pseudo-anonymized MAC addresses, in the Kafka topics we still have data that have not been declared into the Privacy Statement and we cannot modify it[3].

Second, the use of custom software might not be as reliable as exploiting a specific tool tailored for this task. In addition, the lack of a monitoring tool of any kind would likely let us discover any issues with an unmanageable delay.

3.2 All in One Architecture

In order to cope with the issues of the first architecture, we designed and implemented the second one which we called ”all in one”. A block diagram of this solution is depicted in Figure 4.

Here, the idea is to avoid to write on stable storage - on the Kafka queue in this case - something that is not declared in the Privacy Statement and that can be considered a bad practice of any kind.

Essentially, we have to perform MAC hashing and message parsing in s single step and writing the final result to the ”dhcp-stream” Kafka topic.

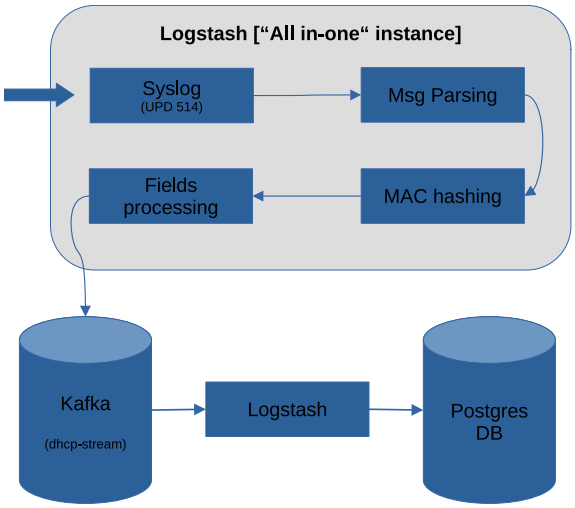

Figure 4: Block diagram of the All in one architecture: MAC hashing and message parsing have been condensed in a single step. In contrast to the previous solution, a Logstash instance has been adopted as tool instead of custom written software.

Instead of using our custom software, we choose a different route and we adopted a Logstash (Mitra and Sy, 2016) instance. We have some experience in using LogStash, which is part of the Elastic Stack framework (Zamfir et al., 2019), but the FVN hashing function is not managed by any of its huge library of plugins. This is what had stopped us from using it at a first stance.

Unfortunately, we have no experience with Ruby language and its ecosystem, but diving into Ruby code is the only way to add extra features outside the ones provided by standard plugins in Logstash distribution. However, we invested a little time and it turned out to be easier than expected.

Figure 4 shows some details of the adopted Logstash instance. Its input plugin listens (over UPD port 514) for syslog messages. Upon arrival, each message is first parsed and then hashed. In contrast to the previous solution, here we have actually inverted the sequence hashing-parsing. In fact, parsing and then hashing is more efficient.

This is due to the fact that the message parsing process follows a set of specific standard rules and each token of the log string is mapped to a specific field name. In this manner, for example, we know that the MAC address will be in a `mac_address` field. In order to hash it, we do not have to search it inside the raw message string as we would do without parsing first.

As already stated we have to hash in two distinct manner, since we need: (a) SHA-256 for compati-

[3]Since it has been recently modified and, at the time of writing, it is a *pending-approval* state by the Data Protection Authority.

bility with the data collected so far and (b) FVN algorithm for better resource utilization (Zhuoyu and Yongzhen, 2022). According to Privacy regulations, we can keep our pseudo-anonymized (hashed) data for 1 year. This means that we have to keep this double hashing for 1 year as well.

We provided Logstash with an open source Ruby implementation of FVN algorithm and we instructed to calculate the hash in a specific field (e.g., `fowler_hash`).

Finally, some processing is applied to the fields. This includes the following: (a) field renaming to maintain compatibility with the JSON structure adopted since the LOGGIT era, (b) adding a `uuid` field, (c) pruning any field that can no longer be exposed (e.g., such as the gateway, or DHCP server IPs or names).

Messages are then pushed downwards to the Kafka queue in form of JSON object such as the following:

```
{
  "ts": "2023-09-18T04:25:23.000Z",
  "dhcpd": {
    "type": "REQUEST",
    "lease": {
      "ip": "10.51.9.150",
      "mac_address": "c203bba34..."
    }
  },
  "uuid": "b3fa8f2f-a596-4d56-...",
  "epoch": 1695011123000,
  "fowler_hash": "ee5b7eb2eae3cedd"
}
```

This architecture solved the Privacy concerns and we believe it is potentially more reliable than our custom software. In fact, it has being grounded on a reputable software - Logstash - which has been carefully configured for our specific task.

However, this approach raises another issue with which any enterprise system has to deal with. Having condensed so many features in a monolithic component, this raises a problem when any maintenance operation is required. In fact, any restart of the virtual machine (e.g., for security updates), any minor or substantial change to the process configuration or bug fix implies shutting down the service and loosing incoming data.

This inconvenience requires a solution which we describe in the following section.

3.3 Split Architecture

We considered several options in order to make the ingestion less brittle. The common denominator among these is to *split* the previous Logstash process into two distinct processes. We are going to call them respectively STEP-A and STEP-B. Each process is hosted on a different virtual machine. We considered the following two options:

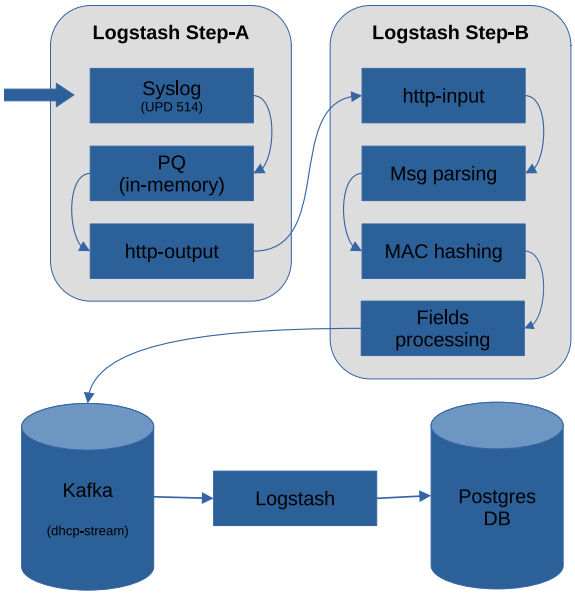

Figure 5: Block diagram of our split architecture: the previous monolithic Logstash component has been split into two Logstash instancies, respectively: STEP-A and STEP-B, running on distinct virtual machines. STEP-A is basically a "pipe" with a buffer, while STEP-B performs a processing over the data.

- STEP-A listens for syslog messages as usual and delivers them to STEP-B via an *http-output* plug-in. This plug-in provides a delivery feedback and thus STEP-A can detect if STEP-B goes offline. In this case, STEP-A still collects data using its Persistent Queue (PQ[4]) on *stable storage*. STEP-A does not do any processing, it is like a *pipe*, but with a persistent buffer.

  In this manner, we would obtain two advantages: (i) in case of STEP-B failure, when it returns online, the fact will be detected by STEP-A and it will restart sending messages (i.e., nothing is lost), (ii) in case STEP-A fails, its PQ ensures that anything not yet delivered to STEP-B is not lost.

  However, the drawback of this solution is that the PQ is in clear text and there is not even a chance to encrypt it since the PQ is just after the receive message phase and before any processing (i.e., "filtering" in Logstash jargon).

- Instead of using the PQ, we exploit a Kafka topic to implement our own persistent queue fla-

[4]The Persistent Queue is a Logstash feature.

vor. This time, STEP-A encrypts the raw message string. Conversely, STEP-B reads from the topic, decrypts each message and apply the processing as in the all-in-one architecture case. In this scenario, AES-256 algorithm would be adopted for encryption/decryption and a strong random password would be shared between the processes using the Logstash key-store service.

In this manner, we would obtain the same features as the previous option, with the extra benefit of securing any sensitive information. The extra complexity due to the requirement of Kafka is not an issue, since its resilience - when properly deployed as a cluster - is much higher than a Logstash instance.

Unfortunately, after testing the feasibility of this approach, our DPO warned us that we cannot do that because this encryption mechanism and its presence on our storage is something which is not declared in our Privacy Statement.

Essentially, it seems that, even if we encrypt whatever persistent queue we choose, any processing that is written on a persistent storage is a data treatment and - at least - it must be declared.

Therefore, we have to relax our goal and come up with a compromise solution which is depicted in Figure 5.

We still have two instancies of Logstash, respectively: STEP-A and STEP-B, running on distinct virtual machines. The former listens for syslog messages and delivers them to STEP-B via an *http-output* plugin, exactly as in the previous first option. However, this time we use the Logstash PQ in a relaxed fashion: using the *in-memory* queue type. As the type suggests, the queue is no more really persistent, but volatile since it is kept in RAM memory. If STEP-A fails, *any non delivered message is lost*. If STEP-B fails instead, nothing is lost: STEP-A buffers messages and will deliver them to STEP-B when it will eventually return online.

STEP-A does not do any processing, it is like a *pipe*, but with a (volatile) buffer. Its simplicity implies that the probability to have to fix it or to change something is low. Instead, STEP-B is more likely to need modifications and updates over time. Maintenance does not imply losing data in this case.

It is worth noting that the PQ in memory is working with no encryption at all. This approach is perfectly fine for the Data Protection Authority point of view as long as data are kept in memory and accessible to just a software process with no practical chances to be exposed or extracted.

Unfortunately, none of our approaches provides a UI (see Figure 1) with details about the ingested data. Also our last architecture behaves like as ort of black box: we can just see the events flowing on the Kafka topic. However, while we do not have a GUI yet, we have activated the monitoring features of Logstash which allows to collect runtime metrics about nodes and plugins via a REST API. In this manner, it is possible to track the pipeline behavior by triggering a REST call by configuring, for example, an external tool such as CheckMK (che, 2023).

4 STATE OF THE ART

Unfortunately, it seems that there is not much knowledge about the process of refactoring a BIG DATA pipeline in literature in terms of patterns and best practices to choose from.

Refactoring areas can be restricted to the five (Peruma et al., 2021) following topics: (i) code, (ii) tools or IDEs, (iii) architecture and design patterns, (iv) unit testing and (v) database. However, most of the literature focuses on the first one (e.g., code refactoring).

In (Peruma et al., 2021), the authors highlight the need for bridging the gap between refactoring, as research, and its adoption in practice, by extracting common refactoring intents that are more suitable for what developers face in reality.

An interesting study (AlOmar et al., 2021) from Xerox Corporation aims to reveal insights into how reviewers develop a decision about accepting or rejecting a submitted refactoring (code) request, and what makes such review challenging. They present an industrial case study where they analyze the motivations, documentation practices, challenges, verification, and implications of refactoring activities during code review. The results report the lack of a proper procedure to follow by developers when documenting their refactorings for review. As a countermeasure for their findings, they designed a specific procedure to properly document refactoring activities.

In our case, since our refactoring regards a specific ingestion process, we could not find any specific guideline to follow and we could not even provide any general one for the same reason. Our contribution is to provide a detailed use case scenario and its solution.

5 DISCUSSION AND CONCLUSIONS

In this paper we discussed the reasons why we have to refactor our ingestion pipeline. Unfortunately, pri-

vacy concerns forced us to abandon the LOGGIT software which was a cornerstone in our pipeline architecture. We did our best to comply with this requirement as soon as possible.

The aim of our contribution is to provide a real-world scenario or use case which might be useful for other companies when dealing with similar issues.

We iteratively refactored our pipeline architecture several times in order to find a balance between our requirements in terms of reliability and the sometimes convoluted rules of the Data Protection Authority.

We discussed each iteration and the choices we made. Our pipeline has been up and running since the first iteration and we had no data loss due to the switch.

While we are still missing the visualization feature of the previous tool (LOGGIT), we managed to provide a basic monitoring facility. In order restore the previous web interface facility, our basic plan is to index the data stream into an Elastic stack (e.g., OpenSearch + OpenDashboard) and replicate the LOGGIT visualizations. We consider the addition of a monitoring and visualization interface as our next future goal.

REFERENCES

(2022). LepidaScpA Home Page.

(2023). Checkmk - An All-in-One, open source IT monitoring solution.

(2023). Data protection in the EU.

(2023). The FNV Non-Cryptographic Hash Algorithm.

AlOmar, E. A., AlRubaye, H., Mkaouer, M. W., Ouni, A., and Kessentini, M. (2021). Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357.

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. (2015). Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 33rd ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM.

Camacho-Rodríguez, J., Chauhan, A., Gates, A., Koifman, E., O'Malley, O., Garg, V., Haindrich, Z., Shelukhin, S., Jayachandran, P., Seth, S., Jaiswal, D., Bouguerra, S., Bangarwa, N., Hariappan, S., Agarwal, A., Dere, J., Dai, D., Nair, T., Dembla, N., Vijayaraghavan, G., and Hagleitner, G. (2019). Apache hive: From mapreduce to enterprise-grade big data warehousing.

Fuller, M., Moser, M., and Traverso, M. (2022). *Trino: The Definitive Guide, 2nd Edition*. O'Reilly Media, Inc.

Hiraman, B. R., Viresh M., C., and Abhijeet C., K. (2018). A study of apache kafka in big data stream processing. In *2018 International Conference on Information , Communication, Engineering and Technology (ICICET)*, pages 1–3.

Mitra, M. and Sy, D. (2016). The rise of elastic stack.

Peruma, A., Simmons, S., Alomar, E. A., Newman, C. D., Mkaouer, M. W., and Ouni, A. (2021). How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. *Empirical Software Engineering*, 27.

Sadeghi-Nasab, A. and Rafe, V. (2022). A comprehensive review of the security flaws of hashing algorithms. *Journal of Computer Virology and Hacking Techniques*, 19:1–16.

Shaikh, E., Mohiuddin, I., Alufaisan, Y., and Nahvi, I. (2019). Apache spark: A big data processing engine. In *2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM)*, pages 1–6.

Zamfir, A.-V., Carabas, M., Carabas, C., and Tapus, N. (2019). Systems monitoring and big data analysis using the elasticsearch system. pages 188–193.

Zhuoyu, H. and Yongzhen, L. (2022). Design and implementation of efficient hash functions. In *2022 IEEE 2nd International Conference on Power, Electronics and Computer Applications (ICPECA)*, pages 1240–1243.